View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digital Commons @ New Haven

University of New Haven

University of New Haven
Digital Commons @ New Haven

Electrical & Computer Engineering and Computer Science Faculty Publications

Electrical & Computer Engineering and Computer Science

2018

If I Had a Million Cryptos: Cryptowallet Application Analysis and A Trojan Proof-of-Concept

Trevor Haigh
University of New Haven

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

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

Follow this and additional works at: https://digitalcommons.newhaven.edu/electricalcomputerengineering-facpubs

Part of the Computer Engineering Commons, Electrical and Computer Engineering Commons, Forensic Science and Technology Commons, and the Information Security Commons

Publisher Citation

Haigh, T., Breitinger, F., Baggili, I. (2018) If I Had a Million Cryptos: Cryptowallet Application Analysis and A Trojan Proof-of-Concept. In Digital Forensics & Cyber Crime: 10th International Conference, ICDF2C, September 10-12, 2018, New Orleans, Revised Selected Papers. Springer.

Comments

This is the authors' accepted version of the paper published in *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (LNICST)*. The volume encompasses the proceedings of the 10th EAI International Conference on Digital Forensics & Cyber Crime, September 10-12 in New Orleans. The version of record for the proceedings volume may be purchased from the Springer web site.
The final authenticated version is available online at https://link.springer.com/conference/icdf2c
Dr. Baggili was appointed to the University of New Haven's Elder Family Endowed Chair in 2015.

If I Had a Million Cryptos: Cryptowallet Application Analysis and A Trojan Proof-of-Concept

Trevor Haigh, Frank Breitinger and Ibrahim Baggili
Tagliatela College of Engineering, ECECS Department
The University of New Haven, West Haven, CT

Abstract. Cryptocurrencies have gained wide adoption by enthusiasts and investors. In this work, we examine seven different Android cryptowallet applications for forensic artifacts, but we also assess their security against tampering and reverse engineering. Some of the biggest benefits of cryptocurrency is its security and relative anonymity. For this reason it is vital that wallet applications share the same properties. Our work, however, indicates that this is not the case. Five of the seven applications we tested do not implement basic security measures against reverse engineering. Three of the applications stored sensitive information, like wallet private keys, insecurely and one was able to be decrypted with some effort. One of the applications did not require root access to retrieve the data. We were also able to implement a proof-of-concept trojan which exemplifies how a malicious actor may exploit the lack of security in these applications and exfiltrate user data and cryptocurrency.

Key words: Cryptowallet, Cryptocurrency, Bitcoin, Coinbase, Android

1 Introduction

The popularity of cryptocurrencies like Bitcoin and Ethereum exploded in 2017; more and more people are buying into digital currencies. Managing digital currencies requires software to buy, sell and transfer digital coins. This software is commonly known as a cryptowallet and is available for all major mobile platforms or online. Similar to a bank account, it is essential that these wallets are secure which is not always the case. For instance, in December 2017 "thieves stole potentially millions of dollars in bitcoin in a hacking attack on a cryptocurrency company [named NiceHash]. The hack affected NiceHash's payment system, and the entire contents of the company's bitcoin wallet was stolen" [14]. While this is certainly one of the more significant incidents that has happened with approximately $60 million dollars stolen, there are other examples, e.g., "an unidentified thief has reportedly stolen more than $400,000 in Stellar Lumens after hacking the digital wallet provider BlackWallet" [10]. An overview of the top 5 hacks is provided by [7].

Besides online cryptowallets, there are also offline applications that one can install on personal devices. For instance, ikream.com[1] lists the 5 best crypowal-

[1] https://www.ikream.com/2018/01/5-best-bitcoin-wallet-mac-os-x-26068 (last accessed 2018-05-08).

lets for Mac OS X: Electrum, Exodus, Jaxx, Coinbase and Trezor. Most of these applications also have a Windows version and are available for Android as well as iOS.

In this paper we examine seven cryptowallet applications for Android – Coinbase, Bitcoin Wallet, Xapo, Mycelium, Bitpay, and Coinpayments. The goal of this research is to discover forensic artifacts created by applications through the analysis of persistent data stored on the device as well as through application analysis (source code analysis) of each application. Our work provides the following contributions:

- The robustness assessments showed that several application use outdated security practices and thus are not protected against reverse engineering and tampering.
- The artifact analysis showed that several applications store sensitive information (e.g., private keys) as well as unencrypted data (e.g., passwords) on the device.
- We describe a proof-of-concept Trojan attack that exploits weaknesses found in one of the applications.

The structure of this paper is as follows: The upcoming Section 2 discusses previous and related work. The methodology is presented in Section 3. The heart of this paper is Section 4 which presents our findings followed by Section 5 which presents the proof-of-concept trojan. The last sections highlight future work and conclude the paper.

2 Related Work

In the related work section we first briefly present the technologies behind Cryptocurrency followed by the state of the art of Android artifact acquisition in Section 2.2. In Section 2.3 we summarize the literature for reverse engineering Android applications. The last section briefly discusses cryptowallet analysis.

2.1 Blockchain Technology

The blockchain is a decentralized, public ledger of all transactions that have been executed [17]. Because of its distributed and public nature, users do not have to put trust in a third-party (e.g. a bank). Anyone can download the entire blockchain and sites like `https://blockchain.info/` allow people to search the Bitcoin blockchain for specific transactions.

Blockchain relies on encryption in order to function where transactions are performed using public/private key pairs to verify if they are authorized by the owner of the wallet [13]. The biggest security challenge of a wallet application is how the keys are managed. If private keys are stored insecurely, then attackers can obtain them and thus steal all of the currency associated with that wallet. Because of this, wallet applications are almost always the focus of attacks

rather than the blockchain itself. In fact, it is practically impossible to attack the blockchain directly because in order to steal one Bitcoin, its entire history would have to be rewritten on the publicly viewable blockchain [18].

The way wallets are managed differs depending on the wallet itself. Some applications (e.g., Bitcoin Wallet[2]) store the information only on the physical device. If the device is lost or damaged, there is no way to recover any currency associated with these wallets (in this case it is highly recommended to have backups). Other applications transfer the information to online servers, like Coinbase[3]. In this case, users access their wallets via a username and password created for the application. Another approach is seen in applications like Electrum[4] which stores the wallet on the device but then allows for the recovery of the wallet via a series of random words that was used to generate the keys.

2.2 Android Artifact Acquisition

Much research has been conducted on Android artifact analysis and therefore we selected some from different application categories to provide an overview. For instance, [20] focused on the analysis for social media applications like oovoo and was able to show that many do not use encryption. They were able to recover images as well as messages. [9] and [12] focused on GPS geodata from various applications on an Android device and demonstrated that applications like Waze provide a lot of helpful information for investigators. Other work focused on extracting media files from encrypted vault applications [21] where the Authors found significant vulnerabilities in 16 Vault applications.

While applications are very different, the approach for obtaining data the researchers took was similar. In referenced work, the researchers created logical copies of the data using the Android Debug Bridge[5] (ADB). ADB is a command line tool for Android that allows for the transfer of files to and from a connected Android device to a forensic workstation via a USB connection or Wi-Fi (workstation and Android device need to be on the same Wi-Fi). The data that can be pulled with ADB is limited by the permission level. For instance, for an unrooted phone, one can only access files viewable by the user, such as Documents, Downloads, or files in external storage. To access application specific data, ADB must have root access (the phone needs to be rooted). Thus, having root access is generally required to access all (most useful) data, such as application folders or stored preference files. While there are many methods to obtain root access on a device, such as those described by [8], they vary wildly depending on the device, Android version, and carrier. Generally, newer devices are becoming more and more difficult to obtain root access on.

Another method to obtain application data without root permission is to backup the application data using ADB. The backup can then be decompressed

[2] https://wallet.bitcoin.com/ (last accessed 2018-05-08).
[3] https://www.coinbase.com/?locale=en-US (last accessed 2018-05-08).
[4] https://electrum.org/ (last accessed 2018-05-08).
[5] https://developer.android.com/studio/command-line/adb.html (last accessed 2018-05-08).

and analyzed on the forensic workstation [4]. This method still requires physical access to the device, though, in order to obtain the backup and application developers can prevent their application data from being backed up via a flag in the application's manifest file.

While most methods, like ADB, utilize software to recover artifacts, there are also hardware solutions. For instance, one solution is to use specialized equipment to detach the flash memory chip from the device's PCB [6]. Naturally, this is a much more invasive process and the device is very unlikely to be restored to its original condition.

2.3 Reverse Engineering Android Applications

Reverse engineering of Android Applications can be done both statically and dynamically. Most work is done by statically analyzing application code. This is an easier process to automate as it does not require an environment in which the application has to execute in. [1] proposed an automated static analyzer that inspects seemingly harmless applications for malicious functions.

Static analysis is usually performed using tools such as Apktool[6] and Dex2Jar[7] to decompile the code into Smali or Java, respectively. Smali is an 'assembly-like' language for Dalvik, Android's Java VM implementation. In most cases the translation from dex to Java is not always perfect so an understanding of Smali is necessary to get full knowledge of the code.

The smali code and other files (such as the manifest or shared preferences) can be modified to change the operation of the application. Many third party application markets contain legitimate applications that were modified to to add malware or advertisements [22]. Static analysis can be used to detect these modified applications. [16] presented a machine learning algorithm that uses static analysis to classify Android applications and detect malware.

Dynamic analysis is performed by running the application on the device and hooking into the process with a debugging tool, such as Android Studio[8] with the smalidea plugin[9]. Dynamic analysis has the advantage of being able to set breakpoints within the code as well as view/manipulate variable values. The downside is that not all code paths may execute. Some applications may also employ methods to hinder dynamic analysis attempts [15] and change their behavior accordingly. Unlike static analysis, dynamic analysis is more difficult to automate; some sample tools that support the automation of dynamic analysis are TaintDroid [5] or Crowdroid [2]. However, in many cases dynamic analysis still has to largely be conducted manually for each application.

[6] https://ibotpeaches.github.io/Apktool/ (last accessed 2018-05-08).
[7] https://sourceforge.net/projects/dex2jar/ (last accessed 2018-05-08).
[8] https://developer.android.com/studio/index.html (last accessed 2018-05-08).
[9] https://github.com/JesusFreke/smali/wiki/smalidea (last accessed 2018-05-08).

2.4 Cryptowallet Application Analysis

Most research into the analysis of cryptowallet applications has been performed on the desktop (Windows) environment. [3] analyzed a machine running two wallet applications as well as a Bitcoin mining application and was able to recover evidentiary artifacts linking Bitcoind transactions to that machine. Similarly, [19] analyzed the process memory of two other wallet applications and were able to, in certain cases, recover private keys and seeds allowing seizure of funds from the wallets.

Limited research has been conducted regarding cryptowallet applications in the mobile space. [11] checked for forensic artifacts in four different applications for Android and iPhone, Android being the most relevant as far as this paper is concerned. The applications were tested on both an emulator and a physical device. They discovered some forensic artifacts leftover from cryptocurrency transactions made with those wallet applications. Cryptocurrency has exploded since this research was completed, so there are many new applications that have become popular and were not examined in the previous work. Furthermore, the researcher was also solely focused on finding forensic evidence of the wallets rather than also evaluating the security of the applications.

3 Methodology

In this section, we discuss how each application was chosen as well as how the methods used to analyze each application and its data. For testing, we utilized a Samsung Galaxy S3 Active running Android version 4.4 (KitKat) and a laptop running Linux to retrieve and analyze the data.

3.1 Application Setup

For this work, we focused on the Android operating system. Next, we chose applications based on the number of downloads. As a result, we analyzed the seven applications listed in Table 1.

Table 1: Chosen Android cryptocurrency applications ordered by number of downloads.

	Application	Package Name	Downloads	Version
1	Coinbase	com.coinbase.android	5m+	5.0.5
2	Binance	com.binance.dev	1m+	1.4.5.0
3	Bitcoin Wallet	de.schildbach.wallet	1m+	6.23
4	Xapo	com.xapo	1m+	4.4.1
5	Mycelium	com.mycelium.wallet	500k+	2.9.12.2
6	Bitpay	com.bitpay.wallet	100k+	3.15.2
7	Coinpayments	net.coinpayments.coinpaymentsapp	50k+	1.0.0.6

After downloading each application to the device we ran and set up each one. All of the applications were installed at the same time. While most applications did not require to create a user account and manage the wallet locally, Coinbase and Xapo forced us to register an account as information is stored online.

3.2 Data Acquisition

Before acquiring the data from the phone, we started each application and performed basic setup operations such as creating an account for applications that required it or setting up passcodes to access the wallet. Next, we utilized ADB to download the data (the APK file and the Application data folder) for each application.

APK File is the application itself and can be found in `/data/app/<package_name>` (where `<package_name>` is the package name of the application as listed in Table 1). The application is important because it contains the compiled source code and other resources. We needed the APK file in order to perform code decompilation and analysis.

Data Folders can be found in `/data/data/<package_ name>`. This folder is created for each application and is usually only accessible by the application itself. Inside the folder one can find items such as settings files and databases. Any existing artifacts would most likely be found in these folders.

Note, to gain access to these folders and to pull the data, the phone has to be rooted. In order to automate the process we created a Python script that utilized monkeyrunner[10] and ADB to pull the data folders for each application. Monkeyrunner is an API that allows a program to control an Android device or emulator. By using monkeyrunner, we were able to automate performing shell commands on the Android device. The process of pulling the data was accomplished in two steps.

Step 1: The data is copied from `/data/data` to `/sdcard/data` using the command `su -c cp -R /data/data/<package_name> /sdcard/data/<package_name>`. This is done because 'sdcard' can be accessed without root access, unlike 'data'.

Step 2: The data is pulled from the device using the command `adb pull /sdcard/data/<package_name>`. This is called in Python using the subprocess package. The command copies the data folders from the device to the forensic workstation. After this is complete, the script cleans up by deleting `/sdcard/data/<package_name>`.

The code for this script can be found in Listing 5 in Appendix A.

[10] https://developer.android.com/studio/test/monkeyrunner/index.html (last accessed 2018-05-08).

3.3 Creating Transactions

In order to populate the applications with artifacts, transactions were performed for / from each application. Specifically, we purchased 0.01255964 BTC ($100 worth at the time of purchase) and passed it from one application to another. This ensured that each application has at least one incoming and one outgoing transaction logged.

3.4 Analysis

After extracting the data from the device, we analyzed both the artifacts and the application source code where the primary focus was on the artifacts. Code analysis was done to assess the general security of the applications.

Artifact Analysis was performed manually on the files extracted from the device. Notable files were XML preference files and database files where we especially focused on XML files (viewed using a text editor) and the SQLite database files (viewed using a SQLite database browser[11]). When analyzing the extracted data, we focused on finding the following items:

Wallet Private Keys are probably the most sensitive and critical artifacts managed by these applications as the entire purpose of the application is to manage and store the private key(s) securely. Finding the private key is essentially the 'golden ticket' as it can be used to siphon all funds from the wallet. Therefore, being able to obtain the private keys is a sure sign of an insecure application.

Wallet Seed is similar to the private keys in that they both lead to direct control of the wallet. Many wallet managers use a list of words, or seed, to generate the key pair. This exists as a recovery mechanism and the idea is that a user would write down the list of words in a secure place and use them to recover the wallet by generating the key pair again. Because of this, if we are able to obtain the wallet seed, we could recover the wallet ourselves and control the funds.

Transaction History can be important from a forensic investigation point of view. While the blockchain is public, it may be difficult to find exactly where the transactions related to the suspect are located. Being able to pull transaction history directly from the wallet application, without needing the login credentials could be a great boon in an investigation. For this reason, we focused on trying to pull as much transaction information as possible.

Application-specific data including passwords, PINs, etc. should be managed securely in any application, especially in ones that manage money. An application could store and manage the keys securely but if they store the user's login credentials in plaintext, for example, then we can simply use that to log into an account, barring any two-factor authentication.

[11] http://sqlitebrowser.org (last accessed 2018-05-08).

Code Analysis was performed to assess the general security of the applications by looking for common reverse engineering countermeasures. Specifically, we looked for the following three properties:

Code obfuscation is the act of purposely making your code difficult to read and understand. For Android applications this means replacing class and variable names with 1-3 letter names (e.g., 'aaa', 'aab', etc.) which is always done using software. For instance, Google's Android Studio has built-in options to obfuscate code using Proguard[12,13]. There are also commercial obfuscation tools that such as Dexguard[14] that come with more capabilities.
Code obfuscation is important because it significantly slows down reverse engineering attempts. Heavily obfuscated code is difficult and time-consuming to navigate so reverse engineers are less likely to find sensitive information in the code. It should be noted that obfuscation does not inherently prevent reverse engineering, however, anything that makes it more difficult is a good security measure to take. Code obfuscation was tested by decompiling each application using JEB[15] and viewing the code.

Signature verification is a method of ensuring an application has not been tampered by a third party. Before Android applications are installed on a device, they must be cryptographically signed by a developer. Any modifications made to the application would also change the signature (assuming the attacker does not have access to the developer's private key). A common security practice is to verify that the installed application's signature matches the signature of the release version and disallow any operations if it does not.
Without this step, applications are vulnerable to modifications with malicious code. An example of this is provided later in the paper where we created a modified version of Coinbase that steals users' credentials (see Sec. 5).
Signature verification was tested by decompiling each application with Apktool, recompiling it, and signing it with our own key. The recompiled application was then installed on the device using ADB and executed to ensure it functions normally. Since we do not have access to the developers' keys, our signed APKs will have a different signature than the official versions.

Installer verification ensures that the application was installed from a legitimate source. When installed, each application records the package name of the application that installed it (e.g., com.android.vending is the package name for the Google Play Store). Because a malicious version of the application could not be installed from the Google Play Store, it must be installed from another source. Assuming the legitimate version is only distributed on the Play Store, the application can be made to only function if the Google Play Store is its installer. This security method is generally uncommon as it prevents the

[12] https://www.guardsquare.com/en/proguard (last accessed 2018-05-08).
[13] Note, Proguard is mostly used to minimize and optimize code and offers minimal protection against reverse engineering.
[14] https://www.guardsquare.com/en/dexguard (last accessed 2018-05-08).
[15] https://www.pnfsoftware.com (last accessed 2018-05-08).

'sideloading' of applications which may be a legitimate method of obtaining the application.
The installer verification was tested the same way as 'signature verification', by redeploying the application to the phone and executing it. Any installer verification should fail as the applications' installer was ADB and not the Play Store.

3.5 Manipulation of an Application

The smali code of an application can be edited to perform a wide array of actions such as removing a pay wall or enabling additional features. A more nefarious option is to include malware in the application that sends user information to a remote server. Without any of the security methods mentioned above, an altered application would function normally and the user would be unaware of the malware on their device. To illustrate how an attack like this works, we constructed a proof-of-concept trojan version of Coinbase.

Note, the biggest challenge of these types of attacks is actually distributing the malware. Without the developer's private key, an attacker cannot upload a malicious version of the application to the Play Store (all applications need to be signed). If this signature is not valid, then Google will reject the upload. Thus, the only way to get malware installed is by social engineering users into 'side-loading'[16] the application which was not part of this research paper.

4 Findings

In this section, we discuss findings for each application as well as assessing whether they implement any of the security features mentioned previously.

4.1 Coinbase

Coinbase the most popular application we analyzed with over 5 million downloads which uses the cloud; keeps most of the wallet data, including the private keys, on their servers. This means that the security of the wallet is primarily dependent on their server security, rather than the physical device storage.

Focusing on the application revealed that Coinbase is not obfuscated. This made modifying the code quite trivial as we show later in our trojan proof of concept (see Section 5). The application also does not implement any signature or installer verification which allowed us to resign the APK, install it with ADB, and run it without any issues. Additional findings:

[16] Side-loading is installing an application directly rather than through a market. This usually requires an additional option to be enabled on the device before the OS will allow the installation.

Plaintext Password was found in the `shared_prefs` XML file which contains various options and preferences for the application. The account password only seems to exist in the preferences if the account is created on the device. If the application is uninstalled and reinstalled, or one signs in to an existing account rather than creating a new one, then the password is no longer shown.

PIN Enabled is a boolean variable that also exists in the shared preferences. This is critical as changing the value from 'true' to 'false' disables the PIN. However it requires a text editor with root access installed on the device, so it may not be a practical attack vector.

Transaction and Account Databases were found containing data items such as transaction amounts, account ids, and account balances. After performing some transactions, the database was populated with plaintext data so even if you cannot get into the account, you can still view the full transaction history. Note, this information is also stored in the blockchain but would require an investigator to know the public key of the wallet (or the transaction ID).

4.2 Binance

Like Coinbase, Binance does not store sensitive wallet information on the device but on their servers, e.g., Binance users can access their wallets from any device through their website or mobile application. Unlike Coinbase, though, Binance does not store any transaction information on the physical device. This was confirmed by testing the application without Internet access. It fails to retrieve any transaction history or wallet information indicating that the information is pulled from the server on the fly rather than stored on the local device.

Code wise, Binance is obfuscated using Proguard. The application verifies its signature and crashes when trying to open an incorrectly signed version. There is no installer verification.

4.3 Bitcoin Wallet

Bitcoin Wallet is built on Bitcoinj[17], an open source, Java Bitcoin implementation that aids the creation and management of Bitcoin wallets. Bitcoinj uses Google's protocol buffer to serialize the wallet data. Thus, it is trivial to read the wallet data using custom or pre-existing software, e.g., wallet-tool. Furthermore, Bitcoinj includes tools to dump wallet data which we utilized to view the data.

By using Bitcoinj's *wallet-tool*, we were able to find the private keys and seed associated with Bitcoin wallet as well as the complete transaction history. Note, Bitcoinj does have the option of encrypting the wallet with a password, but Bitcoin Wallet does not implement this feature.

[17] https://bitcoinj.github.io/ (last accessed 2018-05-08).

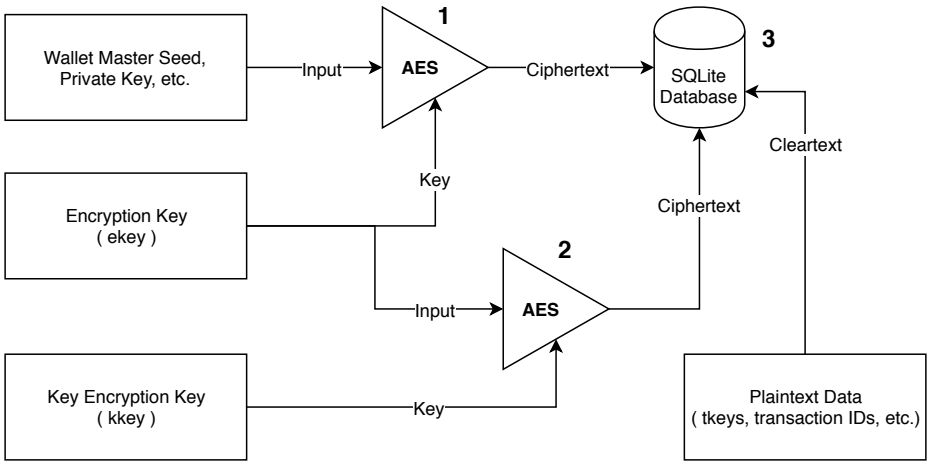

Fig. 1: Current work flow for storing information in mycelium.

4.4 Mycelium

Mycelium stores its transaction data in a SQLite database. Unfortunately traditional SQLBrowsers were fruitless as most of the data is stored in binary. However, the structure of the table looks like it is storing key-value pairs, i.e., a table with two columns where one is the key (*tkey* in the following) and two is the value. To pull out the data from the database, we implemented a Python script and converted the data into different encodings (e.g., string, hex) until we realized that most of the data in the value column is encrypted. To identify what was stored in the database, we analyzed the code that stores/reads from the database and following it backwards until the *tkey* for the desired value was found. Our analysis revealed that besides encrypted data, it also contains unencrypted data. For instance, the transaction ID was found in cleartext which can be used to look up the transaction on a Bitcoin blockchain explorer. Additionally, we found that AES encryption was used.

For the data that is encrypted, an overview is depicted in Figure 1. Mycelium uses a randomly generated encryption key named *ekey* to encrypted sensitive strings (e.g., wallet master seed, private keys). The encrypted information is stored in the database. The *ekey* is then encrypted using *kkey* and stored in the database. While normally *kkey* should be generated from a user password, the developers use a hardcoded string to generate this key.

Using this method, we determined that the *tkey* for the *ekey* was a single byte ('00'). It was also the first entry in the table. We continued using this method to find that the table contained other sensitive information, e.g., private keys or wallet master seed.

In order to decipher the data, we modified the open-source code to decrypt the data. In detail: we created a Java project using Mycelium's encryption classes as well as the class for handling master seed creation. Our own class was then

created which reads from the SQLite database and called the necessary Mycelium functions to generate the default encryption key and decrypt the data. The MasterSeed class has a function to generate a seed from the bytes decrypted from the database. The resulting object contained the seed words which could be used to recreate the wallet on another device.

It is noteworthy that the application has a method of preventing this attack, by generating the key encryption key with a user-provided password rather than one generated from a hardcoded string. This password feature exists in the code, but is not used in the current official version of the application. Once this changes, it will be impossible to decrypt the content (if the user has a strong password that hinder bruteforce attacks).

4.5 Xapo

Like Coinbase and Binance, Xapo also stores the wallet private keys in the cloud and not on the physical device. It does, however, store a plaintext database containing transaction information, much like Coinbase.

As far as the code goes, Xapo was obfuscated to the point where Apktool could not decompile it without errors. JEB was able to view the code, but was not able to write the files without error. Because of this, we could not recompile and resign the application to check for signature verification.

4.6 Bitpay

Bitpay differs from the other applications mentioned so far in that it was developed using Cordova. Cordova is a platform that allows for the development of mobile applications using JavaScript and HTML. The weakness of this platform is that the source code is included in the APK file so it can be viewed by simply unzipping the APK file; no particular software needed. The source code is not obfuscated and also does not implement any signature or installer verifications. Furthermore, we found the following artifacts in Bitpay's data folders:

Wallet Keys were found in a file `com.bitpay.wallet/files/profile`. This file also contained many key pairs including API keys, request keys, and AES encryption keys. In reality, only the wallet private key is needed to steal funds, but all of the other keys exemplify the lack of security implemented by this application.

Transaction History was found in the file `com.bitpay.wallet/files/txsHistory-<wallet-id>`. This file contained transaction information including the transaction id, amount, address of the sender, and time of the transaction.

4.7 Coinpayments

Besides Bitpay, Coinpayments was also developed using Cordova and suffers from the same weakness. Furthermore, Coinpayments allows the user to backup

the application data which can be used by someone with physical access to the device to access the application data by backing it up and decompressing it on a forensic workstation[18]. Note, this procedure does not require root access. The following artifacts were found in Coinpayments' data folders:

API Public/Private key pair found in a database in `net.coinpayments.coinpaymentsapp/app_webview/databases/file__0`. With access to these keys, it may be possible to send requests as the user and transfer funds from their wallet.

Passcode in plaintext is found in the same database as the public/private key pair. This passcode is used to access the application and with it, an attacker/investigator could enter the application and control the funds in the wallet.

4.8 Summary

To summarize, Bitcoin Wallet, Coinpayments, and Bitpay show a complete lack of basic security practices such as encrypting sensitive wallet or application data. It is also noteworthy that 6 of the 7 applications store transaction history on the device, even if they store other wallet data on their servers. Based on your findings, we rank the applications from most to least secure[19]:

1. *Binance* does not store any information on the physical device. Application code is obfuscated and signature verification is performed.
2. *Xapo* does not store wallet private keys, but does store transaction history. The code is heavily obfuscated and even crashes Apktool when trying to disassemble it.
3. *Mycelium* stores all the data on the device, but it is encrypted. It is currently possible to decrypt the data, however a potential solution for this exists in the code; it is just not implemented.
4. *Coinbase* does not store wallet private keys on the device but it does store the transaction history. There is also a specific scenario where a plaintext password can be obtained.
5. *Bitcoin Wallet* also to obtain all wallet information, including private keys, by using an open source tool to dump the wallet data. It is possible to make this more secure by requiring a password to dump private keys but that is not implemented.
6. *Bitpay* provides next to no security as wallet keys are stored in plaintext. Transaction history for the wallet can also be found.
7. *Coinpayments* stores wallet keys in plaintext. Additionally, even if the user locks the application with a passcode, that passcode is stored in a plaintext database and easily retrieved.

[18] https://nelenkov.blogspot.ca/2012/06/unpacking-android-backups.html (last accessed 2018-05-08).

[19] When ranking these applications, server-side security is not considered. This research was only concerned with what data, if any, is present on the physical device.

```
private void login() {
    if(!Utils.isConnectedOrConnecting(((Context)this))) {
        Utils.showMessage(((Context)this), 0x7F0801B0, 1);
    }
    else {
        this.showProgress(true);
        this.mReferrerId = PreferenceManager.
    getDefaultSharedPreferences(((Context)this)).getString("
    referral", null);
        this.getAuthTypeForLogin(this.mEmailView.getText().
    toString(), this.mPasswordView.getText().toString(), this.
    m2faToken, this.mReferrerId, new AuthCallback() {

    ...
```

Listing 1: Application code that handles the user's email and password.

5 Trojan Proof-of-Concept

To illustrate how an insecure reverse engineered wallet may be exploited, we constructed a proof-of-concept trojan for Coinbase that steals user login credentials. This type of attack does not only apply to Coinbase and in many cases the same code used in this attack may be used for other applications. This section details how the trojan was created.

Locating the Data to Steal. The purpose of this trojan is to steal the users' data and upload it to a remote server. In the case of Coinbase, the ideal data to steal would be the user's e-mail and password associated with the application. Coinbase does not store the wallet private keys on the local device but rather on the Coinbase servers. Because of this, the most useful data to an attacker is the user's login information. With this, an attacker gains full access to the user's account and thus can steal the cryptocurrency in the wallet. It should be noted that Coinbase offers two-factor authentication which may prevent an attacker from logging into the account. This additional security is opted-into, so not all users will have it enabled. Even if two-factor authentication is enabled, other user data may be exfiltrated such as the user's credit card details.

To locate where the e-mail and password is used in the code, we decompiled the application to the base smali code. We used JEB for this, however, it can be conducted with free tools mentioned earlier in this paper such as Apktool and Dex2Jar. With no code obfuscation, locating the relevant code was straight forward. The class titled 'LoginActivity' handles the login process. In this class we found a login() method which pulls the e-mail and password from the GUI and submits it to the authentication method. We chose this location to insert our code to steal the user's credentials. The advantage of this location is that the application already does the job of acquiring data from the GUI so all we have to do is copy the values and send them to our remote server.

Listing 1 shows the code in the application that steals the user's e-mail and password from the GUI. This snippet shows how tools like JEB and Dex2Jar can automatically translate the smali code into Java. While not perfect, the Java code is much easier and faster to read and understand.

Editing the Smali Code. After locating the critical section of the code, we implemented the malicious part. The attack consists of uploading the users' e-mail and password via a GET request to a remote server. To do this, we first created a thread class in smali that handles the opening of the URL. This is done because HTTP requests cannot be performed on the main thread of an Android application. The code for this thread is contained in the 'uploader.smali' file and is shown in Listing 2. Note, this snippet could theoretically be reused for any application to open a URL. The only required change is the path to the class (`/coinbase/android/signin`).

After the thread class is created and added to the application, we then needed to call it while the user attempts to login. The snippet in Listing 3 was inserted into the smali file. It is important to ensure that any of the registers used in the new code will not impact the following code as we want the application to function normally with our changes. Our proof of concept constructs a string with the URL and invokes the thread class we created previously to open the URL. The server then reads the e-mail and password from the GET request and logs it to a file. Without analyzing network traffic, the user would have no idea that anything was stolen. Since the request is done asynchronously in another thread, there is no perceptible change in performance and the application continues to function normally.

6 Conclusions and future work

In general, the most secure of the applications we tested, Xapo, Binance, and Coinbase, did not store the wallet private keys locally on the device. This does not mean that the keys are managed securely on their servers, though, and having many keys in one location makes these companies larger targets for attacks as the potential reward for a successful hack is higher. This practice is necessary in order to deliver the platform-agnostic service they offer (i.e., being able to log in from anywhere and access your wallet).

Storing keys securely on the client side (within the application) is trickier and requires secure design. Mycelium almost accomplishes this, and for all practical purposes they do. The only shortcoming is not implementing a user password which would solve the problem of technically being able to decrypt the keys as well preventing someone with access to the device from simply opening up and using the application. Applications like Bitpay, however, fall on the other end of the spectrum and store the private keys in plaintext.

From a forensics investigation point of view, it is certainly helpful to know that transaction information was available for six of the seven applications. Even if one cannot gain direct control of the wallet, having the transaction history and

```
.class public Lcom/coinbase/android/signin/uploader;
.super Ljava/lang/Object;

.implements Ljava/lang/Runnable;

.field public urlstring:Ljava/lang/String;

.method public constructor <init>(Ljava/lang/String;)V
    .locals 3
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    iput-object p1, p0, Lcom/coinbase/android/signin/uploader
   ;->urlstring:Ljava/lang/String;

    return-void
.end method

.method public run()V
    .locals 10

    iget-object v2, p0, Lcom/coinbase/android/signin/uploader
   ;->urlstring:Ljava/lang/String;
    new-instance v8, Ljava/net/URL;
    invoke-direct {v8, v2}, Ljava/net/URL;-><init>(Ljava/lang/
   String;)V
    invoke-virtual {v8}, Ljava/net/URL;->openConnection()Ljava
   /net/URLConnection;
    move-result-object v9

    invoke-virtual {v9}, Ljava/net/URLConnection;->
   getInputStream()Ljava/io/InputStream;

    return-void

.end method
```

Listing 2: uploader.smali – A thread class that opens the given URL.

public address may help in tracing how funds are being moved around. This is especially important considering how popular cryptocurrencies are becoming in the criminal world.

While the scope of this research focused on static analysis, in the future we examine dynamic analysis of the applications and see what may be found in memory. While some of the applications securely handle the private keys by not storing them on the device, it is possible that they could be found in memory at some point during the application's execution time.

Acknowledgements. Blinded for review.

```
#Create URL string (v1 = email, v2 = password)
const-string v0, "http://x.x.x.x:8000/upload?u="
invoke-virtual {v0, v1}, Ljava/lang/String;->concat(Ljava/lang
    /String;)Ljava/lang/String;
move-result-object v0
const-string v7, "&p="
invoke-virtual {v0, v7}, Ljava/lang/String;->concat(Ljava/lang
    /String;)Ljava/lang/String;
move-result-object v0
invoke-virtual {v0, v2}, Ljava/lang/String;->concat(Ljava/lang
    /String;)Ljava/lang/String;
move-result-object v0

#Open URL
new-instance v8, Lcom/coinbase/android/signin/uploader;
invoke-direct {v8, v0}, Lcom/coinbase/android/signin/uploader
    ;-><init>(Ljava/lang/String;)V
new-instance v9, Ljava/lang/Thread;
invoke-direct {v9, v8}, Ljava/lang/Thread;-><init>(Ljava/lang/
    Runnable;)V
invoke-virtual {v9}, Ljava/lang/Thread;->start()V
```

Listing 3: Injected code into LoginActivity.smali to construct the URL and call uploader.smali.

References

1. Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66–72. IEEE, 2011.
2. Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.
3. Michael Doran. A forensic look at bitcoin cryptocurrency. *SANS Reading Room*, 2015.
4. Nikolay Elennkov. Unpacking android backups. https://nelenkov.blogspot.jp/2012/06/unpacking-android-backups.html, June 2012.
5. William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.
6. Andrew Hoog. *Android forensics: investigation, analysis and mobile security for Google Android*. Elsevier, 2011.
7. Sudhir Khatwani. Top 5 biggest bitcoin hacks ever. https://coinsutra.com/biggest-bitcoin-hacks/, Nov 2017.

8. Jeff Lessard and Gary Kessler. Android forensics: Simplifying cell phone examinations. 2010.
9. Stefan Maus, Hans Höfken, and Marko Schuba. 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*, 2011.
10. Avi Mizrahi. Hackers Steal $400k from Users of a Stellar Lumen (XLM) Web Wallet. https://news.bitcoin.com/hackers-steal-400k-users-stellar-lumen-xlm-web-wallet/, Jan 2018.
11. Angelica Montanez. Investigation of cryptocurrency wallets on ios and android mobile devices for potential forensic artifacts. 2014.
12. Jason Moore, Ibrahim Baggili, and Frank Breitinger. Find me if you can: Mobile gps mapping applications forensic analysis & snavp the open source, modular, extensible parser. *Journal of Digital Forensics, Security and Law*, 12(1):7, 2017.
13. Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.
14. Becky Peterson. Thieves stole potentially millions of dollars in bitcoin in a hacking attack on a cryptocurrency company. http://www.businessinsider.com/nicehash-bitcoin-wallet-hacked-contents-stolen-in-security-breach-2017-12, Dec 2017.
15. Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.
16. Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333. IEEE, 2010.
17. Melanie Swan. *Blockchain: Blueprint for a new economy.* " O'Reilly Media, Inc.", 2015.
18. Don Tapscott and Alex Tapscott. *Blockchain Revolution: How the technology behind Bitcoin is changing money, business, and the world*. Penguin, 2016.
19. Luuc Van Der Horst, Kim-Kwang Raymond Choo, and Nhien-An Le-Khac. Process memory investigation of the bitcoin clients electrum and bitcoin core. *IEEE Access*, 5:22385–22398, 2017.
20. Daniel Walnycky, Ibrahim Baggili, Andrew Marrington, Jason Moore, and Frank Breitinger. Network and device forensic analysis of android social-messaging applications. *Digital Investigation*, 14:S77–S84, 2015.
21. Xiaolu Zhang, Ibrahim Baggili, and Frank Breitinger. Breaking into the vault: Privacy, security and forensic analysis of android vault applications. *Computers & Security*, 70:516–531, 2017.
22. Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

A Python script

```
#!/usr/bin/python

# Pulls all data from the device for a given APK
# Run by calling 'monkeyrunner getappdata.py'

import subprocess
import sys
import getopt
from com.android.monkeyrunner import MonkeyRunner,
    MonkeyDevice

def main(pkg_name=None, apk_path=None):
    device = MonkeyRunner.waitForConnection()
    if pkg_name is not None:
        res = pull_data(pkg_name.decode('utf-8').strip(),
    device)
        print(res)
        sys.exit(2)

    if apk_path is not None:
        pkg_name = get_package_name(apk_path)
        res = pull_data(pkg_name.decode('utf-8').strip(),
    device)
        print(res)
        sys.exit(2)

def get_package_name(apk):
    command = "aapt dump badging " + apk + " | grep -oP \"(?<=
    package:\ name\='[^']*\""
    process = subprocess.Popen(command, shell=True, stdout=
    subprocess.PIPE)
    output, error = !process.communicate()
    if output == '':
        sys.exit(2)

    return output
```

Listing 4: Python script to retrieve application data from device

```
def pull_data(pkg_name, device):
    result = device.shell("test -d /data/data/"+pkg_name+" &&
    echo 'true' || echo 'false'")
    result = result.strip()
    if result == 'false':
        return "The specified package name does not exist."

    result = device.shell('su -c cp -R /data/data/'+pkg_name+'
     /sdcard/data/')
    if result is None:
        return "error"
    command = "adb pull /sdcard/data/"+pkg_name
    process = subprocess.Popen(command, shell=True,  stdout=
    subprocess.PIPE)
    for line in process.stdout:
        print(line.decode().strip())
    process.stdout.close()
    output = process.wait()
    device.shell('rm -r /sdcard/data/'+pkg_name)
    return output

if __name__ == "__main__":
    helpstring = 'monkeyrunner getappdata.py <package_name> or
     <path_to_apk>'

    if len(sys.argv) < 2:
        print("ERROR: Please include the apk path or package
    name")
        sys.exit(2)

    arg = sys.argv[1]

    if arg[-4:] == '.apk':
        main(apk_path=arg)
    else:
        main(pkg_name=arg)
```

Listing 5: Python script to retrieve application data from device (cont.)