Security Analysis of Multi-Factor Authenticator

Google Authenticator Vulnerability Mitigation

JIAWEN ZHANG, Simon Fraser University, Canada
YUE ZHANG, Simon Fraser University, Canada

Multi-factor Authentication (MFA) is an authentication method that requires the user to provide two or more verification factors to gain access to a resource such as an application, online account, or a VPN. MFA is a core component of a strong Identity and Access Management (IAM) policy. In addition to enforcing login credentials such as username and password pairs, MFA requires extra verification factors, which decreases the likelihood of a security breach.

CCS Concepts: • **Security and privacy → Authentication**; **Access control**; **Information accountability and usage control**; **Management and querying of encrypted data**.

Additional Key Words and Phrases: Multi-factor Authenticator, Database Encryption, Vulnerability Mitigation

1 INTRODUCTION

High profile security breaches have become frequent this decade. Most of these breaches exploit login credentials of end users to bypass the enterprises' authentication mechanism and steal users' personal data stored in cloud databases. Leading organizations are aware of this issue and are starting to focus on seeking better authentication methods. MFA have become a popular mechanism for achieving this purpose since MFA requires an additional layer of authentication to access the system. Most MFA methodologies are based on one of the following information [4]:

(1) **Something you have**
Some physical objects in the possession of the user, such as a security token, a bank card, a key.
(2) **Something you know**
Certain knowledge only known to the user, such as a password, PIN, TAN.
(3) **Something you are**
Some physical characteristics of the user (biometrics), such as fingerprint, eye iris, voice.
(4) **Somewhere you are**
Some connections to a specific computing network or using a GPS signal to identify the location.

A good example of MFA is to supplement a user-controlled password with a one-time password (OTP) or code generated or received by an authenticator (i.e. a smartphone) that only the user possesses. This kind of authenticator application enables two-factor authentication (2FA) by showing a randomly-generated and constantly refreshing code which the user can use. These apps work even without an internet connection since the randomly generated tokens are time-based or counter-based. There are some well-known authenticator apps such as Google Authenticator, Microsoft Authenticator; some password management providers like LastPass offer the services as well.[4]

2 VULNERABILITY DISCOVERY AND AUTHENTICATORS COMPARISON

2.1 Vulnerability Description

Due to security considerations, Google Authenticator uses SQLite as the database to store unique account keys on your own devices rather than uploading them to a cloud database. A major problem

Authors' addresses: Jiawen Zhang, jiawen_zhang@sfu.ca, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A 1S6; Yue Zhang, yza582@sfu.ca, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A 1S6.

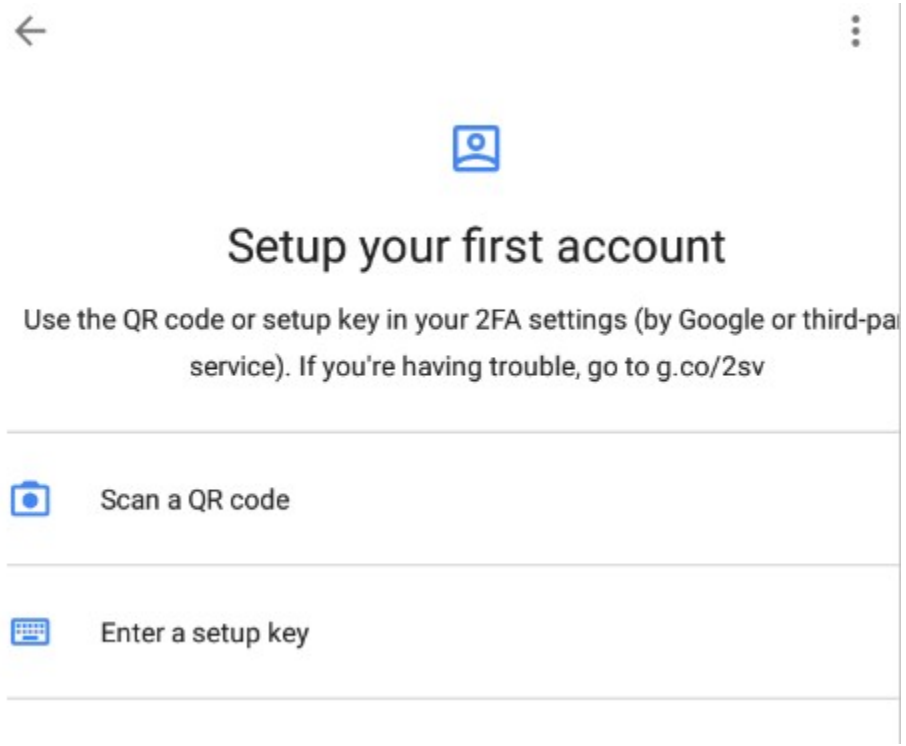

Fig. 1. Screenshot of Google Authenticator on Device 2 before transforming the databases file.

with this application is that it does not encrypt the database file. This causes the data stored in the database could be read by any user or application holding the database file.

2.2 Vulnerability Discovery

We designed an experiment to detect whether the database file of Google Authenticator is encrypted or not.

(1) Prepare an Android emulator, which provides running multiple emulator instances at the same time. We chose *LDPlayer[1]* as our emulator.
(2) Start an emulator device, let's call it Device 1.
(3) Enable root permission of Device 1. The default setting of LDPlayer has already enabled that.
(4) Download Google Authenticator and ES File Manager, other file manager applications are also available.
(5) Create a new instance Device 2 by cloning Device 1 via LDMultiPlayer, such that we have two totally same test environments for further work.
(6) Run Google Authenticator in Device 1, then add an account to that.
(7) A **databasees** file is created under Google Authenticator directory. The file path could vary on different device models and application versions. In this case, the path is */data/data/com.google.android.apps.authenticator/files/databases* under Root. ES File Manager can explore files on any Android device.

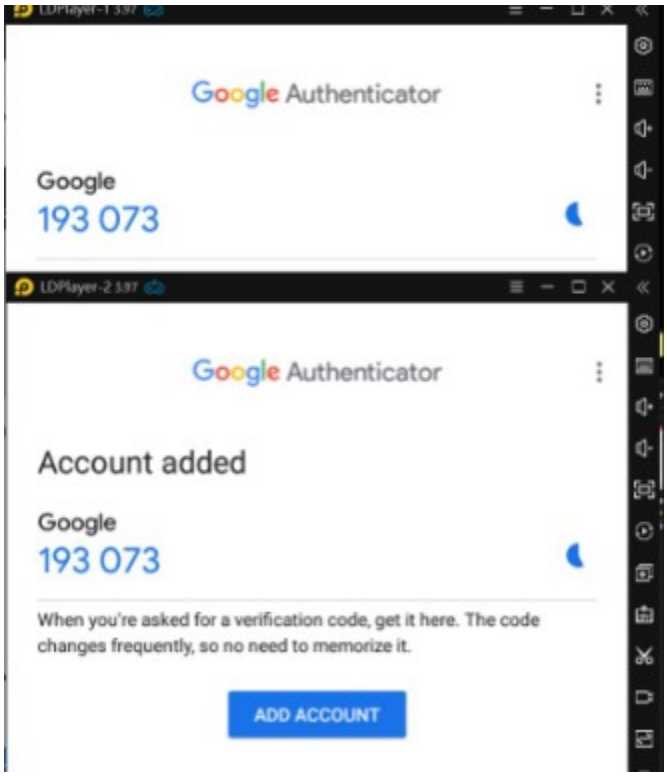

Fig. 2. Screenshot of Google Authenticator on Device 1 and Device 2 at the same time after transforming the databases file.

(8) There are several methods to transfer files between LDPlayer emulator devices. We use ES File Manager to copy and paste the **databases** file to */mnt/shared /* folder under Root. This directory is exposed to the Windows system.
(9) (Optional) Direct to the LDPlayer documents folder with Windows Explorer. The path should be */My Documents/LDPlayer*. Remember that this path is not the LDPlayer install location. The **databases** file should be there.
(10) (Optional) Open the **databases** file with DB Browser (SQLite). If it is open successfully without a password, then we can say this file is not encrypted.
(11) Start Device 2.
(12) Use ES File Manager to copy and paste the file from */mnt/shared/databases* to */data/data/com.google.android.apps.authenticator/files/*.
(13) Open Google Authenticator in Device 2.
(14) If the accounts added in Device 1 is displayed in Device 2, then the **databases** file is uncrypted and can be loaded by any other devices.

In our experiment, the account added in Device 1 could be shown in Device 2 after transforming the **databases** file. Thus, we found that Google Authenticator uses an unencrypted database to store sensitive account information, which could be very dangerous for an application designed for security usage. Figure 1 shows that the Google Authenticator on Device 2 did not have an account

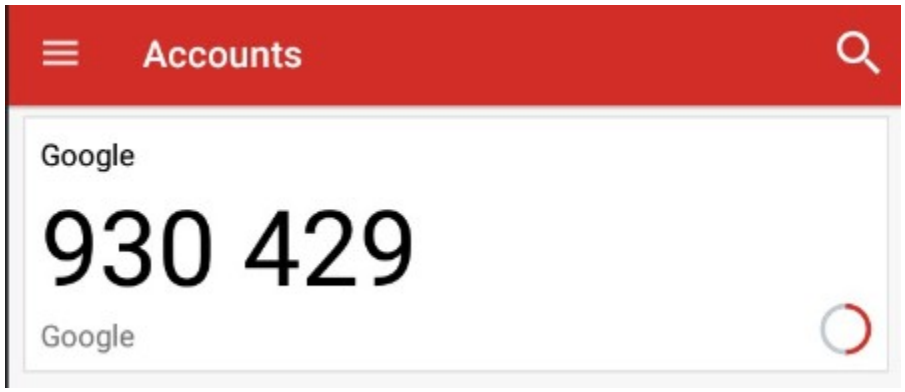

Fig. 3. Screenshot of Lastpass Authenticator on Device 1

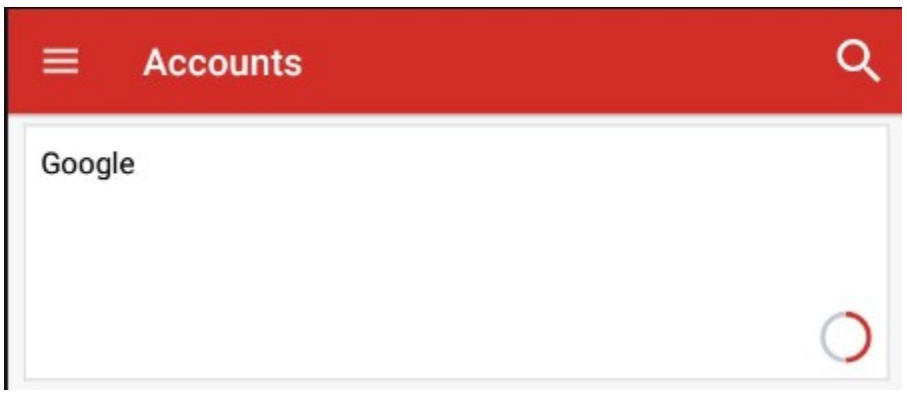

Fig. 4. Screenshot of Lastpass Authenticator on Device 2 after transforming the databases file from Device 1

before transforming. Figure 2 shows that Google Authenticators on both Device 1 and Device 2 had the same data after transforming.

2.3 Authenticators Comparison

After finding the unencrypted database vulnerability, a new question arose: do other 2FA applications have the same problem? Google Authenticator and Microsoft Authenticator are the Top two 2FA applications in the market. For some unknown reasons, Microsoft Authenticator is currently unavailable to download in Google Play. We decided to use another alternative application, Lastpass Authenticator, to compare with Google Authenticator. Applying our experiment, the result showed that the data of Lastpass Authenticator in Device 1 could not be totally read in Device 2, and the readable parts are not sensitive. Therefore, Lastpass Authenticator is more secure than Google Authenticator. Figure 3 and Figure 4 show the experiment results of Lastpass Authenticator.

3 VULNERABILITY EXPLOITATION IN OPEN WORLD

3.1 Attacker Model

Attacker Model. Assume the scenario consists of two victims Alice and Bob. Alice is a participant in a public wireless network; Bob is a public VPS. Alice is using an Android device. Suppose the

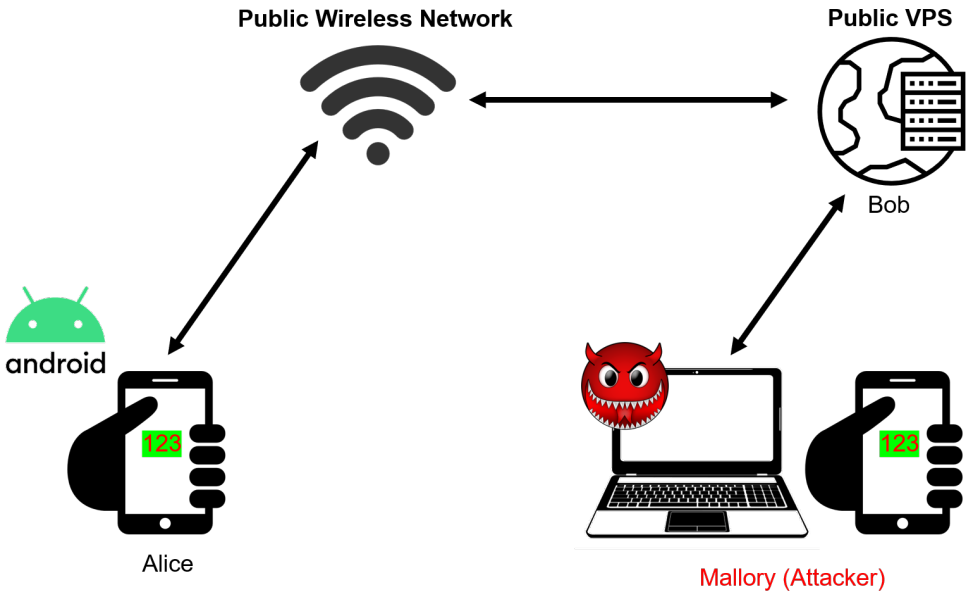

Fig. 5. An attack model that describes the attack scenario of database file stealing and token recovering.

attacker Mallory, who wants to steal the database file which includes the unique account keys that stored in Alice's device by injecting malware in Alice's device or spawn a reverse shell in Android that tunnel communication over HTTP/HTTPS. Assume Mallory is not in range with the public wireless network; he can access Bob's device, but no direct access to Alice's device. Mallory can use Bob as an intermediate to attack Alice's device and steal the data. After Mallory acquires the file, he can input the data into another device that running the same authenticator, the second device will hold the same 2FA token as the first device. Figure 5 represents how the attack works. In our experiment, we transformed the database file from Device 1 to Device 2. After that, Device 2 held the same data as Device 1.

4 MITIGATION IMPLEMENTATION

Google Authenticator uses SQLite as the database to store the unique account key without encryption; we have demonstrated this is not a secure implementation. We propose two mitigation methods; one way is to encrypt the data, the other one is to encrypt the entire database.

4.1 Comparison of Encrypting Data Entries and Encrypting Database

Both encrypting each entry of data and encrypting the entire database can help us to encrypt the SQLite database used by Google Authenticator. To find the appropriate solution, we compare these two methods in three aspects

4.1.1 ***Security.*** If we encrypt the entire database, a hacker only needs to focus on attacking the encrypted database file. Once the hacker succeeds, he/she can get full access to the database. By comparison, the hacker has to crack every single entry of the data to read all data, if each entry of the data is encrypted separately. Furthermore, if different secure algorithms are applied to different entries, we believe it is almost impossible to crack the data.

4.1.2 ***Performance.*** Obviously, encrypting only the database file runs much faster than encrypting each entry of data. In an application life cycle, at most one encryption and decryption is required to process the database file. Otherwise, we have to do the encryption or decryption each time reading or writing data if the separate entry of data is encrypted.

4.1.3 ***Difficulty of Implementation.*** The original source code uses an unencrypted SQLite package. Thus, the easiest implementation could be replacing this unencrypted SQLite package with an encrypted SQLite package. If we want to encrypt each entry of the data separately, we have to change the code structure, which means a great number of changes. In other works, encrypting the database file is much easier than encrypting data in this case.

4.2 Encrypting Data Entries - Yue Zhang

To encrypt each entry of data is a general solution of database encryption. In this case, none 3^{rd} party modules are required. Therefore, we did not meet any annoying dependence broken problem. However, this solution has a large number of conflicts with the original source code and code structure. We have to overwrite plenty of code to implement this method. Unfortunately, because I did not solve all of these read/write code conflicts, the .zip file does not include the source code and APK of this implementation.

4.2.1 ***Implementation.*** There are several waysto encrypt the Strings, but the best we found is Android Keystore API for this. The Android Keystore system lets users store cryptographic keys in a system container to make it more difficult to extract from the device. Once keys are in the Keystore, they can be used for cryptographic operations with the key material remaining non-exportable. Another benefit of using Android Keystore is that we do not have to hardcode the key for encryption and decryption.

The steps of this implementation include:

(1) Create and load a Keystore instance.
(2) Generate a key and a IV for AES-256-GCM algorithm.
(3) Set an alias for the key. The alias can be used to find and load the key from the Keystore.
(4) Encrypt each entry of data before inserting or updating them into the database.
(5) Store the encrypted data into the database.
(6) For decryption, load the encrypted data from the database, then decrypt them.
(7) **Having a unique IV for each encryption is important.**

Because the keys are stored in Keystore container of the device, and the key file is unavailable to export from Keystore, I believe that only the user who knows the aliases can load the keys to do the decryption.

4.3 Encrypting Entire Database - Jiawen Zhang

I did not implement my own encryption since a non-peer-reviewed algorithm could contain flaws. Instead, I used an extension that integrates widely available encryption libraries like OpenSSL libcrypto, LibTomCrypt and CommonCrpto for all cryptographic functions.

4.3.1 ***Implementation.*** To encrypt the entire database, we need the SQLite Encryption Extension (SEE), which is an add-on to the public domain version of SQLite that allows an application to read and write encrypted database files. SEE supports four different encryption algorithms: RC4, AES-128-OFB, AES-128-CCM, AES-256-OFB.[3] However, SEE is only available in the SQLite commercial edition; as a result, I turn to use an open-source extension for SQLite - SQLCipher. SQLCipher has a small footprint and outstanding performance, so it's ideal for protecting embedded application

databases and is well suited for mobile development. It encrypts the database with 5-15% overhead. I verify the security of SQLCipher; here is a list of SQLCipher security features [2]:

(1) The encryption algorithm is AES-256-CBC
(2) Each database page is encrypted and decrypted individually. The default page size is 4096 bytes.
(3) Each page has it's own random initialization vector (IV). The IV is generated by a cryptographically secure pseudorandom generator (e.g. OpenSSL's *RAND_bytes*, CommonCrypto's SecRandom, LibTomCrypt Fortuna). IV are regenerated on write so that the same IV is not reused on subsequent writes of the same page data.
(4) Every page write includes a Message Authentication Code (HMAC-SHA512) of the ciphertext and the IV at the end of the page. The MAC is checked when the page is read back from disk. SQLCipher will throw an exception if the ciphertext or IV have been tampered.
(5) Each database is initialized with a unique random salt in the first 16 bytes of the file. This salt is used for key derivation with PBKDF2-HMAC-SHA512.

I implement the mitigation by substituting the simple public domain version of SQLite with SQLCipher. And I use the Android hardware identifier of the device as the passphrase to encrypt the database since this identifier is different for each device.

4.3.2 ***Difficulties Encountered.*** I encountered a bunch of problems when integrating SQLCiper with the Google Authenticator app.

First, Google Authenticator uses Bazel as its build tool. Bazel is an open-source version of the Google internal build tool - Blaze. I had had no experience with Bazel before, and there are limited resources online for building Android apps with Bazel. I spent a lot of time investigating the file hierarchy under Bazel. Bazel is hard to use with Android Studio. Although, Android Studio has a Bazel plugin, it is not possible to build a production app directly from Android Studio using Bazel, which means there is no debug mode available. I had to monitor the Logcat of the Android virtual device (AVD) to trace the call stacks and track the errors and exceptions, which is difficult and time-consuming.

Second, due to COVID-19, equipment lending at the SFU libraries is suspended, so I can not borrow an Android device (arm64 architecture) to test my modified app. The only way to test the app is to build the app under x86 architecture and test the app on AVD. This resulted in some difficulties for me. I have to resolve the dependency problem since SQLCipher uses native libraries. SQLCipher first builds the utility functions that are implemented in native code such as C and C++ with Android Native Development Kit (NDK), then links the shared object (.so file) while generating production apps. I spent a lot of time resolving such issues.

Third, Google Authenticator enables ProGuard. ProGuard is an open source command-line tool that shrinks, optimizes and obfuscates Java code. It optimizes bytecode as well as detects and removes unused instructions.[5] Usually, these features are excellent since it makes your apk package smaller and makes the building process efficient. However, it misdetects the dependency library for SQLCipher as an unused instruction; thus, the app always crashes even you successfully build the app since it throws a *java.lang.NoSuchFieldError* exception that indicates the filed *mNativeHandle* does not exist. It takes me a good while to catch the issue and find out the reason.

Forth, Keystore can be used for store keys, but not IVs. We still have to hardcode IVs insecurely on devices. This does not affect the security of encrypting with Keystore, but we think the code implementation is inelegant.

There are some other minor problems such as the SQLCipher does not use the default file path to store the data, and so on. Although there were myriad hardships and hazards, I made it.

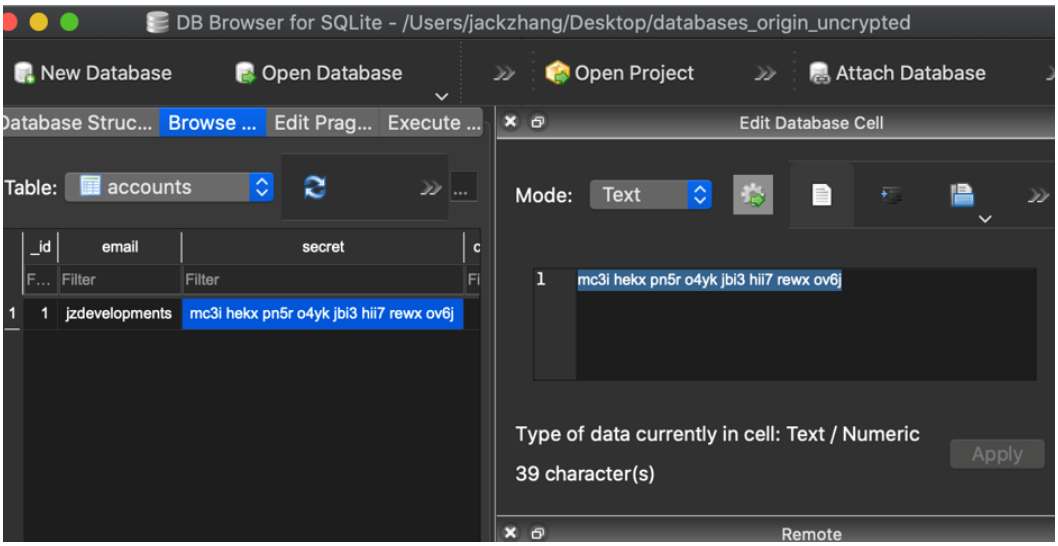

Fig. 6. The unencrypted database file can be opened by SQLite Browser directly.

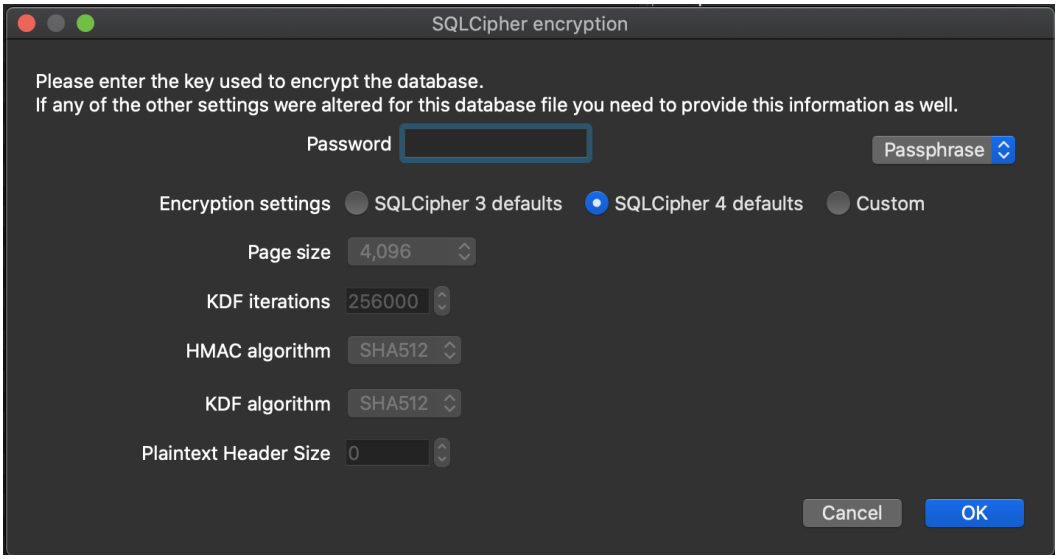

Fig. 7. The encrypted database file can not be opened by SQLite Browser, unless the adversary gains the passphrase, which is barely possible.

4.3.3 ***Results.*** In this section, I demonstrate the result of my mitigated version of Google Authenticator.

If you open the database file that generated by the origin Google authenticator, the unique account key will pop up instantly. Figure 6 shows the unencrypted database file in this situation.

If you try to open the database file that generated by my mitigated authenticator, you will see a dialog that prompts you to enter the passphrase that is used to encrypt the database. Thus, if the

adversary copy the database file into another device, it will not work since the Android hardware identifier is different from each device. Figure 7 is the dialog when opening an encrypted database file.

4.3.4 ***Installation Instruction.*** To install my mitigated app, just drug the *authenticator.apk* into an x86 AVD. And you can explore and test it, enjoy. The *.zip* file contains the completed implementation, you can check that as well.

5 CONCLUSION

MFA plays an important role in the modern internet. However, one of the most 2FA mobile applications, Google Authenticator, is insecure. Google Authenticator uses an unencrypted SQLite database to store sensitive information on Android devices. To fix this vulnerability, we propose two methods, encrypting entire database file and encrypting data entries separately. We compared and implemented both methods and recorded the difficulties we encountered.

REFERENCES

[1] LDPlayer. [n.d.]. LDPlayer Official Website. https://www.ldplayer.net/games/
[2] Zetetic LLC. 2020. SQLCipher Design. https://www.zetetic.net/sqlcipher/design/
[3] SQLite. 2020. SQLite Encryption Extension. https://www.sqlite.org/see/doc/trunk/www/readme.wiki
[4] Wikipedia contributors. 2020. Multi-factor authentication — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Multi-factor_authentication&oldid=993885794. [Online; accessed 14-December-2020].
[5] Wikipedia contributors. 2020. ProGuard (software) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=ProGuard_(software)&oldid=979196522. [Online; accessed 14-December-2020].