Security Analysis of Mobile Authenticator Applications

Can Ozkan
Department of Computer Engineering
TOBB University of Economics and Technology
Ankara, Turkey
canozkan1994@gmail.com

Kemal Bicakci
Securify Information Tech. and Security Training Consulting Ltd.
Ankara, Turkey
bicakci@etu.edu.tr

***Abstract*—Deploying Two-Factor Authentication (2FA) is one of the highly-recommended security mechanism against account hijacking attacks. One of the common methods for 2FA is to bring something you know and something you have factors together. For the latter we have options including USB sticks, smart cards, SMS verification, and one-time password values generated by mobile applications (soft OTP). Due to the cost and convenience reasons, deploying 2FA via soft OTPs is more common. However, unlike smart cards which have tamper resistance property, attackers can access smartphones remotely or physically so that they can fetch shared secret seed value - an important security risk for mobile authenticators. For this reason, it is critical to analyze mobile authenticator applications in this context. In this paper, we report our findings after analyzing eleven different Android authenticator applications. We report that we have fetched cleartext shared secret seed value from storage in five applications and from memory in seven applications using standard reverse engineering techniques and open-source tools.**

***Index Terms*—Android, Mobile Security, Reverse Engineering, Obfuscation, Cryptographic Controls, ProGuard, Android Key Store, Mobile Authenticator, Authentication, Two Factor Authentication**

I. Introduction

In the 21st century, the technology we use has changed dramatically. Online services have gained tremendous popularity [1]. We are now using online systems much more than it was before via portable devices such as tablets and mobile phones for new and evolving applications such as social media, financial operations, online education, etc. We pay our bills, purchase products, even work remotely via online systems [2]. With so much of life happening online, user accounts have become a natural target for criminals. Malicious attacks against user accounts are common today. As a result of account takeovers, users and companies might suffer from financial and reputation loss.

In order to improve account security, one of the technologies deployed is two-factor authentication, commonly referred to as 2FA, a technology of authenticating users through two different factors. When 2FA is applied, knowing the target's credential is not enough for attackers to compromise the account due to the fact that there are now two different and independent means of authentication. First, user enters his/her credential. This is something you know factor and could be personal identifier number (PIN), password, etc. Then, there will be a second layer which might be something you have or something you are. The more common case is something you have, which asks user to prove that something (USB stick, smart card, hardware token or mobile phone) is in possession of the user. Mobile phone based 2FA might be preferable due to convenience or cost reasons. It could be implemented with SMS verification, which works as follows: for the second factor, the system sends the one-time password (OTP) value to the user over the mobile phone network via the Short Message Service. Another convenient (and more secure) way for 2FA is soft-generated OTP, which can be generated by an application installed to mobile devices and can work offline [3]. Almost everyone has a smartphone today and implementing 2FA in this way is free of charge (unlike SMS or hardware based solutions). For this reason, it is important to analyze the software based authenticator applications regarding the security level they provide.

One security issue with software application based OTP generators is that the application generally comprises source code compiled into an APK (for Android) or IPA (for iOS). In order to produce an OTP value (for instance using TOTP algorithm), secret seed value is used. If secret seed value is revealed to the attacker, she could produce all of the OTP values to be used throughout the entire lifetime of the application. By reverse engineering, the attacker can analyze applications installed on user's smartphones and capture the shared secret seed value in cleartext.

In this paper, we analyze the security of mobile authenticators on Android platform. Android has security mechanisms, one of which is referred as mobile application containers isolating the mobile application from the mobile operating system or any other applications installed on the same device. In addition, vendors apply their own software protection mechanisms, which include but not limited to obfuscation, anti-tampering, root check, cryptography and device binding. The combination of mobile application containerization and vendor-supplied protections make the OTP generator applications more secure at least in theory. Our goal in this paper is to analyze and understand the situation in practice and report

978-1-6654-1863-8/20/$31.00 ©2020 IEEE

our findings for the chosen mobile authentication applications.

Our approach is to implement Android Reverse Engineering techniques (mainly applying both static and dynamic methods by executing apktool, Jadx-GUI, Frida, Objection, adb) using non-commercial open-source tools. Upon installing authenticator applications to mobile environment, we investigate whether the shared secret value could be reached in cleartext from storage and/or memory.

In this paper, we report that the secret seed value could be obtained in cleartext with our methodology from both storage as well as from memory in a surprisingly high number of authenticator applications. We discuss the security implications and possible countermeasures as well.

The rest of the paper is organized as follows. Section 2 overviews the related work. Section 3 provides the problem definition, threat model and our assumptions. Section 4 demonstrates our attack simulations. Section 5 shows our findings. Section 6 presents a general discussion and Section 7 gives implications of our results. Finally, Section 8 finishes up our paper by concluding remarks and possible future work directions.

II. Related Work

Although there are a number of researches about TOTP, not much of them are directly related to fetching shared secret value [4] [5] [6] [7]. Philip Polleit and Michael Spreitzenbarth worked on fetching shared secret seed value from mobile authenticators [8]. They analyzed mobile authenticators with respect to cloning possible, device integrity check, encrypted seed on storage, PIN protection, secure SSL protection. Also, Bernhard Mueller has published an article about hacking soft tokens on Android, showing methods how to investigate tokens. Bernard's article focused on analysing tools and strategies used for both static and dynamic analysis with the aim of fetching shared secret seed value [9].

There are researches about Android Keystore as a secure key storage solution [10] [11] [12] [13]. Also, there is a research that proposes a system called SecurePay, which is about securing two factor authentication scheme under fully compromised environment, providing authenticity, integrity of any 2FA transaction [14].

A series of academic efforts involve the development of TOTP for security solutions. Iman, Mardi and Kiki's study discusses One-Time Password installed on a mobile device in which the password is randomized using a combination of SHA256 and Time-based One Time Password algorithms [15]. sdf Jianxun's paper describes TOTP generation algorithm making the counter replace with timer, and builds an improved authentication method consistent with Three-Protocol of HOTP authentication method based on TOTP [16].

Azhari, Lucgu and Carolus' research discusses securing Android mobile banking applications from reverse engineering and network sniffing. They showed that once applications implement code obfuscation, static string encryption method, Native C and C++ implementation and add dead code, it will be harder for reverse engineers to analyse the application [17].

Contributions

We make the following contributions; 1) In addition to analyzing shared secret seed value on storage, we analyze shared secret value on memory as well. 2) We plan our threat modelling as realistic as possible in order to answer how far attackers can go. 3) We analyze 11 2FA application with respect to ProGuard usage, advanced obfuscator usage, anti-tampering resistance, KeyStore usage as well as device binding implementation.

III. Background

In this section, we discuss the necessary background information for our threat model.

A. Android OS and Its Built-in Security Measures

Android is an open source, Linux- based operating system. Being an open source operating system is the first choice of both developers and consumers. Android runs on top of Linux kernel. In Android, Linux kernel is modified in such a way that Linux kernel works better in smartphones and tablets. Although application developers develop their apps in Java or Kotlin, Android OS services are mostly written in C/C++. Android Runtime is an environment on which Android applications run. Android applications are Dalvik Executable. Classes.dex file contains all the byte code in order to be run. Android Run Time (ART) is the default run time environment as of Android 5.0. In earlier versions, DEX byte code is executed on Dalvik Virtual Machine environment (DVM). Before ART, Dalvik used to convert byte code at runtime on the fly when the app is run. This process is known as Just-in-Time (JIT) approach. With the advent of ART, application's byte code does not need to be converted into machine code every time it starts. Because this process is done during installation process. This process is also known as ahead-of-time compilation [18].

There are a couple of built-in defense mechanisms available on Android. They can be listed as follows:

Linux user-based access model: The existing storage layout mechanism on Android is different from the usual Linux kernel storage layout mechanism. Each application on Android has its own directory under /data/data. Due to the fact that each application runs as different user on Android OS, access control for each application is ensured based on per application, that is, per user. User of an application is the owner of the process ID, meaning each application is assigned a separate process ID (PID) by Linux based on its UID. Android OS ensures proper basic access control on files for non-root users. In other words, each application can only access its application specific data directory within the restriction of Linux-based privilege control [19].

Android Sandboxing: Each application runs on its own Sandbox environment on Android. By default, applications can't interact with each other and have limited access to the OS. One application cannot access to another application's resources due to Android Sandboxing that is isolating each application in its virtual machine. Android makes uses of

UID to implement kernel level sandboxing [20]. In normal Linux kernel, every process runs with the current user's UID. Android takes it a step further so that each application is assigned a UID to isolate application resources.

Secure IPC: IPC, standing for Inter-Process communication, describes the mechanisms used by Android application's component or other applications available on the same device to communicate with each other securely.

Application Signing: All Android applications need to be digitally signed with a certificate whose private key is held by the application's developer before installing on Android device. Otherwise, Android application will not be able to run on the device. In other words, any application that is not signed will not be able to be installed to Android device. Therefore, application must be signed before installation process. The Android system uses the certificate as a way of identifying the author of an application and establishing trust relationships between applications [21].

Permissions: Application defines but user grants those permissions. Permissions are declared by the developer of the application. AndroidManifest.xml file will have all the details related to permissions. Installed application asks for permissions in order to use related APIs. If permission is rejected by the user, application will not be able to make API call. While Android version 5.1 and before ask for permissions during the installation process, Android version 6 and successors ask for permission once the permission is needed.

Google Bouncers: In order to prevent malicious Android application to be present on Google PlayStore, Google introduces a service called Google Bouncers that automatically scans applications and developer accounts with the aim of probing malicious activities.

B. Android Application Overview and Architectures

An Android application is an application running on Android OS. Android applications can be written mainly using Java and Kotlin language. However, Android also supports C/C++ native implementation. These are called native application development. On the other hand, there are cross platform ways where developers write code and then build the mobile application for both Android and iOS platforms simultaneously. This type of mobile application development is called cross platform mobile application development and developers can use React Native, Xamarin, Flutter, Ionic and more to build cross platform applications.

Android application development process is as follows: First of all, developers write source code, then source code is compiled through Java compiler (javac) so as to produce Java byte code. These class files cannot be run on Android device because of the fact that Android has its own byte code format, that is, Dalvik. Next, produced Java byte code is passed to Dex Compiler in order to generate Dalvik byte code. Classes.dex is produced as a result of this step. Java byte code is translated into Dalvik executable because Android applications are basically Dalvik executable files. Once dalvik executable files are obtained, AndroidManifest.xml, resources, libraries, dalvik executables are zipped into an apk package. Upon producing apk package, apk needs to be signed so that apk will be ready to be installed. Android applications can be self-signed, meaning developers themselves can sign an application. Developers do not have to use Certificate Authority (CA) to register their public keys. AndroidManifest.xml and other constituents will be discussed deeply at Android Reverse Engineering section. Briefly, every application must contain AndroidManifest.xml and it is summary of an application indicating permissions that application needs, components application include, minimum SDK version, target SDK version and more. It is important to understand this figure because we will follow the reverse steps while reverse engineering. Steps can be summarized in Figure 1:

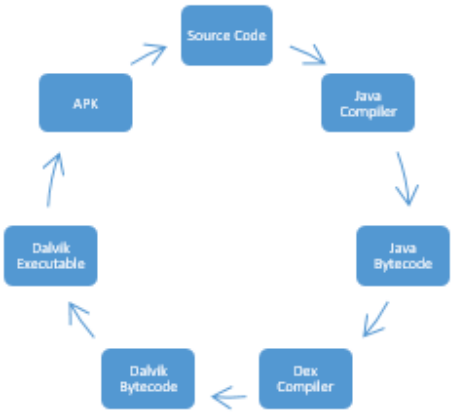

Fig 1: Android Application Development Cycle

C. Android Application Components

Android applications are made up of application components that are Activity, Intent, Service, Broadcast Receiver, Content Provider. An application may contain either some of them or all of them.

Activity: Activity provides a screen that user interacts with. So, user can perform actions. Usually, one activity corresponds to one screen. Most apps contain multiple screens, meaning they make up of multiple activities. Moreover, each activity may start another activity. All of screens in an application are called activity.

Intent: Intent is an object used to request an action from another Android application component. These actions are to start an activity, service and deliver a broadcast.

Content Provider: Content provider is used to present data to external applications as tables. When one application wants to share its data with other applications available on phone, content provider is a way to share its data with other applications and it acts as an interface to perform standard database functions such as insert, query, update, delete if another application has proper permissions.

Broadcast Receiver: It is a component that responds to a system-wide broadcast messages. Broadcast messages may include battery low, headset plugin.

D. Application Security Measures

In section 3.A, we look at default security mechanisms provided by Android OS. Upon overcoming default security mechanisms of Android applications, there is no other security mechanism in front of the cyber criminal unless Android application itself adds additional security mechanism. Here are some of additional security mechanisms further protect the application from cyber criminals.

ProGuard Obfuscation: Although ProGuard is used for shrinking and optimizing the application, it is also used for obfuscation. Consequently, it is harder for the reverse engineers to read dex byte code [22]. Its main job regarding obfuscation is to rename packages names, class names, field names, method names, parameter names, namely, it renames everything related to source code.

Advanced Obfuscation: With the help of advanced obfuscators, code base can be protected by intentionally making use of syntactic and semantic errors into the decompiled program while the original behaviours of the byte code is still preserved.

Android KeyStore: Android KeyStore, letting developers store cryptographic keys in a container, makes cryptographic key extraction process more difficult [23]. The objective is to remain cryptographic keys non-exportable while the keys can bu used for cryptographic operations once they are in key store.

Anti-Repackaging / Anti Patching: Patching is a set of changes to a software application with the aim of updating, fixing, improving software. The benign reason to patch a software is to fix vulnerabilities and bugs, add additional features, improve usability, performance and much more. Patching makes modification of the compiled software application conceivable once the source code is unavailable, requiring an exceptional understanding of the software program. It generally includes dealing with low level codes such as assembly and smali in addition to high level pseudo code. Conversely, the malicious intent is disabling anti-reverse engineering precautions, injecting malicious code to software program, crack application not to pay for subscription fee, software piracy and more. Anti-patching makes software program harder for a reverse engineer to modify. Anti-patching may take form of integrity checks, anti-debugging, resulting in malfunctioning the program or not functioning at all.

Device Binding: In some situations, we want values to be more random where the values are cryptographic keys, OTPs etc. Once we use values that are unique to user's environment such as MAC address, IMEI number then that gives extra randomness regarding values produced.

Encryption Sensitive Information on Storage: Shared secret seed value is one of the most important value an authenticator application has. It is important that it should not be kept in cleartext format. Encrypting shared secret seed value is additional security mechanism to increase security posture of authenticator application. It can be mapped to Insecure Data Storage on OWASP Top 10 mobile.

E. 2FA Techniques and TOTP Protocol

Two factor authentication is an authentication process requiring two different authentication factors to authenticate an account. A factor can be described as a way to authenticate an account into computer system, online service that an account is who they say they are. Most common authentication factor used today is username/password pair. As passwords are vulnerable to brute-force attacks, dictionary attacks, rainbow table attacks, more systems and individuals started using two factor authentication (2FA) in order to secure their digital accounts. People usually consume many systems to login, leading the user to remember too many passwords. This causes another vulnerability what we referred to as password re-use. For this reason, many vendors and service providers suggest using 2FA for their customers in an attempt to avoid data breaches and password loss.

Authentication factors can be mainly categorized as follows:

- **Something you know:** This could be password, personal identifier number (PIN), answers to secret questions, keystroke pattern.
- **Something you have:** User has something in her/his possession such as credit card, hardware token, mobile device that can be sent codes, OTP generator application.
- **Something you are:** This type of authentication refers to unique physical attributes that are inherent to human-being like fingerprint, retina scan, voice recognition.
- **Somewhere you are:** This type of authentication relies on where the user is to prove their identity.

A common example of 2FA requires something you know and something you have. Usually something you know and something you have can be achieved by username/password and SMS verification, respectively. Although SMS verification provides more security than single-factor authentication, SMS-based 2FA is not very secure on account of the fact that SMS messages can be intercepted by the attackers [24] [25]. There are other ways to perform 2FA by using mobile device, one of which is producing the verification code by means of mobile application. Google, Microsoft and other big vendors use one time passwords (OTP). A mobile application installed on user's mobile phone creates a temporary, nearly valid 30 seconds, single use code based on the time of the day with the help of TOTP protocol. This strict timeline makes it hard for attacker to circumvent the OTP value, namely, second factor.

F. Lifecycle Management of TOTP

Published as RFC 6238 by the Internet Engineering Task Force (IETF), time-based one time password (TOTP) algorithm generates single use only passwords, also known as tokens, that are only valid for a certain time period. Generated tokens are based on shared secret seed value.

Moreover, TOTP algorithm is an extension of HOTP algorithm, standing for HMAC based one time password (HOTP), published as RFC 4226. HOTP algorithm is used to produce a one time password based on a shared secret seed value and a counter.

Suppose that user has installed Google Authenticator application on a mobile device. The user has two different choice in order to supply shared secret seed value. First, service provider (Facebook, Google, Bank account etc) generates a shared secret seed value with additional information. Shared secret seed value and additional information (service provider name, algorithm used etc) are embedded in a QR code. Then user scans the produced QR code by a mobile app such as Google authenticator. Secondly, user manually enters shared secret seed value. Upon agreeing on shared secret seed value, mobile authenticator application such as Google Authenticator starts generating OTP value based on the shared secret seed value and current time.

How TOTP is Calculated?

We discussed earlier that OTP value is generated based on shared secret seed value and current timestamp. Current timestamp is converted into Unix Epoch Time, which is the number of seconds that have elapsed since 1 January 1970 00:00:00, not counting leap seconds. Due to the fact that every time current time changes, OTP value will change every second. In order to prevent that time step is calculated according to the following equation:

N = floor (T(unix) / Ts), where

N = number of time steps elapsed since Unix Epoch Time

floor = function rounding a number downward to its nearest integer

T(unix) = number of seconds elapsed since 1 Jauary 1970 00:00:00, not counting leap seconds

Ts = Time step. By default, it is 30 seconds.

Let's assume current timestamp is 1600519285

T(unix) = 1600519285

N = floor (1600519285 / 30) = 53350642

Convert N into hex format. The hex value must have 16 hex characters (8 bytes). If it is not 16 hex characters, prepend with 0's. N(decimal) = 1600519285

N(hex) = 0x0000 0000 5F65 FC75

The steps to calculate OTP value:

1) Calculate N(hex).
2) Convert the hexadecimal value into a 8 bytes array and assign this value to variable m (message)
3) Convert the shared secret key (cozkan23456cozkan) into a 20 bytes array and assign this value to variable K. The shared secret key is a randomly generated 20 bytes number which is base-32 encoded. For readability, this key is divided in groups of 4 characters and all in lower case.
4) Calculate HMAC hash using HMAC-SHA1 algorithm as shown in Figure 2.
   This HMAC hash size is 160 bits (20bytes) Lets assume the HMAC value is
   9a ee 6a 13 70 32 a0 d9 b5 e5 **37 8f 89 28** 76 2f 68 3c 0f 0a

Fig 2: HMAC calculation within TOTP algorithm

5) Get last 4 bits of hash value and convert it into integer value. In this case, it is 0xA, representing integer value of 10. This integer value is the offset.
6) Get the 4 bytes from the HMAC hash based on the offset value.
7) Apply ANDing operation with 0x7FFFFFFF
   378F8928 AND 7FFFFFFF = 5F65 FC75 in decimal 1600519285
8) Calculate the Token = 1600519285 % 10^n where n is the token size. Token = 519,285

This operation is executed on both mobile application and server side. If the values are matched, authentication is allowed. Otherwise, authentication is denied.

G. Android Keystore

Although full-disk encryption and file-based encryption provide protection data on Android storage, it may not be the best option when it comes to storing sensitive data like shared secret seed value, passwords, session IDs, tokens. The problem in this scenario is that if the device is fully compromised and therefore attacker gains root access then attacker may circumvent this scenario once the user has entered their password. As a result, attacker can see unencrypted data including sensitive ones. One of the solution arises is to encrypt sensitive data before storing it under application specific data directory. In this case, another problem arises, that is, data will need to be decrypted at some point. Therefore, there must be a decryption key in order to decrypt the encrypted data. The problem is how decryption key be protected and located? This is where Android Keystore comes into play.

If we read the official documentation of Android Keystore, it says "The Android Keystore system lets you store cryptographic keys in a 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. Moreover, it offers facilities to restrict when and how keys can be used, such as requiring user authentication for key use or restricting keys to be used only in certain cryptographic modes." We can infer from the official documentation that it can store cryptographic keys (public, private keys) and perform cryptographic operations using stored keys (encrypt, decrypt, sign, verify etc)

Keys are generated and stored in Android Keystore. Public key can be used to encrypt sensitive application data like shared secret seed value, password, token before they are stored in the application specific data directory. Likewise, private key is used to decrypt data when needed. While cryptographic operations are being executed on data, keys never

leave Android keystore. In addition to that, an application can only fetch its own keys.

Key extraction can be prevented in two different ways. Firstly, key material never enters the application process. In other words, key material never leaves Android Keystore. When cryptographic operations are done on values, values are provided into a system application that handles all cryptographic operations and returns result to application. Consequently, key stored in KeyStore never leaves the KeyStore. The other is that key material may be bound to the secure hardware such as Trusted Execution Environment (TEE), Secure Element (SE).

H. Android Reverse Engineering

Reverse Engineering is the process of analyzing a system to identify its components, and create a blueprint of the system to understand functions, hidden components. The purpose here is to understand how it was built [26]. Android application development process has been discussed at Section 3.2. Recall that developers write Java code and Java code is eventually compiled into DEX byte code. Reverse engineers work opposite direction of steps as depicted in Figure 3.

Android application with extension of apk is nothing but an archive file containing all apk contents. When we unzip the archive, we will get a list of files and folders as follows: Classes.dex, AndroidManifest.xml, META-INF, res, assets and lib. However unzipping an APK and decompiling an APK does not refer to same thing. The reason why they are not the same is that when you just unzip the apk file you will see the encoded AndroidManifest.xml and compiled dex files. To be able to overcome this issue we need to use apktool so as to decompile apk.

Apk contents are as follows:

AndroidManifest.xml: Every application has AndroidManifest.xml. It is summary of the application and specifies main components of application as well as holding most of configuration details about Android application. minSdkVersion and targetSdkVersion are specified in this section.

META-INF: Certificates are in this directory. It contains information about developer who has written the application such as name, company name, validity period. If certificate gets expired, app loses its presence on Google PlayStore. Also, it checks integrity of the files that are present in this package.

res: This folder contains raw resources necessary for application such as images, application icons, bitmaps, layout definitions, strings.

Classes.dex: These are Dalvik byte code with DEX file format. Source code written by the developers is eventually compiled into DEX file format. This DEX file is actually executed on Android device.

lib: This directory contains native libraries mostly written in C/C++.

assets: This folder contains files such as music, video.

Once we decompile our apk via apktool reverse engineer will see APK contents listed above and additionally smali codes. AndroidManifest.xml will be in a human-readable format. Smali is human-readable format of Dalvik byte code and refers to the Android instructions. We can do an analogy between assembly code in C/C++ and smali in Android. Smali is like assembly language in different computer architectures. Then, in order to analyze classes.dex files we have two options. First, we can use dex2jar tool in order to translate dex file into jar then use JD-GUI. Secondly, we can use jadx-gui tool so that we do not have to use dex2jar tool and JD-GUI separately. You can use just Jadx-GUI instead of using both dex2jar and JD-GUI. Upon completing these steps we will be able to see corresponding Java source code. Besides, there is a folder called smali once we decompile apk with the help of apktool. In this folder, there are smali codes of the application. If there is an obfuscation on Java code, you can use both obfuscated Java code and smali code so that you can pair codes and as a result, you figure out obfuscated code.

One of the key factor for success while reverse engineering is to find out where to start for analysis. Android applications can be large; therefore, reverse engineers cannot analyse all aspects of applications. For this reason it is essential for reverse engineers to know where to start their analysis. There are three main bases regarding where to start analysis. First, what is your objective while reverse engineering? In most cases, we do reverse engineering to answer a specific question. In this paper, these are what is the algorithm generating OTP value, where to fetch the shared secret seed value, whether there is device binding, the process of generating OTP value and more. We should remember what our objective is and go back to it when we are lost while reverse engineering. Secondly, reverse engineers should track API calls. Reverse engineers should keep track of API calls related to their objective. Lastly, application entry points. Sometimes, we cannot know where to start the analysis. In this case, application entry points are good starting points [27].

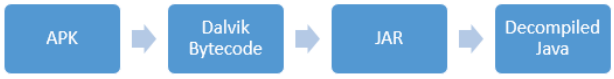

Fig 3: Android Reverse Engineering Process

Our objective in this paper is to fetch shared secret seed value in cleartext format from both storage and memory. We will perform steps as depicted in Figure 3. As a result, we see pseudo source code, then search for keywords like secret, totp, hotp, seed, hash, generate and more. What is next, we investigate API calls and do API call traceback to understand how they operate and look for clues regarding shared secret seed value. In addition, we investigate class definitions and instances in code to be able to see file directory storing shared secret seed value. In AndroidManifest.xml, there are activities related to screen displaying current OTP value. Also, we investigate intent-filters and actions related to those activities.

I. Android Reverse Engineering Tools

In this section, we discuss necessary tools required for our threat model and attack simulation. We install static analysis tool to perform analysis without running applications. In

addition, we install dynamic analysis tools to perform analysis while applications are running.

- **apktool:** Apktool, used for reverse engineering of closed source compiled APKs, can decode Android applications almost to its original form and rebuild them after making some modifications. In our research, we use apktool to decompile applications and analyze AndroidManifest.xml, classes.dex, libraries, certificates etc. In addition, we use apktool to rebuild the application after making some modifications on smali code in order to test anti-tampering resistance. This process is also known as re-packaging.
- **adb:** ADB, standing for Android Debug Bridge, is a tool letting you communicate with Android Emulator, in our research, Genymotion. In system administration world, it is like SSH. We use adb in an attempt to perform tasks such as installing apps, remote connection to Android emulator, directory traversal on Android file system, analyzing of application specific data directories under /data/data, pushing and pulling files to/from Android smartphone. To sum up, it provides Unix shell. It works based on client-server model. Server listens client to connect.
- **Jadx-GUI:** Jadx-GUI, Dex to Java decompiler, is a tool used for reverse engineering of Android applications. It is a GUI-based tool used to produce Java source code from Android dex and apk files. To be able to decompile dex classes and decode AndroidManifest.xml we generally use JADX-GUI tool. Moreover, it includes built-in de-obfuscator in an attempt to de-obfuscate ProGuard usage. With the help of GUI based features, we can perform things like full text search, find usage, jumping to declaration, viewing decompiling code with highlighted syntax.
- **Android Studio:** Android Studio is the official Integrated Development Environment (IDE) for Android app development, based on IntelliJ IDEA and is required to build Android applications. On top of IntelliJ's powerful code editor and developer tools, Android Studio offers more features enhancing your productivity when building Android apps. We use Android Studio in order to analyze .hprof files, that is, heap dump files.
- **Memory Profiler:** Memory Profiler is a component in Android Profiler that helps you identify memory leaks. It shows a realtime graph of your app's memory usage and lets you capture a heap dump, force garbage collections, and track memory allocations.
- **JD-GUI:** JD-GUI is a standalone graphical utility that displays Java source codes of .class files. You can browse reconstructed source code with JD-GUI for instant access to methods and fields. It allows you to browse class files and Java modules hierarchy.
- **Frida:** Frida is a dynamic instrumentation toolkit for developers, reverse engineers and security researchers allowing them to inject JavaScript code at runtime. This tool allows researchers to hook into applications and performs run time analysis. Frida's capabilities include but not limited to run time application analysis, instantiate Java objects, overwrite Java method implementations, scan process memory for occurrences of a string, enumerate native function calls.
- **Objection:** Objection, short for object injection, is a runtime mobile exploration toolkit allowing reverse engineers to bypass SSL pinning, enumerate keystore, explore classes, patch applications, interact with file system, discover loaded classes and more. Objection is powered by Frida dynamic instrumentation toolkit.
- **SQLite Browser:** Android has built-in SQLite database implementation. SQLite is an open source database allowing users and developers to perform CRUD (Creating, Reading, Updating and Fetching) operations on database. SQLite Browser is a tool used to connect and perform database operations on SQLite database.

IV. Problem Definition, Threat Model and Assumptions

Recall that OTP value is a product of TOTP algorithm with mainly two parameters. These two parameters are the same for both server side and mobile application, which are shared secret seed value and current timestamp. To be able to generate OTPs registration process also known as provisioning must be done. During registration process shared secret seed value is agreed upon. Upon completing provisioning, shared secret seed value is shared between two entities.

When users log into their accounts or execute an 2FA protected transaction they have to prove that they possess OTP value. Since shared secret seed value, HASH function and current timestamp are same for two entities, both server and the mobile authenticator application generate the same OTP value. The problem in this case is that everything is known except shared secret seed value to attackers. If attackers somehow fetch shared secret seed value then they may bypass second factor, which is something you know for our situation. We can conclude the main problem as "knowing shared secret seed value might be enough to bypass the second factor if second factor is an OTP value produced by software based authenticator applications".

As for assumptions, we suppose the fact that provisioning step has been successfully finished and mobile authenticator has been closed at least once. It is much more realistic once mobile authenticator application has been closed at least once. Then continue producing OTP value. Furthermore, in this research attacker has root access to system. This can be achieved as follows: The attacker exploits a vulnerability on the system, physical access to rooted Android device, device is rooted. Nowadays it is common to find vulnerabilities that lead to remote code execution and vulnerabilities to evelate the privileges. [28] [29] [30] [31] [32] [33]. Also, rooting the Android device is not uncommon. Some applications on various store including Google Play Store require root permission to be executed. As a result, this is a realistic scenario for today's environment.

In our research, our threat modeling tries to fetch secret key both from storage and memory. We executed black-box tests, meaning we do not have any knowledge and source code regarding applications. We take APK file and start analyzing. We use Genymotion as emulator application with Google Nexus 5, Samsung Galaxy S 10 as smartphones, with API level of 23 and 29 respectively. We have installed necessary tools (apktool, adb, jadx-gui, Android Studio, Frida, Objection) on Ubuntu VM, virtualized via Virtual Box. In addition, we install Android Studio on Windows 10 client machine. These machines are on same subnet. Our threat modeling is as follows: After downloading apk, we decompile the application. Next, we perform static code analyzes in order to understand apk's functionalities, methods and deeper understanding of code in relation to OTP operation process. Once we gain understanding of APK we install apk into emulator by using adb. What is more, we perform registration process by entering manually shared secret seed value. In our research, we enter shared secret seed value manually if application has that option to make sure that we obtain shared secret seed value truthfully. Next, we close authenticator application. Then we close mobile phone. Next we open phone and we are ready to attack against shared secret seed value. Once we complete these steps, we proceed to the next step.

A. Threat Modeling of Fetching Shared Secret Seed Value from Memory

There are two different procedures when it comes to fetching shared secret seed value from memory. The first procedure is as follows, which is by using adb tool.

1) Currently, we should see OTP value on application screen.
2) When we see it, we will connect to emulator by using adb with root.
3) Identify authenticator's process id (PID) via ps command.
4) Execute exit command in order to come back to your host. Note that you are still connected to the emulator.
5) Create a heap dump by using adb. Command is as follows: adb shell am dumpheap PID /data/local/tmp/appname.hprof.
6) For analysis, pull the generated file to your host machine.
7) Analyze heap dump on Android Studio by using Memory Profiler feature.

The other way is to directly use Android Studio. Here are the steps that we follow in order to fetch shared secret seed value from memory.

1) Open Android Studio.
2) Click Android Profiler button.
3) Hit "start a new profiling session" button.
4) Select appropriate emulator and then select authenticator application.
5) It starts profiling the application.
6) Prepare initial setup of application and provisioning step.
7) Go back to Android profiler and select the memory section.
8) Hit Java dump heap. Then analyze the memory.

You can take many heap dumps as you wish. But you can not take consecutive seconds of dump heap. You take a heap dump, at least 3 - 4 seconds later you can take another heap dump. This is main limitation of open source tools, which is discussed in Shortages of Our Threat Modeling section.

B. Threat Modeling of Fetching Shared Secret Seed Value from Storage

Here are the steps that we follow in order to fetch shared secret seed value from storage.

1) Currently, we should see OTP value on application screen.
2) Analyze application in order to find where mobile authenticator stores shared secret seed value.
3) Navigate to that directory on file system.
4) Analyze the folder containing shared secret seed value.

C. Shortages of Our Threat Modeling

In our research, we only use open source tools. We did not use any commercial tool. The main shortage of our threat model comes from fetching shared seed value from memory. You take a heap dump by using adb with the following command : **adb shell am dumpheap PID /data/local/tmp/appname.hprof**. You hit these commands any time you want and you take a heap dump of that time, such as t0. At least, after 3 4 seconds, you can take another heap dump. Unfortunately, you can not take heap dump of consecutive second. This is our main limitation regarding fetching shared secret seed value from memory. However, in order for authenticator to generate OTP, shared secret seed value must be on memory in cleartext at some point. Without consecutive heap dumps of every second, you may not be able to fetch the shared secret seed value in cleartext format in memory.

V. Attack Simulation

A. Chosen Applications

Chosen applications are as follows: Google Authenticator, Microsoft Authenticator, Red Hat Free OTP, Blizzard Authenticator, Epic Authenticator, Oracle Authenticator, Oracle Authenticator, SAP Authenticator, Sophos Authenticator, Twilio Authenticator, Pixplicity Authenticator and SaasPass Authenticator.

Although there are more applications available on Google PlayStore it is infeasible to analyze all mobile authenticator applications. It is enough to choose 11 application regarding representativeness. We choose eleven applications according to the popularity on Google PlayStore. Besides, they are all free to download, install and use. Also, they all implement TOTP and HOTP algorithm.

B. Security Measures Investigated (What and How?)

In this section, we describe how we test whether additional security mechanisms are applied on mobile authenticator application.

ProGuard Obfuscation: ProGuard shortens names of your app's classes and members with simple letters such as A, B, and junky strings such as p003io, p002 so on and so forth. Upon decompiling application if we see class names, method names, instance variable names or parameters we can conclude that there is ProGuard applied.

Advanced Obfuscation: After checking de-obfuscation button under Tools section at Jadx-GUI tool, it de-obfuscates the ProGuard obfuscation to some extend. However, if there is advanced obfuscator applied, there will be mostly junky, irrelevant class names, instance variables, method names, parameters as well as dead code in order to astonish reverse engineer. We test whether applications have advanced obfuscator in place in this way.

Android KeyStore: We use objection in order to learn if there is Android Keystore. We setup and configure Frida tool. It is ready to attach mobile authenticator applications. Then we use objection, which is powered by Frida. Once mobile authenticator applications run, we explore application by using objection related to KeyStore usage.

Anti-Tampering / Repackaging: We test anti-patching by modifying some smali codes. After doing modification we build the application with apktool again. If it allows us to modify application, we can conclude that there is no anti-tampering / repackaging implemented.

Encryption Sensitive Information on Storage: If shared secret seed value is encrypted in storage we can conclude the fact that there is encryption in place.

Device Binding: After decompiling source code we look at device specific values such as imei number, mac address and audit these values regarding where and how they are used.

C. Steps of Simulated Attacks

It is infeasible to simulate attacks against all mobile authenticator applications listed above. We will simulate attack against Google Authenticator for both storage and memory.

Recall our assumption that we can execute commands with root privileges.

Once we configure related reverse engineering tools, fulfill provisioning process as shown in Figure 4, close Google Authenticator app, close mobile phone and start phone, then we are ready to execute attacks against shared secret seed value on Google Authenticator.

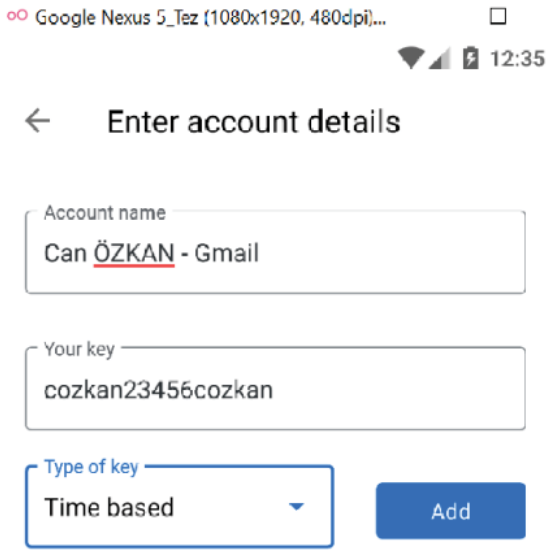

Fig 4: Google Authenticator Provisioning Process

Simulated Attacks with respect to Storage

After reversing Google Authenticator application in a static manner, we come up with opening a sqlite3 database whose name is **databases** as seen in Figure 5. The rest is to analyze the database in an attempt to fetch shared secret seed value in cleartext as seen in Figure 6 and 7.

Fig 5: Google Authenticator Static Reverse Engineering

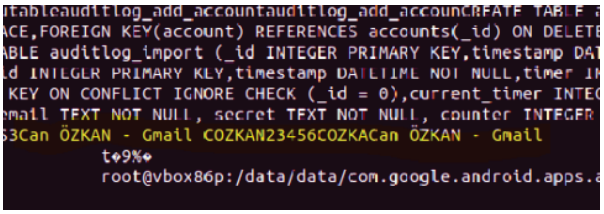

Fig 6: Google Authenticator Fetching Shared Secret Key from Storage

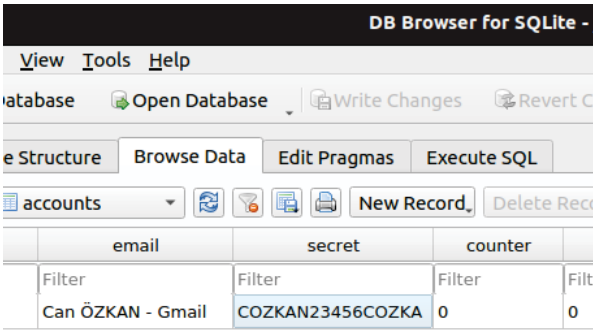

Fig 7: Importing Google Authenticator's database into SQLite Browser

Simulated Attacks with respect to Memory

In this section, we attack Google Authenticator against shared secret seed value. After fulfilling our assumptions, we are ready to execute our threat modeling. We perform heap dump operation on time t. What we mean is we do not have any rule regarding timing. We take sample heap dumps randomly on time t. Time t is chosen randomly. We take random heap dumps. In other words, we do not have specific algorithm regarding heap dump timing. The point in this case is we cannot take heap dump of every consecutive second. Once you perform heap dump, you can take another heap dump at least 3 - 4 seconds later. Having took a heap dump randomly on time t, we analyze it on Android Studio's Android Profiler component as seen in Figure 8 and see the shared secret seed value in cleartext format.

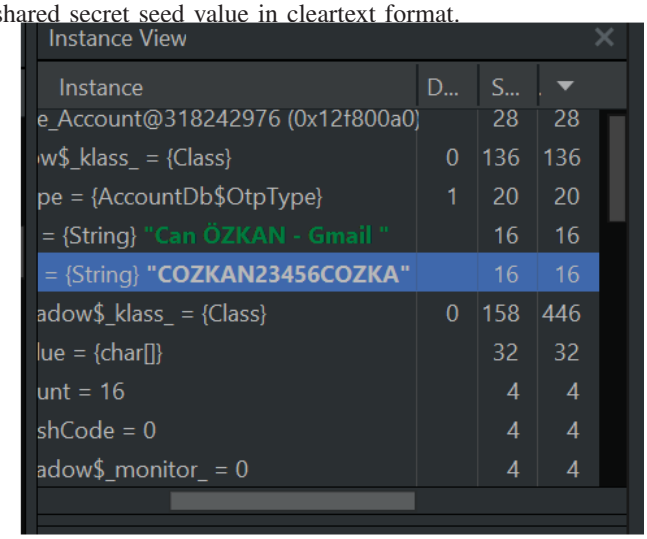

Fig 8: Google Authenticator Fetching Shared Secret Key from Memory

VI. Findings

Red Hat Free OTP: This authenticator, with version Free OTP 1.5, holds the shared secret seed value on storage at directory /data/data/org.fedorahosted.freeotp/shared_prefs/tokens.xml with encoded format. Authenticator holds the shared secret seed value on memory on time t as encoded. T is chosen randomly.

SAP Authenticator: We analyzed SAP Authenticator with version 1.2.8. There is a database file at /data/data/com.sap.csi.authenticator/databases/DataVault. This file is encrypted. So, we could not retrieve the shared secret seed value from storage. Also, we look at other directories on storage. On the other hand, we could not retrieve the value from memory on time t. T is chosen randomly. Moreover, in this authenticator there is a method called getIMEI(), it returns an empty string. This method is never called somewhere in the codebase. Basically, it is a deadcode.

Sophos Authenticator: We analyzed Sophos Authenticator with version 3.2. Shared secret seed value is in cleartext at /data/data/com.sophos.sophtoken/databases/databases. Also, we would be able to fetch the shared secret seed value from memory on time t.

Oracle Authenticator: This authenticator, with version 9.2, holds the shared secret seed value on storage at /data/data/oracle.idm.mobile.authenticator/databases/OMADb.db file in encrypted format. We see the email address in cleartext format; however, we see shared secret seed value in encrypted format. When it comes to fetching shared secret seed value from memory, we would be able fetch the value. OAMAccount class' instance holds the values in unencrypted form on t time. Moreover, this authenticator fetches specific values such as MAC address, imei number, android_id, then passes them into compliance related methods. While those values are not used for device binding, they are used for compliance related methods.

Epic Authenticator: In this authenticator, we would be able to fetch the shared secret seed value from both storage and memory in cleartext format. Seed value is stored at /data/data/com.epic.authenticator/shared_prefs/com.epic.otpkit.keystore.OtpAccountStore_ACCOUNTS.xml file in cleartext. At memory, OtpAccount class' object holds the shared secret seed value in cleartext format on time t.

Pixplicity Authenticator: We analyzed Pixplicity Authenticator with version 1.0.4. In this authenticator, we could not be able to fetch the shared secret seed value neither both storage nor memory in cleartext directly. Once we analyze deeply, information such as shared secret seed value, token type, issuer label and more are inserted into /data/data/com.pixplicity.auth/databases/tokens.db. Next, we analyze that secret value is stored in the type of BLOB. That's why we could not fetch shared secret seed value in cleartext directly. On the other hand, shared secret seed value is broken down into byte arrays. They perform what is known as security by obscurity for memory side.

Blizzard Entertainment Authenticator: We analyzed Blizzard Authenticator with version 2.4.2.9. We could not getch shared secret seed value in cleartext format. Moreover, this authenticator fetches device specific values such as imei number, MAC address, android ID and mobile phone model. Once they fetch value, they are passed as parameter to getUUID() method. In this method, these values are performed string concatanetion operation and the method returns hash value of string. Once you login a device, it controls whether the

TABLE I
AUTHENTICATOR APPLICATIONS AND THEIR ADDITIONAL SECURITY MECHANISMS

Authenticator Apps	Free OTP	SAP	Sophos	Oracle	Epic	Pixplicity	Blizzard	Twilio Authy	Google	Microsoft	SaasPass
ProGuard	-	-	-	+	-	-	-	-	-	-	-
Advanced Obfuscator	-	-	-	-	-	-	-	-	-	-	-
Device Binding	-	-	-	-	-	-	-	-	-	-	-
Anti-Repackaging	-	-	-	-	-	-	-	-	-	+	-
Insecure Storage	-	-	+	-	+	-	-	+	+	+	-
Android KeyStore	-	-	-	+	-	-	-	+	-	+	-

TABLE II
AUTHENTICATOR APPLICATIONS AND THEIR SHARED SECRET SEED VALUE FROM STORAGE AND MEMORY

Authenticator Apps	Free OTP	SAP	Sophos	Oracle	Epic	Pixplicity	Blizzard	Twilio Authy	Google	Microsoft	SaasPass
Fetch Seed from Storage	-	-	+	-	+	-	-	+	+	+	-
Fetch Seed from Memory	-	-	+	+	+	-	-	+	+	+	+

device the user is currently logging in has changed or not. If changed, user redirects to the restore page. Although these values are not used for adding randomness to the algorithm, they are used for account security itself.

Twilio Authy Authenticator: This authenticator, with version 24.3.4, holds the shared secret seed value on storage at /data/data/com.authy.authy/shared_prefs/com.authy.storage.tokens.authenticator.xml file. In memory, OtpAuthPayload class' object holds the shared secret seed value in cleartext format on time t.

Google Authenticator: In this authenticator with version 5.10, we would be able to fetch the shared secret seed value from both storage and memory in cleartext format. Seed value is stored under /data/data/com.google.android.apps.authenticator2/databases/databases file in cleartext. In memory, AutoValue_Account class' object holds the shared secret seed value in cleartext format on time t.

Microsoft Authenticator: In this authenticator with version 6.2006.4198, we would be able to fetch the shared secret seed value from both storage and memory in cleartext format. Seed value is stored under /data/data/com.azure.authenticator/PhoneFactor-wal. In memory, SecretKeyBasedAccount class' object holds the shared secret seed value in cleartext format on time t.

SaasPass Authenticator: In this authenticator with version 2.2.24, it is not possible to fetch the shared secret seed value from storage; however, it is possible in memory. In memory, OTPAuthBarcode class' object holds the shared secret seed value in cleartext format on time t.

As seen in both Table 1 and Table 2, the result of applications tested can be summarized as follows: 5 applications out of 11 hold shared secret seed value in cleartext format on storage. In other words, 5 / 11 applications do not apply any kind of encryption, encoding at all. 7 applications out of 11 hold shared secret seed value in cleartext format on memory in time t, which means more than half of applications do not apply any kind of encryption or encoding operation for memory. Also, Blizzard Entertainment Authenticator is the only application that warns the user about the fact that device is rooted and it may cause security issues. None of the applications except Microsoft Authenticator applies software anti-tampering. We could successfully re-package applications except Microsoft. As a consequence, attackers can decompile applications, add malicious payload, repackage applications, sign applications and distribute applications with malicious purpose or they can repackage the applications so that they remove security mechanisms such as root detection. What is more, it is important to note that no application applies advanced obfuscator in order to avoid skilled reverse engineers. By nature, there is no device binding regarding OTP generation phase. All authenticators we tested implement TOTP and HOTP. In addition to these, Blizzard Entertainment Authenticator fetches MAC address, IMEI number and phone model in order to secure not OTP value but OTP user's account.

While Android Keystore is de facto method in order to secure cryptographic keys, only 3 out of 11 applications implement Android Keystore.

VII. DISCUSSION

Android Keystore is used to store cryptographic keys in a secure manner. Android Keystore generates a key, then stores it securely. With that key, applications encrypt sensitive information, in this case, shared secret seed value. Although Twilio Authy Authenticator application possesses Android Keystore, it stores shared secret seed value in cleartext format under shared preferences. This may indicate a misuse of Android Keystore or this may indicate backup issues. It is a problem when encrypted secret keys are imported to device B as long as decryption key is only in device A. The first thing that came to our mind with respect to key store is that it may be about backup feature. We thought when applications had Android Keystore, backup feature could not work properly due to encryption keys. However, we see that although Microsoft Authenticator and Twilio Authy Authenticator possess key store, they have backup feature. But in this case although they implement Android Keystore, they store shared secret seed value in cleartext format.

We also note that all mobile authenticator applications apply TOTP and HOTP algorithm regarding producing OTP value.

These algorithms do not require any device specific values such as MAC address, IMEI number, leading OTP value accounts more compact and portable to another device in case users may lose their mobile device, or buy new mobile device. If an application requires device specific unique values, it will be more difficult to transfer/backup OTP accounts to another device.

Philip Polleit and Michael Spreitzenbarth's work include but not limited to fetching shared secret seed value from storage. However, we also showed that they can be fetched on memory as well. Their work emphasizes on whether cloning the database is possible, device integrity check, PIN protection, Secure SSL Connection. Shared secret value can be attacked over network by listening to network. If shared secret seed value is transported over an unencrypted channel, it can be fetched in cleartext as well. However, our threat model does not cover that. As a matter of fact, we emphasize on using static and dynamic reverse engineering techniques in order to fetch shared secret seed value. In addition, we investigate whether or ProGuard and advanced obfuscators are in place. Because obfuscating the source code make static analysis harder for reverse engineers. Besides, Android Keystore is a de facto method to store cryptographic keys. Another issue is that cryptographic keys need to be stored securely so that reverse engineers cannot reach them easily. Our work concentrates on more reverse engineering side.

VIII. Implications of Our Results

In most situations, it is easier to fetch shared secret seed value from memory than storage. In order to protect shared secret seed value on storage, there are various options like encryption with key stored in keystore, using White-Box Cryptography. On the other hand, shared secret seed value may be secured by security by obscurity on memory. Although security by obscurity is not a perfect solution, it may deter reverse engineers a bit.

In the event applications do not apply ProGuard and advanced authenticators, it is likely that reverse engineers can analyze and understand the application. Also, authenticator applications should not only rely on built-in protections that are available on Android OS. Applications should also take precautions against software tampering. In most cases, we would be able to re-package applications. We could add a meterpreter reverse tcp payload, sign application and distribute application. Once users install repackaged APK, they will connect back to our command and control server. On the other hand, this is also used to bypass security mechanisms such as root detection methods. We could delete root detection methods, repackage the application and continue our analysis.

IX. Conclusion and Future Work

Firstly, it is highly important to deploy two factor authentication in order to increase account security. Because it adds an additional factor to improve security posture compared to single factor authentication.

As shown in Table 2, in most situations it is easier to fetch shared secret seed value from memory.Using Android Keystore or White Box Cryptography is sufficient to protect shared secret seed values on storage.

Few applications have additional security mechanisms. The others solely rely on built-in security mechanisms available on Android OS. For this reason, devices having exploitable vulnerabilities, users with low security awareness and rooted devices are prone to be vulnerable to security attacks especially against shared secret seed values. We can conclude that once shared secret values are fetched by attackers for these authenticators, 2FA might suddenly downgrade to single factor authentication.

Future work may include developing a tool that takes heap dump of every consecutive second.

References

[1] J. U. Sheetal, Purohit.D.N. and V. Anup, "Increase in number of Online Services and Payments through Mobile Applications Post Demonetization", 2019, pp 35-36

[2] R. Jones, "Mobile banking on the rise as payment via apps soars by 54 % in 2015", 2016 [Online]. Available: https://www.theguardian.com/business/2016/jul/22/mobile-banking-on-the-rise-as-payment-via-apps-soars-by-54-in-2015

[3] "What is Two-Factor Authentication" [Online] Available: https://authy.com/what-is-2fa/

[4] W. S. Park, D. Y. Hwang and K. H. Kim, "A TOTP-Based Two Factor Authentication Scheme for Hyperledger Fabric Blockchain", 2018

[5] H. B. Hasbullah, I. A. Lawal, A. A. Mu'azu, and L. T. Jung, "A TOTP-Based Enhanced Route Optimization Procedure for Mobile IPv6 to Reduce Handover Delay and Signalling Overhead", 2013

[6] E. Huseynov, J. M. Seigneur, "Hardware TOTP tokens with time synchronization", 2019

[7] B. U. I. Khan, R. F. Olanrewaju, F. Anwar and M. Yaacob, "Offline OTP Based Solution for Secure Internet Banking Access", 2018

[8] P. Polleit and M. Spreitzenbarth , "Defeating the Secrets of OTP Apps", 2018

[9] B. Mueller, "Hacking Soft Tokens Advanced Reverse Engineering on Android",2016 [Online]. Available: https://gsec.hitb.org/materials/sg2016/whitepapers/Hacking%20Soft%20Tokens%20-%20Bernhard%20Mueller.pdf

[10] Poonguzhali P., P. Dhanokar, M. K. Chaithanya, and M. U. Patil, "Secure Storage of Data on Android Based Devices", 2016

[11] T. Cooijmans, J. D. Ruiter and E. Poll, "Analysis of Secure Key Storage Solutions on Android", 2014

[12] T. Cooijmans, "Secure key storage and secure computation in Android", 2014, pp 15-51

[13] G. Kasagiannis and C. Dadoyan, "Security Evaluation of Android Keystore", 2018, pp 21-29

[14] R. K. Konoth, B. Fischer, W. Fokkink, E. Athanasopoulos, K. Razavi and H. Bos, "SecurePay: Strengthening Two-Factor Authentication for Arbitrary Transactions", 2020

[15] I. Permana, M. Hardjianto and K. A. Baihaqi, "Securing the Website Login System with the SHA256 Generating Method and Time-based One-time Password (TOTP)", 2020

[16] J. X. Zhao, "Research and Design on an Improved TOTP Authentication", 2013

[17] Azhari, L. Q. Muhammad and C. G. N. Tama, "Android Mobile Banking Application Security from Reverse Engineering and Network Sniffing", 2016, pp 462-465

[18] S. R. Kotipalli and M. A. Imran, "Hacking Android", 2016, pp 87-101

[19] "System and Kernel Security" [Online] Available: https://source.android.com/security/overview/kernel-security

[20] P. K. Granthi and S. M. Bansode, "Android Security: A Survey of Security Issues And Defenses", 2017, pp 543-544

[21] "Signing Your Applications" [Online]. Available: https://stuff.mit.edu/afs/sipb/project/android/docs/tools/publishing/app-signing.html

[22] "Shrink, Obfuscate, and Optimize Your App" [Online]. Available: https://developer.android.com/studio/build/shrink-code

[23] ”Android Keystore System” [Online]. Available: https://developer.android.com/training/articles/keystore#java
[24] D. Buttan, ”Hacking the Human Brain: Impact of Cybercriminals Evoking Emotion for Financial Profit”, 2020, pp 19-20
[25] A. S. Chaudhari, ”Security analysis of SMS and related technologies”, 2015, pp 34
[26] E. Eilam, ”Reversing: Secrets of Reverse Engineering”, 2011, pp 3-12
[27] M. Stone, ”Android App Reverse Engineering 101” [Online]. Available: https://ragingrock.com/AndroidAppRE/
[28] S. Bakken, D. Weinstein, ”OnePlus Device Root Exploit: Backdoor in EngineerMode App for Diagnostics Mode”, 2017, [Online]. Available: https://www.nowsecure.com/blog/2017/11/14/oneplus-device-root-exploit-backdoor-engineermode-app-diagnostics-mode/
[29] C. Davenport, ”OnePlus left a backdoor in its devices capable of root access”, 2017, [Online]. Available: https://www.androidpolice.com/2017/11/15/oneplus-left-backdoor-devices-capable-root-access/
[30] J. Gu, V. Zhang, and S. Shen, “ZNIU: First android malware to exploit dirty cow vulnerability,” 2017. [Online]. Available: https://www.trendmicro.com/en_us/research/17/i/zniu-first-android-malware-exploit-dirty-cow-vulnerability.html
[31] D. Goodin, ”Attackers exploit 0-day vulnerability that gives full control of Android phones”, 2019, [Online]. Available: https://arstechnica.com/information-technology/2019/10/attackers-exploit-0day-vulnerability-that-gives-full-control-of-android-phones/
[32] E. Xu and J. C. Chen, ”First Binder Exploit Linked to SideWinder APT Group”, 2020, [Online]. Available: https://www.trendmicro.com/en_us/research/20/a/first-active-attack-exploiting-cve-2019-2215-found-on-google-play-linked-to-sidewinder-apt-group.html
[33] L. Davi, A. Dmitrienko, A. R. Sadeghi and M. Winandy, ”Privilege Escalation Attacks on Android”, 2010, pp 5-6