appears in: Proceedings of the 11th International Conference on IT Security Incident Management & IT Forensics (IMF 2018)

Defeating the Secrets of OTP Apps

Philip Polleit and Michael Spreitzenbarth
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Abstract

Despite the increasing number of cases of data theft (such as Equifax), the classic password is still in many places the sole security feature for user authentication. However, numerous possibilities for extending this now anachronistic form of access control already exist. One such option is the use of *one-time passwords (OTP)*. These passwords are increasingly used for additional authentication (in addition to user name and password) of the respective user to service providers on the Internet and the applications that generate these are therefore referred to as so-called *two-factor authentication apps (2FA apps)*. This paper investigates 16 such 2FA apps for the Android operating system and focuses on the extent to which these applications can offer a similar level of protection when compared to classical hardware tokens (e.g., YubiKey, SecurID-Authenticator).

Keywords

2FA; OTP-Apps; Android; Mobile Forensics;

I. Introduction

A. Motivation

The current information age is essentially characterized by the digital form of its information transmission. It no longer transmits physically embodied things such as letters, but rather their digital counterparts – i. e., PDFs, scans, emails – are transferred. On the one hand, digital transformation takes away the haptics of documents and, on the other hand, offers a wide range of editing and reworking options. These circumstances make it difficult for the recipient of a digital message to establish the authenticity of the content and its sender. This opens up the potential for abuse. Fake news is ultimately a form of expression that is currently leading to a *digital loss of trust*. The information age is therefore essentially dependent on secure mechanisms for user authentication.

“The Federal German Criminal Police Office has found a collection of approximately 500,000,000 spied access data in an underground economy platform on the Internet. The data consists of email addresses and associated passwords.” [1]

For a long time, the combination of user name and password served as the sole security feature for user authentication in many places. From a historical point of view, this is simply the consistent adoption of the principle of the watchword from the analogue world. A look at the most frequently used German password in 2017 questions the suitability of this transfer to the digital world. It reads: `123456` and is followed by `123456789` [2].

The above-mentioned press release of the Federal German Criminal Police Office is to some extent exemplary and serves as an indication of a computer forensic phenomenon. Until a few years ago, it seemed sufficient to use access protection in the form of simple password queries, but today the situation has changed considerably. The reasons for this vary in nature and include people who tend to be more comfortable (using trivial passwords) and, at the same time, the increasing opportunities for cybercriminals to obtain access data (e. g. reduced costs for resources to crack bruteforcing of obtained password hashes).

For example, they make use of phishing by gaining the access data via websites that are deliberately imitated for the purpose of confusion, or by using methods to obtain password databases from service providers and evaluate their contents accordingly. A prominent example of the theft of data carried out directly at the service provider is the hack of the PlayStation Network (PSN), dating from 2011, which took place from 17 to 19 April 2011, when unknown perpetrators gained access to 77 million customer accounts, including credit card data [3]. A current example is the theft of data at Equifax Inc. from the beginning of September 2017. The data (user name, password, date of birth, address, credit card number, driver’s license number, etc.) of 143 million socially insured persons from the USA, UK and Canada had been stolen [4].

In addition to the service providers directly affected, collateral damage that can hardly be calculated occurs wherever the user is registered with the same password. This allows data thieves to access other services and resources of the injured users.

While the choice of weak passwords, the multiple use of the same password for different services, and the disclosure of access data on phishing sites are due to the user’s area of responsibility, data theft from the service provider’s servers

behaves differently. Here the user entrusts his data to the provider for safe storage. It is completely dependent on its security implementations.

Recognized cases of data theft show that some vendors do not salt their customers' passwords stored on their servers, a fact that has long been a standard feature. This process increases the entropy of the password hash. Such stored passwords are considered to be theft-proof, since only their irreversible equivalent (its hash value) is present.

Despite the fact that salting is a simple process to protect passwords, the online portal LinkedIn stored the passwords of its members on its servers until June 2012 without salt. Only after more than 117 million unswitched password hashes (SHA-1) had been stolen during a data theft, LinkedIn started salting their password hashes [5]. A similar behavior pattern has been noticed for the social network last.fm, where 43 million unswitched password hashes (MD5) were stolen in March 2012 [6] and who started to salt their password hashes afterwards – to name just two examples.

Multi-factor authentication (MFA) can be used to overcome the aforementioned weaknesses. It is therefore natively supported by an increasing number of companies (including Apple, Google (since 2010) and Microsoft) [7]. MFA guarantees proof of identity using several independent and different factors (components).

These factors can be grouped into three categories:

- The category *Knowledge* includes incorporated information such as user names, passwords or PINs.
- While the category *Being* is an element inseparable from the user, such as his fingerprint, or another biometric date.
- Finally, the category *Possession* describes things physically available to the user. This is e.g., an EC card or a hardware token.

From the latter category, the hardware token is the most common. This generator can be used to create a *one-time password (OTP)*. Since it is only dynamically generated at the time of its use (on client and server side), it cannot be stolen from the server as described above, and due to its short half-life (usually 30 seconds of validity) it also protects immediately against replay attacks (the re-use of a previously eavesdropped OTP for unauthorized authentication with the web server).

Hardware tokens must always be carried by the user for authorization purposes and are of course subject to a risk of loss. For the random finder, they are of no value as long as they cannot be assigned to a specific account (e.g., by lettering) and the finder also knows the corresponding password. In addition, the procurement and maintenance of the equipment is associated with financial and logistical costs. For the aforementioned reasons, smartphones are increasingly being used as quasi tokens, especially by private users, but also in business environments. These are usually carried along by the user anyway and offer several possibilities of proof of identity. The OTP sent by the server can be received via email or SMS (cf. mTAN procedure). However, this paper focuses on the use of smartphones to generate OTP using specific applications, which are often referred to as 2FA (two-factor authentication) apps. Since this is a pure software implementation, this is also referred to as tokenless two-factor authentication.

Google's Android operating system was chosen as the platform for the investigation. The reason for this is the far greater prevalence with a current market share of 87.7% worldwide, compared to Apple iOS with 12.1% [8]. The subject of the study are the 16 most popular 2FA applications from Google Playstore, according to the users' rating.

B. Forensic Use

One of the central questions of any criminal procedure under the rule of law is certainly the question of causality, i.e., the question of the responsible perpetrator. In connection with computers as a means of committing a crime, or networks as a crime scene, user authentication becomes the focus of computer forensic consideration. Computer forensics experts will be confronted with the question of who was sitting at the computer at the time of the crime or whether the login credentials on the later compromised server were used by the authorized person or misused by third parties. Otherwise, the chain of evidence ends at the interface between the incriminated hardware and its users (the perpetrator actually responsible).

As 2FA apps and the OTPs they use are being used more and more frequently to further secure the process of user authentication, they will be increasingly computer forensic relevant in the future.

Otherwise, the defense strategy could in future consist of simply asserting that someone has stolen the OTP (or the shared secret) from the cell phone of the accused and then acting on his behalf. For computer forensics, it is therefore necessary to deal with the questions of the integrity of the OTP procedures (and / or their implementation in the 2FA apps).

The present work highlights the current weaknesses of the examined 2FA apps and shows their potential for abuse. Based on the identification of the relevant files and their specific storage paths, this work can serve as a compendium for the identification of identity theft cases.

C. Results

The results of the 16 2FA apps examined can be summarized in advance as follows. In terms of encryption, two apps only used a different notation (in byte values) to store the shared secret. Six of the 16 apps actually encrypted the shared secret.

Thus, half of the apps ($n = 8$) did not secure the “shared secret” at all! In the test, only one application (DUO) detected during an automatic safety audit and noted that the test device was rooted and could cause problems. In nine cases, the ”shared secret” could be transferred from one device to another and identically OTP could be generated there. Only three of the sixteen apps offered an additional security layer (e.g. in the form of a PIN query at program start) in the test.

D. Paper Outline

In Section II we briefly describe the mathematical basis for the calculation of OTP. We also provide an up-to-date market overview of the 2FA apps included in Googles PlayStore to better assess their relevance. Followed by this introduction, Section III shows the experimental setup and presents the results of the 16 analyzed applications. Finally, the summary is given in Section IV.

II. Background

Within this section we want to give a brief overview of the important mathematical background to the readers. This background is important to understand why we will later focus on stealing the secret from an installed 2FA app. Within the second part of this paragraph we want to give a market overview and also describe which apps we focused on for our research.

A. Mathematical Background

The idea of using one-time passwords was formulated in November 1981 by Lamport [9]. In principle, OTP can be generated in different ways. Three types are distinguished and presented below:

- time-controlled method,
- challenge-response controlled method and
- event-driven method.

All three methods only transmit the computation result of their algorithm from the client to the server for authentication. The calculations, which are usually an iterative application of non-reversible, cryptographic hash functions, can include different values (e. g., time, counter) depending on the procedure. Common to all OTP processes is the use of a common secret password. This is exchanged in a separate way during initialization or stored in the hardware token during manufacture. The server now performs the same calculation and compares its result with the transmitted value. If they agree, the client is verified.

Maintaining the confidentiality of the shared password is essential for the procedure. In a formula, the process of generating OTP described in RFC 2289 [10] can be displayed as follows:

$$S = H(r_A \,||\, ggKW) \tag{1}$$

The procedure is initialized by defining the start value S. This is the result of the cryptographic hash function H on the concatenated values of a random number r_A and the common secret password $ggKW$. The latter is called shared secret, the random number r_A is called seed. Now OTP (KW) can be generated by applying the hash function H to the start value S several times iteratively.

$$KW_N = H^N(S) \tag{2}$$

The procedure presented by Haller et al. 1998 in RFC 2289 [10] provides for a decrementing of the counter N by 1 for each generated OTP. Thus, the validation of the OTP KW_{N-1} transmitted by the client on the server side takes place by means of a one-time application of the hash function $H(KW_{N-1})$. If the result matches the stored value for $H(KW_N)$, the OTP is valid. In the time-based one-time password algorithm TOTP (see RFC 6238 [11]), the current time (instead of a counter as shown above) is included in the calculation of the OTP.

Depending on the configuration, the algorithm typically generates a new OTP every 30 seconds. In order to avoid time differences between server and client, the server can also accept the previously valid TOTP. The procedure is based on the Keyed-Hash Message Authentication Code (HMAC, RFC 2104 [12]), a cryptographic message authentication code.

Table I
TOP10 2FA APPS FOR ANDROID

App Name	Number of Downloads	Average User Rating
Google Authenticator	10 to 50 M	4,3
Blizzard Authenticator	5 to 10 M	4,5
Authy 2-Factor Authentication	1 to 5 M	4,3
Microsoft Authenticator	1 to 5 M	4,0
VIP Access	1 to 5 M	3,9
DUO Mobile	1 to 5 M	3,9
FreeOTP Authenticator	100.000 to 500.000	4,6
LastPass Authenticator	100.000 to 500.000	4,2
Okta Verify	100.000 to 500.000	3,3
2FA ONE	1.000 to 5.000	4,2

B. Market Overview of 2FA Apps for Android

End of December 2017 the Google PlayStore shows 238 hits for the search term *2FA Apps*. Table I lists ten of the most popular apps and gives you an impression of their relevance based on the download figures.

The applications are also called software tokens. They use the same algorithms to generate OTP that are also used by hardware tokens. Since they are installed as ordinary Android applications (based on Java), their security is fundamentally dependent on the operating system Android. Regardless of this, some of the apps try to ensure the security of the shared secret by encrypting it.

On closer inspection, the number of applications relevant for the following investigation is reduced by those which are used solely for access to a specific provider (e. g., Blizzard Authenticator, 2FA ONE). Only applications that can manage multiple accounts with different vendors and use the TOTP standard are important for the investigation. This ensures comparability. Finally, the criterion of the respective customer rating is used to determine the most popular applications.

III. EXPERIMENTAL SETUP AND 2FA RESULTS

For our experiments we used two rooted Nexus 5 with Android 5.1.1 and one rooted Samsung Galaxy S3mini with Android 4.1.2. The main use case for the S3mini was to guarantee that cloned apps even run when the hardware of the device is different from the one, the app was installed initially.

During our Research we will focus on the question if cloning a 2FA app is possible and of course, how the secret is stored and protected within the app. To achieve this, we install the app on one of our test devices and connect it to dropbox as our test platform. As soon as the app is working proper, we start to do a forensic analysis of the stored data. After finding the secret and further configuration data we copy the secret to a second device where we installed the same 2FA app.

At this point we record if the app is working and generating identical OTPs, or if just copying the secret is not enough. Followed by this first approach, we try to copy and manipulate configuration variables or device identifiers to let the app think that it is still running on the same device than initially.

Within the following sections we will explain our results for all tested apps an give an overview of all 2FA apps for Android that we analyzed within the results section (see Section III-Q).

A. Google Authenticator

The first of the examined 2FA apps is Google Authenticator in version 4.74. This app stores the shared secret within the folder `/data/data/com.google.android.apps.authenticator2/databases/` in a file called `databases`. This file is a common SQLite database and contains the shared secret and the service name it belongs to in plain text (see Listing 1).

Listing 1. Google Authenticator database containing shared secret
```
1|Dropbox|rff14xngz3bzhe5g7fhji4rzra|0|0|0||Dropbox
```

This file was then copied to a second device to test if cloning of the app is possible. In our experiment – even with totally different hardware – the Google Authenticator was able to generate identical OTPs then the legit app does (see Figure 1). This shows, that the app is not using hardware identifiers or any other kind of unique information to protect against this attack.

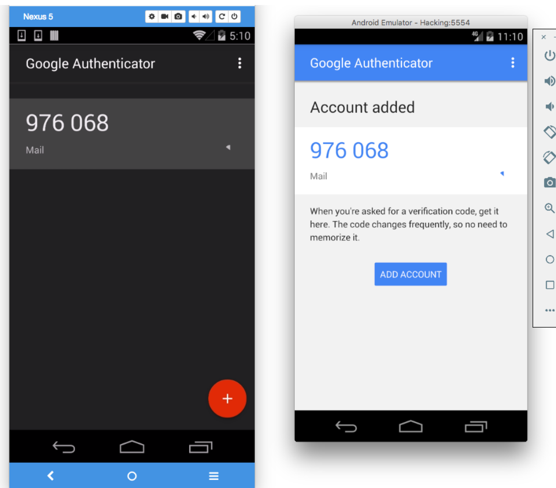

Figure 1. Cloned Google Authenticator app on a real mobile device (left) and an emulator (right)

During our experiments we are also looking at possible features like PIN protection of the app to protect against not authorized users and also to check the integrity of the device. Both features were missing in Google Authenticator.

When it comes to network traffic and a possible second attack vector, Google Authenticator was silent and doesn't transmit any data over the network. Neither during installation nor later while using the app.

B. *Microsoft Authenticator*

The second app that we analyzed is Microsoft Authenticator in version 4.4.0. In contrast to the testing of the Google Authenticator 2FA app, the Microsoft Authenticator was not configured for Dropbox, but as a second factor for the access to the email service Hotmail. This was necessary because the Microsoft app is currently not supported by Dropbox. The configuration was also more extensive. The app first requested the input of the existing credentials (user name and password) and then requested an alternative email address. Microsoft Authenticator sent a PIN to the secondary email. Only after entering this PIN within the app it started to generate OTPs. Afterwards, we investigated where and how the Microsoft Authenticator app stores the shared secret.

After doing the forensic analyzes we could identify the shared secret within the folder `/data/data/com.azure.authenticator/databases/`. The string `LklUuSlf7Pplqg==` was identical to the shared secret clearly displayed by Microsoft for manual input. This showed that the Microsoft Authenticator stores the shared secret in an unencrypted SQLite database called `PhoneFactor` (see Listing 2).

Listing 2. Microsoft Authenticator database containing shared secret

```
1|00000000000000000000000000000000|Microsoft|ppolleit@********.de||1|LklUuSlf7Pplqg==
|f2f3bc9ff98229a3
```

The evaluation of the encrypted transmitted network traffic was informative, using the tool mitmproxy. It not only contained the transmitted password for the email account, but also the PIN for validation. In addition, the app sent the names of other email addresses used on the smartphone to Microsoft.

To test the OTP push functionality of the app, Microsoft (https://www.login.live.com) logged in with the usual credentials (user name and password of a test account) and entered the smartphone (with the concrete app) as a second factor. After logging off and on again, the server sent an authentication request to the app. At the same time, a pop-up window with a SessionID appeared on the login page in the browser and on the smartphone. This was called: 9R9LS. The request has now been acknowledged on the smartphone with *Confirm*. This process was recorded with the mitmproxy. Subsequently, the network recording was searched for the value of the SessionID. It turned out that the app corresponded regularly with the server `login.live.com` and uploaded data there (by POST) (https://131.253.61.80/ListSessions.srf). In response to one of these periodic packages, the server finally sent the SessionID:

Listing 3. Microsoft Authenticator server response containing the SessionID

```
<?xml version="1.0" encoding="utf-8" ?>
.....
```

```
<S:Header></S:Header>
<S:Body xmlns:ps="http://schemas.microsoft.com/Passport/SoapServices/PPCRL">
<ps:ListSessionsResponse Success="true">
<ps:ServerInfo ServerTime="2017-10-18T08:20:19Z">
        BL2IDSLGN2A058 2017.10.13.11.54.05
</ps:ServerInfo>
<ps:Sessions>
        <ps:Session>
                <ps:PUID>00064000B77C724B</ps:PUID>
                <ps:SessionID>89230AA50481CA60</ps:SessionID>
                <ps:DisplaySessionID>9R9LS</ps:DisplaySessionID>
                <ps:State>Pending</ps:State>
                <ps:RequestTime>1508314799</ps:RequestTime>
                <ps:ExpirationTime>1508315099</ps:ExpirationTime>
                <ps:SessionType>Device</ps:SessionType>
                <ps:Browser></ps:Browser>
                <ps:OperatingSystem></ps:OperatingSystem>
                <ps:Country></ps:Country>
                <ps:FirstSign></ps:FirstSign>
                <ps:SecondSign></ps:SecondSign>
                <ps:ThirdSign></ps:ThirdSign>
        </ps:Session>
</ps:Sessions>
</ps:ListSessionsResponse>
</S:Body>
</S:Envelope>
```

Triggered by clicking on *Confirm* within the app, it then sent the following data to the server resource https://131.253.61.80/ApproveSession.srf:

Listing 4. Microsoft Authenticator message containing the OTP

```
<?xml version: "1.0" encoding="UTF-8"?>
<s:Envelopexmlns:s=http://www.w3.org/2003/05/soap-envelope
....
appid: {F501FD64-9070-46AB-993C-6F7B71D8D883}
da: <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
        Id="BinaryDAToken0"
        Type="http://www.w3.org/2001/04/xmlenc#Element">
        <EncryptionMethodAlgorithm="http://www.w3.org/2001/04/xmlenc#tripledescbc">
        </EncryptionMethod>
<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:KeyName>http://Passport.NET/STS</ds:KeyName>
</ds:KeyInfo>
<CipherData>
        <CipherValue>CX+f1eYc....yh/GtrY0HU......E104dK3oF
        .....
        </CipherValue>
</CipherData>
</EncryptedData>
<ps:ApproveSessionRequest>
.....
        <ps:UserPUID>00064000B77C724B</ps:UserPUID>
        <ps:SessionID>89230AA50481CA60</ps:SessionID>
        <ps:SessionState>Approve</ps:SessionState>
        <ps:ClockSkew>-1872</ps:ClockSkew>
</ps:ApproveSessionRequest>
</s:Body>
</s:Envelope>
```

As can be seen in the above excerpt (Listing 4), the app sent a message to the server. with a TripleDES encrypted `BinaryDAToken0`. This messages can be intercepted and redirected to another service that uses the same authentication or used by an attacker to simultaneously accept his session instead of the one of the legitimate user.

C. Authy 2-Factor Authentification

The next app within our experiment is Authy 2-Factor Authentification in version 23.0.5. The installation of Authy required the specification of a phone number. Afterwards, an SMS or a call for the transmission of a PIN could be sent to this number. The SMS contained, besides the PIN, the following URL: https://www.authy.com/register?pin=977126&cellphone=49-170-940-XXXX&device_app=authy

After completing this registration, Authy asked for a password to create encrypted online backups. Then the dropbox account could be added. It was then investigated where and how the 2FA app Authy stores the shared secret.

We found the shared secret within the folder `/data/data/com.authy.authy/shared_prefs/` in the file `com.authy.storage.tokens.authenticator.xml` (see Listing 5).

Listing 5. Authy 2-Factor Authentification xml-file containing the shared secret


```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<int name="key_version" value="64" />
<string name="com.authy.storage.tokens.authenticator.key">
        [{"accountType":"dropbox",
        "decryptedSecret":"WYC6536XVI6JCXJSYK7KRCW4E",
        "upload_state":"uploaded",
        "encryptedSecret":"taBJZbXkjgXIkN66Ljj8Fw0tW3xDhPNrT7iyV67k06A==",
        "originalName":"Dropbox:philipevalu@wegwerfemail.info",
        "timestamp":1507582523,
        "salt":"Feji1dC33bNEjovTd2VZFuhss8ZjwSxZ",
        "digits":6,
        "name":"Dropbox:philipevalu@wegwerfemail.info",
        "id":"1507582647",
        "isNew":false,
        "hidden":false}]
</string>
</map>
```


Thus, it could be shown that Authy stores the shared secret both unencrypted and also encrypted in the XML file. The value listed under *decryptedSecret* was identical to the key displayed by Dropbox during setup.

Even though we are now able to read the shared secret in plain text, we were not able to clone the app on another device due to the fact that Authy is making heavy use of device dependent identifiers to protect against this kind of attack.

Network traffic recorded during installation showed that the app contacted an Amazon AWS instance (kinesis.us-east-1.amazonaws.com). For this purpose, there was a certificate `s3.amazonaws.com` in the program directory (after the first start of the app). After completion of the setup, this certificate was removed from the app. Furthermore, the SSL-encrypted network traffic showed that the content was encoded again and therefore not plain text. The Amazon Kinesis streams offer the so-called *Server-Side Encryption* [**?**].

D. DUO Mobile

During the installation of the DUO Mobile application in version 3.16.1, it carried out a comprehensive security analysis of the smartphone and presented its results in a generally understandable way as shown in Figure 2.

The value named under `otpSecret` was identical to the key shown in the Dropbox for manual input without encryption (see Listing 6). This showed that DUO Mobile saves the shared secret unencrypted in the json-file `accounts.json` which is located under `/data/data/com.duosecurity.duomobile/files/duokit/`.

Listing 6. DUO Mobile json-file containing the shared secret


```
{
        "name": "philipevalu@wegwerfemail.info",
        "otpGenerator": {
        "otpSecret": "HVWB64JEXHST5XG2RG5J5NFWCI"
},{
        "logoUri":
        "android.resource://com.duosecurity.duomobile/drawable/ic_dropbox"
}
```


It was possible to copy the database to another of our test devices. The app in the second device generated identical OTP.

The following setup was selected to evaluate the security of the implemented OTP push function: As before, the app was set up on the smartphone and connected to the evaluation computer via proxy configuration.

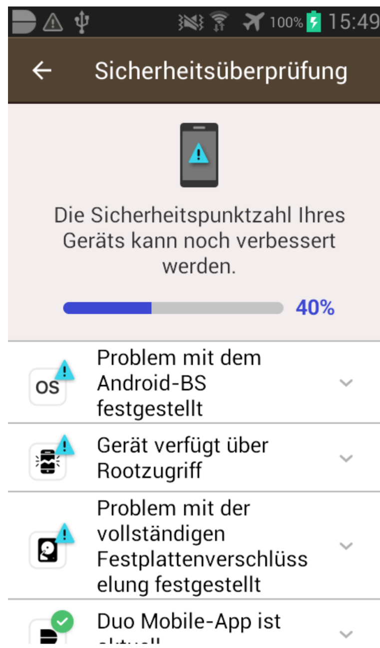

Figure 2. Device Integrity Check of DUO Mobile

DUO provides its customers with OTP push as a second factor when logging on to their homepage. This logon process was initiated and caused an OTP to be pushed to the smartphone. Unfortunately, we were not able to intercept the OTP push traffic.

E. FreeOTP

First of all, we investigated where and how the App FreeOTP – in version 1.5 – stores the shared secret. This 2FA app saved the shared secret unencrypted in the file `tokens.xml`. However, the notation was stored as a byte value (see Listing 7).

Listing 7. FreeOTP xml-file containing the shared secret


```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="tokenOrder">
        ["Dropbox:philipevalu@wegwerfemail.info"]
</string>
<string name="Dropbox:philipevalu@wegwerfemail.info">
        {"algo":"SHA1",
        "type":"TOTP",
        "secret":[-29,114,7,-22,0,127,-97,50,83,-106,31,8,39,-67,-124,-111],
        "issuerExt":"Dropbox",
        "issuerInt":"Dropbox",
        "label":"philipevalu@wegwerfemail.info",
        "counter":0,
        "digits":6,
        "period":30}
</string>
</map>
```


The further analysis of the app showed that it does not produce any network traffic. It was also possible to copy the database between the original smartphone and the second test-device to generate identical OTPs in both apps.

F. Sophos Authenticator

Within this section we will show how Sophos Authenticator in version 3.1 stores and protects the shared secret that is needed for generating OTPs. Similar to Google Authenticator this 2FA app also stores the secret within the SQLite file `databases` in `/data/data/com.sophos.sophtoken/databases/`.

Listing 8. Sophos Authenticator database containing the shared secret

```
1|Dropbox:philipevalu@wegwerfemail.info|RKDPFG6YDNAUWJXTH4ESHQZIXU|0|0|30|6|0|0
```

The value specified in the database file – as seen in Listing 8 – was identical to the secret displayed by Dropbox for manual input without encryption. This showed that Sophos Authenticator can store the shared secret in an unencrypted form in the SQLite database, which is also unencrypted. Furthermore, the app on the second test device generated the same OTP as that of the smartphone after exchanging the database file. As with the FreeOTP app, the analysis of the Sophos Authenticator revealed that it does not exchange data with the Internet.

G. *Push Authenticator*

Push Authenticator in version 1.1 has been the next app that we analyzed. Similar to many other 2FA apps for Android, it stores the secret within the SQLite file `databases` in `/data/data/com.net.singular.authenticator/databases/`.

Listing 9. Push Authenticator database containing the shared secret

```
1|Dropbox:philipevalu@wegwerfemail.info|WRY5SJPSIJO3OGGFY55OAU2YLQ|0|0|0
```

The value specified in the database file – as seen in Listing 9 – was the plaintext version of the secret. Due to the fact that Push Authenticator does not protect the itself against manipulated devices or high privileged attackers, cloning of the app was easily possible.

H. *OTP Authenticator*

At first, we investigated where and how OTP Authenticator stores the shared secret in version 0.1.1. The unencrypted shared secret could be found within the file `secrets.dat` located in `/data/data/net.bierbaumer.otp_authenticator/files/`.

Listing 10. OTP Authenticator file containing the shared secret

```
[{"secret":"A7WGQX3JFOLDHSB6XXCL7B2OV4======","label":"Dropbox -
Dropbox:philipevalu@wegwerfemail.info"}]
```

As can be seen in Listing 10 the secret is again stored in plaintext, but this time the app could not be cloned as it seems that it is storing device dependent information within one of its configuration files in an obfuscated way.

I. *Yandex.Key*

The app Yandex.Key in version 2.6 stores the shared secret encrypted in the SQLite database `ru.yandex.key.db` as can be seen in Listing 11.

Listing 11. Yandex.Key database containing the shared secret in an encrypted manner

```
2|7068696c69706576616c754077656777657266656d61696c2e696e666f44726f70626f78|||
|1:2487687582b37ecc4d5f1d76abd7dddb6d02781fd5307806:S7CFR5Q2GQPS66GFC5JA5PDU6
RGUHT7QQYU4LYMPITKOCFKRLJXJFZZK4IGVWMKMVNYA||TOTP|Dropbox|philipevalu@wegwerf
email.info|Dropbox|1:948b55b3f5bf2fe214754d9acf975ba1d8c38248ce36f9af:SODXRKE
QTD7LKKGIGV4HZR4ANTFQL5GFI4TZHIHDPEW4TG6BM3FECEAHTCYYFDXO6GTAUGUW|otp_logo/2F
A_Dropbox.png|0|0|0|0|0|0|0|0|0|0
```

For encryption, Yandex.Key uses the public domain library NaCl and device dependent information. Thus, we were not able to clone the app to another device.

J. *Symantec VIP Access*

First of all, it was examined where and how the app Symantec VIP Access stores the shared secret. With this app, the shared secret could be located in encrypted form within the xml-file `com.verisign.mvip.main_preferences.xml`.

Listing 12. Symantec VIP Access xml-file containing the shared secret in an encrypted manner

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="credData">
        [{"data":"C7B52A9D699
        .....
        "AUTH:HOTP:SHA1:6:SIGN",
        "id":"214BFD6EE....918118B00",
        "sval":"73EB995D4C420C6587050FC3D555D7F5"}]
</string>
```

```
<long name="serverTimeOffset" value="-525" />
<string name="keydidhash">
        3E53A565B.....206E1</string>
</map>
```

When looking at the source code of the app, you can see that it is using device dependent information like manufacturer name, device name and serial number to protect against cloning the app. These information are sent to the backend server of Symantec and there they are used to authenticate the device. Thus been said, it was not possible to clone the app to one of out test devices.

Now the OTP-push functionality of the app was examined. For this purpose, Symantec offers the possibility of sending an OTP-push at https://vip.symantec.com/. In the usual test setup (the smartphone was connected to the Internet via a proxy) there was no notification on the smartphone. When looking at the traffic you can see that the app is contacting https://40.114.55.233/incident-vip/1 instead of the legitimate URL (see Listing 13 for more details). This behavior is known from many newer Symantec products as soon as they notice that the traffic is manipulated or intercepted.

Listing 13. Symantec VIP Access packet that shows the manipulated network connection

```
19/01/2018 07:51:50 || VIP Access for Android?
Version 4.1 || SYMC51224959
0||353211060022320||golden||JZO54K||GT-I8190||user||samsung||armeabiv7a||
samsung/goldenxx/golden:4.1.2/JZO54K/I8190XXAMA1:user/releasekeys||
4.1.2|| || Device detection check reported an issue || :
```

K. 2FA Token

At this point, we investigated where and how the App 2FA Token – in version 1.0 – stores the shared secret. This 2FA app saved the shared secret unencrypted in the file `tokens.xml`. However, the notation was stored as a byte value (see Listing 14).

Listing 14. 2FA Token xml-file containing the shared secret

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="tokenOrder">
        ["Dropbox:philipevalu@wegwerfemail.info"]
</string>
<string name="Dropbox:philipevalu@wegwerfemail.info">
        {"algo":"SHA1",
        "type":"TOTP",
        "secret":[101,29,-33,-8,29,30,-104,35,44,-104,104,14,-86,124,23,-102],
        "issuerExt":"Dropbox",
        "issuerInt":"Dropbox",
        "label":"philipevalu@wegwerfemail.info",
        "counter":0,
        "digits":6,
        "period":30}
</string>
</map>
```

The further analysis of the app showed that it does not produce any network traffic, but – as seen many times before – It was also possible to copy the xml-file between the original smartphone and the second test device to generate identical OTPs in both apps.

L. Launchkey

Launchkey in version 3.1.0 requires registration via SMS (only within the USA) or email. A mail with a link was sent to the email address provided for registration. After this URL was called, the phone was connected and the following unlink code was presented to decouple the phone from the account in case of loss.

It was first examined where and how the app Launchkey stores the shared secret, but the shared secret (or an encrypted equivalent) could not be found in any of the files created by the app. In addition, the app only generated OTP if the smartphone was connected to the Internet. At the same time, the app could be copied between the devices and afterwards displayed identical OTPs on both devices.

When looking at the encrypted network traffic, it seems that the app stores the secret not locally but on the backend servers and only transmits the generated OTPs. Locally, we could only find an identifier that is used to link the stored shared secret with the corresponding device.

This kind of solution helps to protect against attacks that we already showed as working scenarios where an attacker with physical or low level filesystem access is able to extract the shared secret. But at the same time, it opens the door for remote attacker that now just need to bruteforce the ids used for linking of shared secret and user device.

As a positive remark, this app was one of only two we tested that was using a PIN to protect the app itself against unauthorized users.

M. CyAuth Cylocklite

Even before the possibility of importing shared secrets (e.g., via a QR-code) the app asked for a Google account. The account stored in the smartphone was selected for this purpose. As a result, the shared secrets were automatically synchronized and stored within the Google Drive of the connected account.

Then it was investigated where and how the CyAuth Cylocklite authenticator app stores the shared secret. The forensic analysis showed that the app stores the shared secret in encrypted form in the SQLite database `cylocklite_db` located at `/data/data/com18deglab.cylocklite/databases/`.

Listing 15. CyAuth Cylocklite database containing the shared secret in an encrypted manner

```
1|Dropbox:philipevalu@wegwerfemail.info|34|LXNbng61aQUJ6mdZT67TTzRB2SEoS6/OzZ
h4IytNtYvBKpTRKaJnvmCQx8Ka2G2o|||||1
```

Due to the fact that device dependent information is used for encrypting the shared secret, a simple cloning of the app has not been possible.

N. Topicus KeyHub

We investigated where and how Topicus KeyHub stores the shared secret in version 8.0. The unencrypted shared secret could be found within the SQLite database file `ouder.db` located in `/data/data/nl.topicus.heimdall/databases/`.

Listing 16. Topicus KeyHub database containing the shared secret

```
1|||Dropbox|philipevalu@wegwerfemail.info|SC5TCJAXFUFIYFXARI7SM4CZEU||
```

As can be seen in Listing 16 the secret is again stored in plaintext and this time the app could again be cloned. On the second test device it was enough to import this file. Afterwards, both devices generated identical OTPs.

O. Latch

Latch in version 1.7.1 initially required registration via an email address. A link with a password was then sent to this address. After calling the URL the dropbox account could be added.

It was then investigated where and how Latch stores the shared secret: It was stored in the file `b.xml` locally within `/data/data/com.elevenpaths.android.latch/shared_prefs/`, but in an encrypted manner – as can be seen in Listing 17. The encryption is based on a hardcoded key that an experienced attacker or investigator can easily reverse engineer and gather the shared secret in plaintext.

Listing 17. Latch xml-file containing the shared secret in an encrypted manner

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="registration_id">
        APA91bEb.... qqS12tbS
</string>
<int name="appVersion" value="21" />
</map>
```

Furthermore, the app contacted the registration server during the initial installation (https://52.30.153.108/www/registerMobile). As can be seen in the network traffic, the password selected for registration with the service provider has not been further secured – besides the default SSL encryption.

P. Okta Verify

Okta Verify was the only that 2FA app that we have seen that was making use of the on Android preferred encryption method: using a local keystore [13]. During initialization, the app creates the file `ast5d82b549-e6a0-4ce6-9003-03e3 1143ca4f.keystore` and encrypts all sensitive data. Furthermore, it uses device dependent information to protect against the attack of cloning the app to another device.

Table II
OVERVIEW OF EXPERIMENT AND RESULTS
(X = YES, O = NO, - = UNWANTED BEHAVIOR, + = WANTED BEHAVIOR)

2FA App Name	Cloning Possible	Encrypted Secret	Device Integrity Check	PIN Protection	Secure SSL-Connection	Secure OTP-Push
Google Authenticator	X-	O-	O-	O-	N/A	N/A
Microsoft Authenticator	X-	O-	O-	O-	X+	O-
Authy 2-Factor Authentification	O+	O-	O-	X+	X+	N/A
DUO Mobile	X-	O-	X+	O-	X+	X+
FreeOTP	X-	O-	O-	O-	N/A	N/A
Sophos Authenticator	X-	O-	O-	O-	N/A	N/A
Push Authenticator	X-	O-	O-	O-	N/A	N/A
OTP Authenticator	O+	O-	O-	O-	N/A	N/A
Yandex.Key	O+	X+	O-	X+	N/A	N/A
Symantec VIP Access	O+	X+	O-	O-	X+	X+
2FA Token	X-	O-	O-	O-	N/A	N/A
Launchkey	X-	N/A	O-	X+	X+	N/A
CyAuth Cylocklite	X-	X+	O-	O-	X+	N/A
Topicus KeyHub	X-	O-	O-	O-	X+	N/A
Latch	O+	X+	O-	O-	O-	N/A
Okta Verify	O+	X+	O-	O-	X+	N/A

Q. *Results*

In an overall assessment, it turned out that many of the apps examined resembled each other. They are often more or less direct derivatives of the Google Authenticator. This was often already evident from the directory structure and the use of the databases-file to store the shared secrets. Apps varied in terms of encryption used. While many did not encode, or only by different notation forms (bytecode), other apps used external libraries to implement the encryption. Even if the encryption was based on hardcoded keys – as shown by Latch – most of the used encryption proved to be solid. The same applies for most of the transmitted data between 2FA app and backend systems. The only negative example was again the app Latch that sends the user credentials in plaintext within the SSL connection. Table II provides an overview of the survey results for all examined 2FA applications.

IV. CONCLUSION

First of all, the use of two-factor authentication is fundamentally important and desirable. If, in the more detailed analysis, there may have been some deficiencies in some apps, this additional layer of security at least will not worsen the level of overall security.

As has been shown, the apps differ greatly in terms of their security implementation. Few apps had a self-sufficient security architecture, while most others depended directly on the security of the operating system. This would not be necessary. Although Android basically provides a quite secure environment (sandboxing, POSIX, SELinux) for this purpose, devices with a low patch level can be rooted relatively easily and the security architecture can then simply be leveraged during a possible attack. As explained at the beginning of this study, it is the purpose of the second factor to add the component possession to the existent classical component knowledge (e.g. username and password). The theft of the elementary shared secret, which has been presented in a few examples, then undermines the factor possession. At this point (by duplicating the shared secret), the generator loses the attribute possession. The shared secret and the OTP generated from it are thereby degraded to further passwords (factor knowledge).

In addition, the usage of 2FA-apps carries the risk that the thoughtless user may also use it's smartphone for login onto the secure server (here the web browser) and at the same time for generating the necessary OTP. Such an approach would also remove the second factor (something like violating the OOB principle) and work towards any malware on the smartphone.

It could be shown that under certain circumstances (e. g. platform security / implementation of the app) 2FA apps can be a useful supplement to user authentication. Suitable applications include, for example, private email accounts, password safes or access to online stores. However, for the protection of mobile online banking or other security-critical applications, the use of 2FA apps (here referred to as TAN apps) should be verified very critically.

V. Acknowledgements

We would like to thank all of the anonymous reviewers and especially to the shepherd – Felix Freiling – who helped to make this paper even better.

References

[1] Bundeskriminalamt. (2017, July) Hacker-Sammlung gefunden: 500 Mio. E-Mail-Adressen und Passwörter betroffen. Website. [Online]. Available: https://www.bka.de/SharedDocs/Kurzmeldungen/DE/Kurzmeldungen/170705_HackerSammlung.html

[2] H. P. Institute. (2017, December) The Top10 of German Passwords. Website. [Online]. Available: https://hpi.de/pressemitteilungen/2017/die-top-ten-deutscher-passwoerter.html

[3] N. Jurran. (2011, April) Angriff auf Playstation Network: Persönliche Daten von Millionen Kunden gestohlen. Heise Website. [Online]. Available: https://www.heise.de/newsticker/meldung/Angriff-auf-Playstation-Network-Persoenliche-Daten-von-Millionen-Kunden-gestohlen-1233136.html

[4] Equifax. (2017, September) Equifax Releases Details on Cybersecurity Incident, Announces Personnel Changes. Website. [Online]. Available: https://www.equifaxsecurity2017.com/2017/09/15/equifax-releases-details-cybersecurity-incident-announces-personnel-changes/

[5] L. Franceschi-Bicchierai. (2016, May) Another Day, Another Hack: 117 Million LinkedIn Emails And Passwords. Motherboard Website. [Online]. Available: https://motherboard.vice.com/en_us/article/78kk4z/another-day-another-hack-117-million-linkedin-emails-and-password

[6] M. Varmazis. (2016, September) And the worst passwords from the last.fm hack are.... Sophos Website. [Online]. Available: https://nakedsecurity.sophos.com/2016/09/02/and-the-worst-passwords-from-the-last-fm-hack-are/

[7] Two Factor Auth (2FA). [Online]. Available: https://twofactorauth.org/

[8] R. van der Meulen, "Gartner Says Demand for 4G Smartphones in Emerging Markets Spurred Growth in Second Quarter of 2017," Gartner, Tech. Rep., August 2017.

[9] L. Lamport, "Password Authentication with Insecure Communication," in *Communications of the ACM*, vol. 24, no. 11, November 1981.

[10] N. Haller, C. Metz, P. Nesser, and M. Straw, "A One-Time Password System," IETF, Tech. Rep., 1998.

[11] D. M'Raihi, "TOTP: Time-Based One-Time Password Algorithm," IETF, Tech. Rep., 2011.

[12] H. Krawczyk, "HMAC: Keyed-Hashing for Message Authentication," IETF, Tech. Rep., 1997.

[13] Google. (2018, January) Best Practices for Security & Privacy - Android Keystore System. [Online]. Available: https://developer.android.com/training/articles/keystore.html