SAFEPASS
Presenting a Convenient, Portable and Secure Password Manager

Onur Hakbilen, Piraveen Perinparajan, Michael Eikeland and Nils Ulltveit-Moe
Faculty of Engineering and Science, University of Agder, Jon Lilletuns vei 9, 4879 Grimstad, Norway

Keywords: Security, Authentication, Password Manager, Portability.

Abstract: SAFEPASS is a password manager implemented as a self-contained application, developed with principles and ideas based on industry best practices and analysis of existing popular password managers. All password managers try to solve the same problem of avoiding bad passwords and poor user habits when managing passwords. Security measures are from a high-level perspective similar across competing products, however, each of them have some deficiencies, although typically not the same. SAFEPASS aims at being an all-around good password manager for all purposes that avoid these deficiencies. It is based on modern technologies from the JavaScript and .NET ecosystem including React, Xamarin, and ASP.NET Core. By using the Flux architecture, SAFEPASS gives the average user ability to tweak most of its core functionality while staying within recommended security margins. Advanced users are also given room for customization through more technical security options. SAFEPASS does, in particular, have a focus on security, portability, convenience and good design.

1 INTRODUCTION

With a fast evolving technology, more services are becoming available on the Internet. As an Internet user, you need to be able to identify yourself online for a vast amount of services. Security credentials, such as usernames and passwords for these services, need to be guarded by a mechanism, a "security guard", to ensure their privacy. The research question that this paper aims at solving, is *the lack of good, portable password managers that work well across different platforms (McCarney et al., 2012).*

Password managers can be divided into three groups (Gasti and Rasmussen, 2012): Password managers that can be used with no precaution (Class I), password managers that do not guarantee privacy (Class II) and password managers that provide neither security, integrity or authenticity (Class III). According to the authors, the built-in password manager in Chrome is an example of a class III password manager. They point out that these kind of password managers *can* be used, but it is up to the user to build layers of security around it. SAFEPASS described here is intended as a Class I password manager.

Most services and organisations enforce specific rules for creating and managing passwords. However, such rules tend to restrict peoples degree of freedom when creating new passwords, which in turn makes it harder for the user to create and remember a good password (Inglesant and Sasse, 2010). Different services have different rules for password generation and maintenance, which makes it even harder for a user to remember all these passwords. Often they then end up writing their password on a paper, which defeats the purpose of a password (Inglesant and Sasse, 2010). Password restrictions do not guarantee increased security by themselves, but do however increase the probability of a hacker guessing the password in a brute-force attack (Atwood, 2017). The reason for this is that users cannot cope with the number and complexity of passwords and resort to insecure workarounds as a consequence (Mazurek et al., 2013): *"If only security managers understood the true cost for users and the organization, they would set the policies differently."* Better holistic methods for password maintenance and generation are in other words needed.

The ideal option is to give the user freedom to choose a password without forcing them to follow a set of rules. Unfortunately, creating and remembering a strong password requires more mental effort by the user, especially when there are many different services available. Users furthermore tend to use the easiest way by reusing passwords or using too simple

Hakbilen, O., Perinparajan, P., Eikeland, M. and Ulltveit-Moe, N.
SAFEPASS - Presenting a Convenient, Portable and Secure Password Manager.
DOI: 10.5220/0006603102920303
In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, pages 292-303
ISBN: 978-989-758-282-0

passwords (Mazurek et al., 2013).

1.1 Bad Password Habits

LastPass, a password manager, performed an analysis of the Gmail leak that happened in 2014 (LastPass, 2017c). The analysis revealed several weaknesses, for example that people's passwords were short, weak and were reused across accounts. Research conducted by RoboForm also gives cause for concern regarding password habits (Roboform, 2017a)(Roboform, 2017b):

- Most people either write down their passwords in clear-text or use phrases containing numbers or letters they remember;
- 23% of the surveyed reported that they always use the same password, while 59% reported that they use 0-5 different passwords;
- 74% log in to 6 or more different websites every day;
- Only 8% use a password manager.

1.2 Password Managers

A way to solve the password administration problem above is to use a password manager that can store passwords securely. At first glance this may seem counter-intuitive considering a password manager creates a single point of failure by keeping all the eggs in the same basket. However, this "basket", if firmly secured, can be considered a better option than reusing and creating weak passwords. As we see it, a password manager ideally offers four important functions:

- **Securely Store user Credentials:** To securely store user credentials (username, passwords and service identification), the password container must itself be secure. Most password managers encrypt their container with a master password, a single, strong password - like a key to a chest, with the idea that the user needs to remember only a single strong password.
- **Generate Strong Passwords:** A user should ideally have unique passwords for all of one's services to reduce the impact of compromised passwords. The implied effect is that if one of the services is compromised, none of the user's other services are. A good password generator will ensure this.
- **Assist the user in Entering Passwords:** A password manager can for example support copy/paste of username/passwords or automatically fill login, registration and other forms on a webpage.
- **Synchronize Across Devices:** Users typically have passwords on many different devices, and want a consistent, synchronized view of the passwords across these devices.

Table 1 shows an overview of different features that LastPass, Dashlane, 1Password and KeePass offer. There are two features that many password managers do not offer: changing encryption algorithm and customizing the key derivation function (KDF). Having support for changing these functions is useful in case future vulnerabilities are identified for these cryptographic functions. Cloud-based password managers like LastPass, Dashlane and 1Password focus on usability while KeePass, which is offline-based focuses on customization including plugins.

Table 1: Overview of popular password managers (*requires plugins).

	LastPass	Dashlane	1Password	KeePass
Customizable encryption algorithm	✗	✗	✗	✓
Customizable KDF	✗	✗	✗	✓
Multi-platform	✓	✓	✓	✓
Open source	✗	✗	✗	✓
Multi-factor authentication	✓	✓	✓	✓*
Password metrics	✓	✓	✓	✓
Breach alert	✓	✓	✓	✗
No data leaks	✗[1]	✓	✓	✓
External third-party storage	✗	✗	✗	✓
Password sharing	✓	✓	✓	✗*
Password generator	✓	✓	✓	✓
Password changer	✓	✓	✗	✗
Privacy respecting	✓	✗	✓	✓
Beginner friendly	✓	✓	✓	✗
Subscription restricted features	✗	✗	✗	✓

2 RELATED WORKS

A systematic security analysis of five popular web-based password managers (LastPass, RoboForm, Mylogin, PasswordBox and NeedMyPassword) was performed in (Li et al., 2014). The researchers found critical vulnerabilities in all the password managers, and in four of them an attacker could steal arbitrary credentials from a user's account. The authors emphasize the need for using a defence-in-depth strategy to mitigate this problem. Our objective has been to follow such a strategy when developing SAFEPASS.

There also exists some former research on password manager prototypes. A cloud-based storage-free browser based password manager is described in (Zhao and Yue, 2014). This password manager was implemented in Firefox and does not appear to be a cross-platform solution to the extent that our solution

[1] https://blog.lastpass.com/2015/06/lastpass-security-notice.html/

is. This approach has a heavy server-based solution that requires high availability. It uses secret sharing techniques to store shares at different cloud locations. Our solution uses a lightweight server and can work in offline-mode. We use secret sharing for memory protection, and rely on the cryptographic strength of the master password encryption algorithm and key derivation function for the protection of the master password. Both use Password-Based Key Derivation Function 2 (PBKDF2) as key derivation function.

A web-based password manager supporting biometric authentication is described in (Yang et al., 2014). SAFEPASS has broader platform support and uses a more modern client design than this tool. Support for biometric authentication can if necessary be added to SAFEPASS in the future by using the Xamarin fingerprint API for Android 6.0[2].

Other cloud-based password systems are for example the Password Multiplier and Passpet (Halderman et al., 2005)(Yee and Sitaker, 2006), which essentially are password generators instead of password managers.

3 ADVERSARY MODEL

The adversary model assumed in this paper, is an adversary that may attempt to intercept the communication between the password manager and the password storage. The paper also considers risks if the adversary has compromised the client on which the password manager is running as well as risks if the server side is compromised. The following sections elaborate on the adversary model.

3.1 Constraints

This subsection describes constraints to the adversary model that the system cannot completely mitigate.

Memory Protection: Passwords will at some point have to exist as clear-text in the computer memory for the user to make use of it. This is a weakness that an attacker with access to the computer can exploit to retrieve the password either from memory or, in some cases, even directly from the screen. Although there are no perfect solutions to this problem, there are some preemptive measures that can be, and are made in order to limit potential attacks:

- Keep only the requested password in clear-text and leave the rest encrypted
- Remove the password from memory as soon as it is no longer needed.

Advanced Attacks: It is impossible to be completely secure against an adversary, especially if the attacker has managed to compromise the machine running the password manager, e.g. using Zero-day exploits. SAFEPASS mitigates this threat by using techniques such as code obfuscation and cryptographic key sharing to delay and reduce the risk of an attacker identifying the password. However an experienced and determined attacker may still be able to reverse engineer the implementation to attack the solution.

3.2 Assumptions

- **Equipment:** We expect the users to keep their software updated in order to be able to have a fully functional experience of our service. Operating systems and web browsers in particular.
- **Protocols:** We assume that the underlying cryptographic protocols (AES) are secure and work as intended.

4 DESIGN

The JavaScript ecosystem saves us from the burden of developing code for each platform (mobile, web, desktop etc.), enabling consistent and predictable behavior across all targeted platforms with modern technologies that are always up-to-date. When the project started we set a few functional requirements on how information is presented, how the user can manage records and other user stories based on an agile approach (Agile, 2017). We also had non-functional requirements in mind when developing the prototype:

- **Usability:** A well designed user interface ensures that the user can intuitively and effectively benefit from the service without losing focus on the important aspects. Important features should be easily accessible in a convenient way.
- **Confidentiality:** A password manager must maintain data confidentiality as it contains passwords for other services. All communications and storage should be secured using well-tested cryptographic algorithms.
- **Integrity:** The password manager must also maintain data integrity to avoid the risk that an adversary can modify data in the password manager.
- **Availability:** The service must be available on multiple platforms, but most importantly, it must

[2]Xamarin fingerprint API:https://developer.xamarin.com/guides/android/platform_features/fingerprint-authentication

be accessible without requiring any internet connection.

With SAFEPASS we focused on creating a self-contained application that can be hosted semi-independently from the backend. With a self-contained application, we are referring to a standalone web application in contrast to a multi-page website that is completely controlled by the server. The backend is not aware about what is being sent to it, which is an intentional design choice on our part. The idea being that no one, not even the people hosting the SAFEPASS server instance, should have access to the sensitive information. This design principle helps making SAFEPASS trustworthy.

A self-contained application enables us as developers to swap out the backend API without affecting the frontend. The web client application maintains its own state (through the Flux architecture), allowing for a simple, stateless server thus allowing it to offer offline-mode. SAFEPASS is designed as shown in Figure 1 where each client is designed to handle the synchronization of the password file.

Flux is the application architecture that Facebook uses for building scalable client-side web-applications[3]. Flux applications have four major parts: the dispatcher, the stores, actions and the views which can be React components. Figure 2 shows the low-level client architecture based on the Flux architecture which makes the data proceed in the same direction (often referred to as one-directional, or unidirectional architecture). Instead of binding each view-component to a particular model, the Flux architecture allows all components to share the whole state, like a database which supports React's declarative programming style. This pattern, in contrast to a traditional Model View Controller (MVC) based architecture, works great with single-page applications where the application might contain thousands of components. The Flux pattern makes the application scalable, organized, maintainable and prevents complex data loops in comparison to MVC (Gackenheimer, 2015).

The Flux architecture consists of 4 modules by default (fluxxor, 2017), however SAFEPASS implements this architecture using the Redux library as Javascript state container which in addition includes middlewares and reducers (except action-creators):

- **View:** The view is the graphical user interface which presents the data.
- **Actions:** An action is an object which gets dispatched by the view. The action tells the application about the event that occurred and may include some data. We have separated important tasks such as encryption to its own module called **service**. Services are essentially just plain JavaScript code that can be used outside actions.
- **Middleware:** Middlewares have the ability to stop, forward or manipulate an action. Redux comes with support for adding custom middlewares.
- **Reducer:** A reducer (Redux module) manages the application state by using the data provided by an action. The reducer is not mutating the state, but instead returns a new state allowing state rollback. Multiple reducers can be combined to represent the whole application state.
- **Store:** The store gives the view access to the state and the dispatcher which is used to dispatch an action.

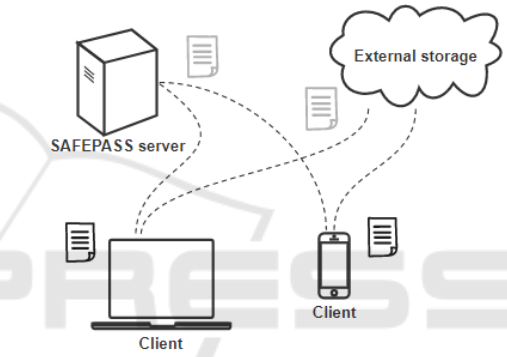

Figure 1: High level overview of the architecture.

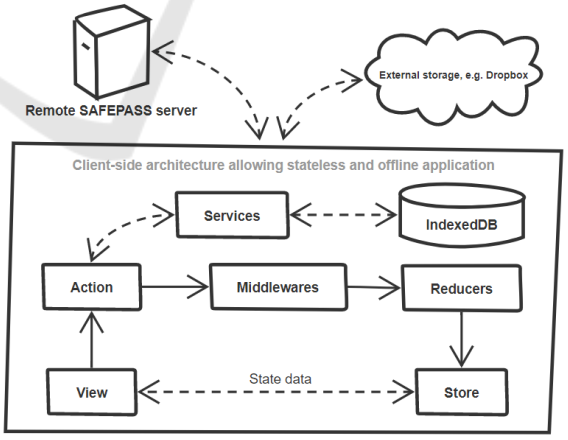

Figure 2: Low level client architecture with Flux.

4.1 Key Derivation

The authentication models for most publicly available password managers are very similar. A common approach is to use a key derivation function (KDF) to generate keys based on the master password. A key

[3]Flux: https://facebook.github.io/flux/docs/overview.html

derivation function is designed to perform key stretching based on a password to get a high-entropy key of the correct length as well as to be computationally intensive to make bruteforce attacks infeasible. Legitimate users will only need to derive a key once during authentication, while an attacker running a bruteforce attack will be restricted by the computational complexity of the KDF.

A well known KDF is the Password-Based Key Derivation Function 2 (PBKDF2) (Yao and Yin, 2005). It is used by LastPass[4], 1Password[5] and Dashlane[6]. However, with the increasing amount of computational power available, PBKDF2 seems to lose popularity as it is only CPU intensive. KeePass on the other hand uses Argon2 which was announced as the winner of "Password Hashing Competition (PHC)" in 2015 (Biryukov et al., 2016). Argon2 is not only CPU intensive, but also memory and thread intensive (Biryukov et al., 2016). SAFEPASS uses PBKDF2 by default but can be customized with other supported functions. This puts SAFEPASS ahead of its cloud based competitors.

Generating different keys for encryption E_k and authentication A_k prevents the other from being compromised if one of them was to be compromised. There are three preferred methods to generate unique keys based on the master password:

1. **Use Different Key Derivation Parameters**
2. **Use Different Salts**
3. **Use an Intermediate Key**

LastPass for example, uses option 1 with PBKDF2 for authentication and encryption and does that by using different iteration counts for each of the keys (LastPass, 2017a). There are no security disadvantages between solution 1, 2 and 3 as long as they are used correctly.

In the scenario of a locally stored encryption key, the last option would be the most optimal method. With solution 1 and 2, the legitimate user needs to compute the KDF twice with n iterations for each key (in total $2n$ iterations), whereas the attacker only needs to compute one of the keys to brute-force the original password. Instead, the user can compute an intermediate key, $k_n = KDF(password, salt, n)$, then the authentication key with $KDF(k_n, salt, 1)$ and encryption key with $KDF(k_n, salt, 2)$, resulting in $n+3$ iterations and therefore reducing the computational workload for key derivation.

SAFEPASS uses option 2 and always re-computes both keys and never stores them upon authentication. The user is therefore required to enter the password every time the application needs to decrypt the file. LastPass, 1Password and Dashlane on the other hand do not always require the user to type the password. If SAFEPASS was to follow the same path, method 3 could be used to efficiently calculate both keys. However, it allows the user to "remember" the password for a limited time. The method used to remember the encryption key is securely stored in different parts of memory using Shamir's Secret Sharing Scheme (Shamir, 1979). Code obfuscation is additionally used during code generation to make reverse engineering harder.

4.2 Authentication

Two popular authentication methods available today are token based authentication using the OAuth2 protocol (Hardt, 2011) and sessions based authentication with cookies. In token based authentication the client manages the token and can decide where to store it.

RFC7523 JSON Web Token (JWT) is a profile for OAuth 2.0 client authentication and authorization grants. It is a compact URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed using JSON Web Signature (JWS). The protocol uses basic HTTP authentication for authenticating the user, and issues a JWT token that can be used in subsequent requests to access resources on the server.

Session based authentication on the other hand is managed by the server. In contrast to token based authentication, the server keeps track of the client state which results in more memory usage on the server (affects scalability). Token based authentication is therefore popular for API-servers which do not require any state-tracking. Because the SAFEPASS client is a self-contained application that manages its own internal state, and because both have more or less the same security risks, using a token based authentication is more beneficial in our case. SAFEPASS authenticates the client as follows:

- **Request Login:** The client must prove its identity. First, the client requests login by providing user email. The server responds with a KDF scheme and authentication salt used in calculating the KDF (The KDF can be customized by the user through advanced settings).
- **Generate Keys:** The client calculates $KDF(password, salt, KDF scheme)$. The key is then sent to the server.

[4]https://helpdesk.lastpass.com/account-settings/general/password-iterations-pbkdf2

[5]https://support.1password.com/pbkdf2

[6]https://blog.dashlane.com/dashlane-explains-military-grade-encryption

- **Result:** If the provided key equals the server stored key, the client receives an access and refresh token (as described in OAuth2). The client can then use the access token in subsequent authenticated requests.

4.3 Data Confidentiality, Integrity and Availability

Authentication is an important part of the security, but keeping the password file safe on the client and in transit is equally important. Because most password managers are not open-source it is hard to analyze their actual implementation. However the product documentation indicates that they do follow security best practices in terms of cryptographic algorithms.

By default, SAFEPASS uses AES-CBC with a 256-bit encryption key. Data is contained in a JSON file that has a header and a payload. The header field contains the metadata about the KDF and cipher used for encryption. This allows the application to adjust to any specified cipher algorithm and KDF supported by the system libraries.

All sensitive data is encrypted by default. However, the payload is structured to have a private and a public field, where the public field may contain partially encrypted records. Because the public field is vulnerable to unauthorized alterations, the client must verify data integrity. This is done by using hash based messaging authentication code (HMAC) to sign the file. The master password, because it is something only the user knows, can be used to create a signature hash that is included in the file header. However, the signature can be used by an attacker to brute-force guess the original password. This is countered by using the KDF in addition as follows: $S_k = HMAC(KDF(password, KDF scheme), file)$.

To make the data available across devices, SAFEPASS has a robust and secure synchronization mechanism. In contrast to other cloud-based password managers, SAFEPASS allows the user to synchronize to an external cloud storage such as Dropbox as shown in figure 3. The user is able to toggle the synchronization and the application ensures that every conflict that might occur between the local and remote file is resolved automatically or by help of user actions as shown in figure 4.

5 SOLUTION

We believe we have created a secure, usable and convenient password manager by combining a clever design paired with modern technologies. When compared to other password managers, SAFEPASS is both a cloud based password manager like LastPass, and an offline password manager like KeePass.

Figure 3: SAFEPASS can synchronize to Dropbox.

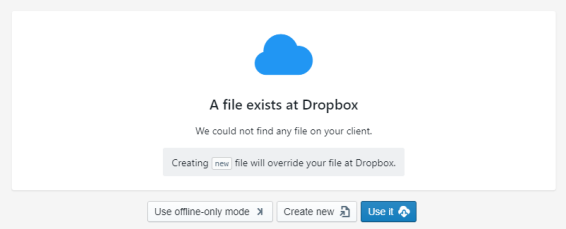

Figure 4: SAFEPASS always notifies user about synchronization issues.

SAFEPASS can automatically synchronize between devices and work offline without requiring internet connection. For this to work, we are using service workers which is a newly introduced feature to the client-side web programming (Gaunt, 2017). SAFEPASS is taking advantage of popular frameworks and technologies (ASP.NET Core, React, Redux, IdentityServer4, Electron etc.) to avoid reinventing the wheel.

5.1 Technologies

As previously discussed, by using technologies from the same ecosystem we can share the same codebase between platforms. SAFEPASS uses the React framework [7] to create the user interface (UI) and Redux [8] for data handling on the frontend. React is a component-based Javascript user interface library that supports a declarative programming style[9]. It is used by popular web applications such as Facebook and Instagram, and has a large set of community generated components that implement high-level functionality such as credit card forms,

[7]React: https://facebook.github.io/react
[8]Redux: http://redux.js.org/
[9]React https://reactjs.org

editors, touch screen handling etc. Redux on the other hand is a JavaScript state container which can be integrated to any JavaScript application to handle data management [10]. Furthermore, it uses Electron[11] to run the web application as a desktop application. The user interface is designed using the Blueprint UI toolkit which gives the application a desktop feel (Palantir, 2017). We have added our own styling and responsive elements according to our needs with custom CSS files.

Webpack[12] is used for building and bundling the client. It allows additional plugins such as code obfuscation, while also enabling us to use ECMAScript 6 which is a newer JavaScript standard compared to what is currently being used by most browsers (ECMAScript 5). We configured Webpack to bundle for both web and desktop application, as well as making it adapt the output to the environment. In this way, we can use the same project to build a desktop application with minimal effort.

The backend is built on the .NET Core framework including ASP.NET Core and IdentityServer4. We used MongoDB[13] as the primary database in our prototype, but switched to Entity Framework Core (EFC)[14] to support multiple SQL databases.

Both the main mobile application (SAFEPASS) and the application for two-factor authentication (SAFEPASS Guard) are developed using Xamarin[15]. Traditionally, developers had to know multiple languages to target every platform as shown in Table 2. Xamarin is a tool for cross-platform mobile development and provides a way to simultaneously make native mobile applications that work on Android, iOS and Windows devices. It uses the C# language which means that familiar concepts such as generics, LINQ and Task are available. It is open-source and part of the .NET ecosystem.

Table 2: Programming and view languages for each mobile-platform.

OS	Android	Windows Phone	iOS
Programming Language	Java	C# / Visual Basic	Objective-C / Swift
View	AXML	XAML	XiB / Storyboard

[10]Redux: http://redux.js.org
[11]Electron: https://electron.atom.io
[12]Webpack: https://github.com/webpack/webpack
[13]MongoDB: https://www.mongodb.com
[14]EFC: https://ef.readthedocs.io/en/latest
[15]Xamarin: https://www.xamarin.com

5.2 Backend

SAFEPASS has a minimal backend (as the heavy lifting is done at the client-side) based on the ASP.NET framework [16]. ASP.NET Core supports dependency injection out-of-the-box which makes the backend dependencies reusable, testable and readable as well as middlewares to control the HTTP pipeline. IdentityServer4 is an authentication middleware that provides OAuth2 grants using access tokens. We are using the "resource owner password flow" because the client is the resource owner (owns the password files) and can only prove identity with the password directly to the backend. As for now, we are not separating the IdentityServer4 service from the rest of the application, but in production we will be using a micro-service based architecture to organize the backend.

When authenticating, the client must issue access token through the authentication endpoint provided by the ASP.NET controller, and not directly to IdentityServer4. This makes it possible to offer two-factor authentication, track login history and verify the account status before issuing a token. Additionally, the user can register a two-factor device using the SAFEPASS Guard mobile application. This is done by generating a token based on the IMEI-number which is both stored locally on the device and the backend database. This token is used to generate a one-time password based on the current timestamp (TOTP (M'Raihi et al., 2012)) which is used in authentication with the server. Since this method is based on a secret token, we will be able to combine the user's fingerprint and device secret through technologies like touch ID (Rouse, 2014) to generate a token. This will prevent an attacker from logging into the account even if the IMEI or the device itself is compromised.

The backend is additionally responsible for registering the browser fingerprint. This is done using middlewares that read each HTTP request to find a specific header containing browser data. Even though the client is able to fake the information, it is a valuable service that gives the user ability to get an overview of all devices that has/had access to their account.

5.3 User Interface

In the React application, the client focuses on both intuitive design and organized data flow. There has

[16]ASP.NET Core:https://docs.microsoft.com/en-us/aspnet/core/

been focus on simple navigation, consistency, quality and usability. The user is able to add data such as credentials in form of ”records” as shown in figure 5. Each record is designed to be flexible and allows additional fields on the fly. Furthermore, the user is able to create categories to keep the records organized as illustrated in figure 6. Adding, modifying and deleting records will prompt the user to enter the password as shown in figure 7.

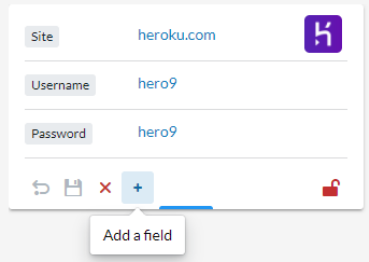

Figure 5: A record is represented as a ”card”.

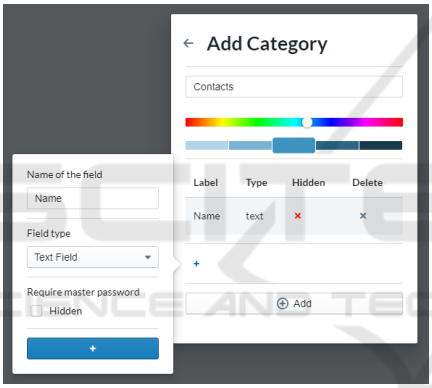

Figure 6: Creating a category.

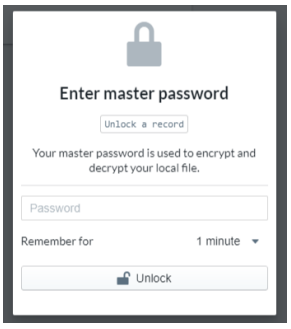

Figure 7: User is prompted to enter the master password.

5.4 Synchronization and Encryption

The synchronization mechanism consists of several parts: sync-, status- and merge service. The status service is used to check for any conflicts before the sync or merge service is called. Based on the result,

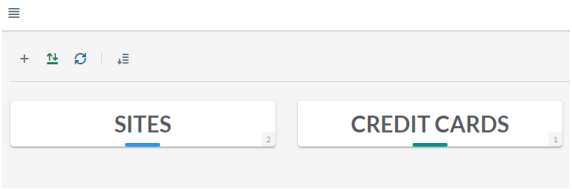

Figure 8: Overview of available custom categories.

the application shows a dialogue to tell the user about the conflict. For instance, when the application detects a corrupt file, the user is notified and can choose the next action. However, if there are no conflicts other than outdated files, the synchronization mechanism will automatically sync both locations through the sync- and merge service. The merge service compares the modified property of each record between the local and remote file to combine them.

In addition to the synchronization services, there are cryptography services that provide encryption, decryption, secret sharing, hashing and key derivation. These are combined with the synchronization mechanism to ensure that the sensitive data is always secured and available. The encrypted payload timestamp is used to determine if the password is needed to prevent prompting password when the encrypted content is not changed.

The cryptography service is abstracted as much as possible to support additional algorithms in the future with minimal effort. We are using third-party libraries in addition to the Web Crypto API[17] to support other algorithms and concepts like secret sharing.

5.5 Middlewares and Actions

Both actions and middlewares play important roles in the client-side application. There are actions for managing records, tweaking settings etc. Any of them can be dispatched by any view-component. When an action is dispatched it can not pause itself. This is an issue because some actions rely on the password which the user must enter. Moreover, there are actions which need to do authenticated requests and requires the access token to be valid. To solve this problem we created two middlewares:

- **The Password provider Middleware:** queues the action until the user enters the master password.
- **The Token Refresher Middleware:** queue the action and use the refresh token to issue a new access token from the server.

Both middlewares are designed similarly. They

[17]Web Crypto API: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API

will inspect an action to check if it has properties: requires and forward. The *requires* field is an array of strings that indicates if this action needs token refreshing, the password or both. The *forward* field contains a function that wraps the action's logic and is used to pass the action with the master password.

6 DISCUSSION

SAFEPASS is developed with the intent to resist the most common attack models on password managers:

- **Brute Force Attack:** Since the user is forced to choose a master password that is at least 8 characters, it is exponentially more time consuming for every extra character. SAFEPASS also uses a KDF to protect the master password.
- **Rainbow Table and Dictionary Attack:** Every password in SAFEPASS database is hashed *and* salted using a KDF. This greatly increases the ability to resist dictionary- and rainbow table-attacks (Oechslin, 2003).
- **Intercepting the File over the Network:** The fact that the password file is encrypted with strong crypto algorithms including a strong key makes it harder for an attacker to brute-force decrypt the file (assuming that the file is intercepted). However, SAFEPASS also prevents unnecessary transfer and always checks the signature of the file before it is downloaded.

The solution uses the JWT profile of OAuth for authentication, and the password manager connects to the authentication server via HTTPS to ensure the confidentiality of the request. It is important to run JWT over HTTPS with server certificate verification enabled, since the protocol otherwise uses a weak hash function (HMAC-SHA1) that is vulnerable to man-in-the-middle attacks (Kiani, 2017). The HTTPS JavaScript library ensures that the client only will connect to servers with a trusted server certificate by the operating system. Web browsers as well as Xamarin in Android have TLS/SSL server verification on by default, which mitigates the risk of man-in-the-middle attacks from untrusted certificates. However, even if an attacker was able to steal the web token, the attacker would only be able to download the password file encrypted with the master password. The security of the passwords would then hinge on the security of the chosen master password.

Even if an attacker managed to find the password, SAFEPASS offers a two-factor authentication scheme through the mobile application. This requires the user to be in possession of both the password and the registered mobile device.

Material Design: Many of Android's standard applications, such as Google Drive and Google Maps use Material Design[18]. We chose Material Design for two reasons. Firstly, it was introduced in Android 5.0 in 2012 (Xamarin, 2017), which means it has been around for quite some time. By choosing Material Design, we knew that the design of it was something users were familiar with. We think seeing as a familiar design might make it more attractive for users to install the product. Secondly, we wanted something minimalist; something that did not complicate the design, but at the same time was appealing. We landed on Material Design since it has a proven track record when it comes to usability. Google has set clear guidelines to ensure usability and accessibility.

Choice of Database: Any data in SAFEPASS that needs to be stored in a database is tied to the user model. This makes document-based databases favourable, as a single query will return all relations regarding the user. However, relational databases such as MySQL are popular choices when choosing a database system. SQL-databases have traditionally been the industry standard for decades, but have lately been challenged by NoSQL-databases such as MongoDB. There has been some controversy surrounding MongoDB where certain outdated versions, when configured improperly, would bind to all interfaces often leaving the database insecure (Wen et al., 2016). This generated a lot of negative publicity for MongoDB. However, the argument could be made that neglecting to properly update and configure a production database is the fault of the system administrator.

As mentioned earlier, we decided to instead have support for multiple database in the current version of SAFEPASS. For this, we are using Entity Framework Core to benefit from object-models and LINQ queries which makes it easier to work with the database in-code. On the other side, the Entity Framework allows us to change to any supported databases with minimal effort in case there is any issue with the current database. Additionally, this allows for a lightweight database like SQLite which does not require any setup on the development machine. We can also then transfer the database files between machines for testing purposes.

6.1 In Retrospect

Developing the mobile application in React Native may have saved us valuable time considering the code-base would be similar to that in web (React). React Native is cross-platform similar to Xamarin. However, testing for iOS-devices would still require

[18]Material Design: https://material.io/

an OS X machine which we did not have available. Furthermore, unlike Xamarin, React Native does not target Windows devices. This means that the current implementation with Xamarin has a broader potential platform support than with React Native.

With the project becoming larger and more complex, we noticed that it became harder to work with the frontend. Because JavaScript is a dynamically typed language there is no error-checking or any text-editor assistance while coding. This makes it both harder to debug the application as well as making the developer struggle to understand e.g. what a function requires. However, there are strict JavaScript supersets such as TypeScript which the current version of SAFEPASS uses. With TypeScript, we have to write more code, however it reduces false assumptions of variables and return types which saves us time on the long run. This is especially true for the WebCrypto interface that has a complex API.

6.2 Scalability

The most time-consuming function for interactive use of SAFEPASS is the key derivation function, which offers a configurable delay to reduce the risk of brute-force attacks on the password manager. The program runs mostly stand-alone on the client during normal operation. Other functions, for example server synchronization etc. are not time critical and cause little performance overhead. The encrypted JSON file is small, meaning that storage and computational requirements are negligible on modern mobile as well as desktop environments.

6.3 Compared with other Password Managers

SAFEPASS has the following advantages compared to the listed password managers in table 3:

- **Tweakable:** SAFEPASS allows customization of internal core functions including KDF and cipher algorithms.
- **Secure and usable**
  We have seen that most password managers follow best practices when it comes to security. However, they have distinct goals when it comes to features. KeePass for example gives the advanced user possibility to tweak most of its core functions (KeePass, 2017). Cloud based managers like LastPass and 1Password, on the other hand, allows limited customization of internal algorithms used for encryption and key derivation (1Password, 2017)(LastPass, 2017b). SAFEPASS tries to fill the gap between both types of password managers as shown in table 3, but at same time allows inexperienced users to use it.
- **Privacy Respecting Operation:** By using SAFEPASS you are not forced to synchronize your private data to the SAFEPASS server. Your data, or any information that can reveal sensitive information is never uploaded unprotected. SAFEPASS additionally offers offline-mode which means that you can work completely segregated from the server. While reading the Dashlane white paper (Dashlane, 2017a), we noticed privacy issues in their password changer feature: *"To change a password for a particular website, a Dashlane's client sends current saved password to Dashlane's servers along with a new strong password generated on the client."* (Dashlane, 2017b). Because it is theoretically accessible by anyone administrating the server, we have chosen to mark Dashlane as not a fully privacy respecting password manager (based on the credentials, they can find who you are).

Table 3: Overview of popular password managers including SAFEPASS for comparasion (*requires plugins, **future work).

	LastPass	Dashlane	1Password	KeePass	SAFEPASS
Changeable encryption algorithm	✗	✗	✗	✓	✓
Changeable KDF	✗	✗	✗	✓	✓
Multiplatform	✓	✓	✓	✓	✓
Open-sourced	✗	✗	✗	✓	✓
Multi-factor authentication	✓	✓	✓	✓*	✓
Password metrics	✓	✓	✓	✓	✓
Breach alert	✓	✓	✓	✗	✓**
No leaks	✗[19]	✓	✓	✓	✓
External storage e.g. Dropbox	✗	✗	✗	✓	✓
Sharing with others	✓	✓	✓	✗*	✓**
Password generator	✓	✓	✓	✓	✓
Password changer	✓	✓	✗	✗	✓**
Privacy respecting	✓	✗	✓	✓	✓
Beginner friendly	✓	✓	✓	✗	✓
Subscription restricted features	✗	✗	✗	✓	✓

7 CONCLUSION

SAFEPASS is a secure, usable and convenient password manager that combines a clever design paired with modern technologies. It is developed with the intent to resist the most common attack models for password managers. Even with the number of password managers already out there, our analysis suggest that existing password managers do not provide the desired level of usability as well as sufficient trustworthiness and security. A backbone based on the Javascript development environments allows the solution to offer usability and security at its core

[19] https://blog.lastpass.com/2015/06/lastpass-security-notice.html/

by reusing well-known design components. This is achieved using modern web-based technologies and a client-centric Flux architecture which makes SAFEPASS scalable. It offers a self-contained application that can be hosted semi-independently from the backend and that works as a stand-alone application, in contrast to more commonly used multi-page websites that are completely controlled by the server. SAFEPASS furthermore uses the PBKDF2 key derivation algorithm by default, however it can be swapped out with other algorithms to support upgrading of cryptographic algorithm. The average user can appreciate a clean, sleek and intuitive user interface with notable features such as customizable categories and records, temporarily remembering the master password, synchronization across platforms and devices, and two-factor authentication. More advanced users can take on more technical settings, such as customizing options in cryptographic algorithms and key derivation functions.

The chosen design base has allowed for rapid development of the password manager by extensive reuse of existing frameworks. The solution is already a working proof of concept with a professional look on par with commercial solutions in the area.

8 FUTURE WORK

Despite developing a fully functional password manager, there were some features that were either not fully implemented or not implemented at all due to limited time.

Password Recovery: Most services allow the user to create a new password by sending a link to a preapproved email. In the case of a password manager, the user's email service credentials might be safely stored in a password vault. Besides, the master password is the key needed to the decrypt the password vault. The server does intentionally, and for good reason, not have the master password in cleartext.

Administrative Tools: Since SAFEPASS is intended to be available as a self-hosted service, a few administrative tools would be useful for local setups. At the very least an administrator should be able to manage users. Other implementations, such as statistics and metrics, storage usage inspection and server-wide settings are possible features that could be implemented as administrative tools.

Browser Extension: Many password managers have support for browser extensions to inject credentials into forms on the website. Essentially, this is a feature which most users expects from a modern password manager. However, through our research we decided to not implement this browser extension as we saw how other password managers struggled to secure their extensions from attackers, LastPass being an example of this (Lawler, 2017).

Password Changer: A password changer is an emergency feature that should be consided implemented in the future. We could for example send instructions to the user's machine and let the client manage the password changing locally (possible with automation tools such as Selenium (Selenium, 2017)). To support many websites, we could for example enable contributors to create instruction schemes for different sites.

ACKNOWLEDGEMENTS

This research has been supported by the Centre for Integrated Emergency Management at University of Agder, Norway.

REFERENCES

1Password (accessed 2017). How pbkdf2 strengthens your master password - 1password support. https://support.1password.com/pbkdf2/.

Agile (accessed 2017). User stories: An agile introduction. http://www.agilemodeling.com/artifacts/userStory.htm.

Atwood, J. (accessed 2017). Password rules are bullshit. https://blog.codinghorror.com/password-rules-are-bullshit/.

Biryukov, A., Dinu, D., and Khovratovich, D. (2016). Argon2: New generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy*, pages 292–302. IEEE.

Dashlane (accessed 2017a). Dashlane explains: Military grade encryption - dashlane blog. https://blog.dashlane.com/dashlane-explains-military-grade-encryption/.

Dashlane (accessed 2017b). What is password changer — dashlane. https://csdashlane.zendesk.com/hc/en-us/articles/ 202699181-What-is-Password-Changer-and-how-does-it-work-.

fluxxor (accessed 2017). Fluxxor - what is flux? http://fluxxor.com/what-is-flux.html.

Gackenheimer, C. (2015). Introducing flux: An application architecture for react. In *Introduction to React*, pages 87–106. Springer.

Gasti, P. and Rasmussen, K. B. (2012). On the security of password manager database formats. In *ESORICS*, pages 770–787.

Gaunt, M. (accessed 2017). Service workers: an introduction — web — google developers.

https://developers.google.com/web/fundamentals/getting-started/primers/service-workers.

Halderman, J. A., Waters, B., and Felten, E. W. (2005). A convenient method for securely managing passwords. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 471–479, New York, NY, USA. ACM.

Hardt, D. (2011). Rfc 6749 - the oauth 2.0 authorization framework. https://tools.ietf.org/html/rfc6749. (accessed 2017).

Inglesant, P. G. and Sasse, M. A. (2010). The true cost of unusable password policies: Password use in the wild. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 383–392, New York, NY, USA. ACM.

KeePass (accessed 2017). Database settings - keepass. http://keepass.info/help/v2/dbsettings.html.

Kiani, K. (accessed 2017). Four attacks on oauth - how to secure your oauth implementation. https://www.sans.org/reading-room/whitepapers/application/ attacks-oauth-secure-oauth-implementation-33644.

LastPass (accessed 2017a). Lastpass technical security whitepaper. https://enterprise.lastpass.com/wp-content/uploads/ LastPass-Technical-Whitepaper-3.pdf.

LastPass (accessed 2017b). Password iterations (pbkdf2) — user manual. https://helpdesk.lastpass.com/account-settings/general/password-iterations-pbkdf2/.

LastPass (accessed 2017c). The scary truth about your passwords: An analysis of the gmail leak — the lastpass blog. https://blog.lastpass.com/2014/09/the-scary-truth-about-your-passwords-an-analysis-of-the-gmail-leak.html/.

Lawler, R. (accessed 2017). Critical security flaws found in lastpass on chrome, firefox (updated). https://www.engadget.com/2017/03/22/ critical-exploits-found-in-lastpass-on-chrome-firefox/.

Li, Z., He, W., Akhawe, D., and Song, D. (2014). The emperor's new password manager: Security analysis of web-based password managers. In *USENIX Security Symposium*, pages 465–479.

Mazurek, M. L., Komanduri, S., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., Kelley, P. G., Shay, R., and Ur, B. (2013). Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 173–186, New York, NY, USA. ACM.

McCarney, D., Barrera, D., Clark, J., Chiasson, S., and van Oorschot, P. C. (2012). Tapas: Design, implementation, and usability evaluation of a password manager. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 89–98, New York, NY, USA. ACM.

M'Raihi, D., Machani, A., Pei, M., and Rydell, J. (2012). Totp: Time-based one-time password algorithm. https://tools.ietf.org/html/rfc6238. (accessed 2017).

Oechslin, P. (2003). Making a faster cryptanalytic time-memory trade-off. In *Crypto*, volume 2729, pages 617–630. Springer.

Palantir (accessed 2017). Blueprint documentation. http://blueprintjs.com/docs/.

Roboform (accessed 2017a). Password security survey results- part 1. https://www.roboform.com/blog/password-security-survey-results.

Roboform (accessed 2017b). Password security survey results- part 2. https://www.roboform.com/blog/password-security-survey-results-part2.

Rouse, M. (2014). Apple touch id. (accessed 2017).

Selenium (accessed 2017). Selenium web browser automation. http://www.seleniumhq.org/.

Shamir, A. (1979). How to share a secret. *Communications of the ACM*, 22(11):612–613.

Wen, S., Xue, Y., Xu, J., Yang, H., Li, X., Song, W., and Si, G. (2016). Toward exploiting access control vulnerabilities within mongodb backend web applications. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 143–153. IEEE.

Xamarin (accessed 2017). Material theme - xamarin. https://developer.xamarin.com/guides/android/user_interface/material-theme/.

Yang, B., Chu, H., Li, G., Petrovic, S., and Busch, C. (2014). Cloud password manager using privacy-preserved biometrics. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 505–509. IEEE.

Yao, F. F. and Yin, Y. L. (2005). Design and analysis of password-based key derivation functions. *IEEE Transactions on Information Theory*, 51(9):3292–3297.

Yee, K.-P. and Sitaker, K. (2006). Passpet: convenient password management and phishing protection. In *Proceedings of the second symposium on Usable privacy and security*, pages 32–43. ACM.

Zhao, R. and Yue, C. (2014). Toward a secure and usable cloud-based password manager for web browsers. *Computers & Security*, 46:32 – 47.