CryptoChronicles

LessPM's Quest to a Passwordless Utopian Ecosystem

T-742-CSDA

Håvard Nordlie Mathisen
Reykjavik University
havard22@ru.is

Instructor
Jacky Mallet

Department of Computer Science
Reykjavik University
April 14, 2023

Contents

Abstract

We describe the implementation of LessPM, a passwordless password manager using Rust as a programming language. While traditional passwords have a long and respectable history, they have become increasingly vulnerable to attacks such as brute-forcing, phishing, and keylogging.

LessPM seeks to address these deficiencies with a sophisticated authentication method, called WebAuthn. Through asymmetric cryptography, WebAuthn provides a secure authentication experience with the help of user-owned devices such as smartphones or hardware authenticator devices. This would eliminate the use of traditional passwords.

To secure LessPM's passwords on the server, we employ AES-256 and Argon2 for encryption, WebAuthn's Credential ID, randomly generated salts, and pepper for key derivation.

We also use Hypertext Transfer Protocol Secure (HTTPS) to secure communication between client and server and configured Cross-Origin Request Sharing (CORS). Additionally, we used an encrypted version of JSON Web Tokens (JWT, RFC7519), inspired by JSON Web Encryption (JWE, RFC7516), to maintain user authentication between client requests.

We created a separate client as a visual proof-of-concept to adhere to The Law of Demeter,

Ask Jacky whether she is sure about the footnote here, given that it is an abstract.

while the server handles requests and securely stores user data. This approach offers promising prospects for a passwordless future in digital security.

This report provides valuable insights into the implementation and potential benefits of using WebAuthn and a passwordless approach, along with the security measures employed to protect user data on the server.

1 Introduction

In today's digital landscape, utilizing various identifiers (such as usernames, email addresses, or phone numbers) combined with passwords has become a prevalent method for verifying an individual's identity and ensuring their authorization to access restricted materials.

This is not a novel approach. Polybius' *The Histories* [Pol23] contains the first documented use of passwords, describing how the Romans employed "*watchwords*" to verify identities within the military. This provided a transparent, simple way to allow or deny entry to restricted areas of authorized personnel only. The story of secret writing (in this context referenced as cryptography) goes back the past 3000 years [Doo18], where the need to protect and preserve privacy between two or more individuals blossomed.

Fernando J. Corbató is widely credited as the father of the first computer password when he was responsible for the Compatible Time-Sharing System (CTSS) in 1961 at MIT [Lev84]. The system had a `"LOGIN"` command, which, when the user followed it by typing `"PASSWORD"`, had its printing mechanism turned off to offer the applicant privacy while typing the password [65]. Given the long history of passwords and their importance, one could argue that it was a natural and judicious step in the evolution of computer systems.

The study "The Memorability and Security of Passwords", conducted in 2004, provides insights into password creation strategies, including tips on improving password entropy and methods for easy recall of passwords [Yan+00]. With an emphasis on diversity in character selection, password length, and avoiding dictionary words, the study suggests that acronym-based passwords offer a delicate balance between memorability and security [Yan+00].

However, as technology has advanced, the limitations of password-based authentication have become increasingly apparent, leading to the development of more sophisticated methods like Universal Authentication Framework (UAF) [FID17] and We-

bAuthn [Wor21] through the Fast IDentity Online Alliance (FIDO) and The World Wide Web Consortium (W3C).

This report explores the implementation of LessPM, a passwordless password manager that leverages WebAuthn (Web Authentication), an open standard and collaborative development effort between FIDO and W3C to provide a secure authentication experience through biometric scanners on devices such as a smartphone or a hardware authenticator device to provide a secure experience. Free from the constraints of traditional passwords, while placing a strong emphasis on security.

LessPM is designed with a multi-layer security approach to ensure confidentiality and integrity of user authentication and the passwords belonging to other services that user's are registered on. By utilizing WebAuthn's asymmetric cryptographic [a]nature [W3C21b], LessPM authenticates users strongly through the help of authenticator devices such as a smartphone or hardware authentication device in possession of the user. When registering in LessPM, the user is prompted to use a biometric sensor on their device. The device constructs a keypair that the implemented WebAuthn server receives and stores the public key of the keypair before registering the user. This process can be seen in Figure 1.

When the user attempts to authenticate in the future, the user is again prompted to use the biometric sensors on their device. Upon initiating this process, the server issues a challenge which is signed by the private key on the authenticator device, yielding a signature back when a biometric sensor is successful in authenticating the authenticity of the user. The server can then verify this signature with the help of the public key stored int he previous step to authenticate the user. This process can be seen in Figure 2.

The aforementioned described processes constructs an environment such that a malefactor is required to first access the authenticator device and then to bypass the device's biometric sensors security in order to authenticate as the user.

Passwords are encrypted using AES-256, a widely recognized symmetric encryption algorithm [Sch00; DR02], with a Credential ID (CID) that is derived from the public key belonging to the keypair from WebAuthn, a 128-bit randomly generated salt, unique for each password, and a 128-bit pepper.

[a]Asymmetric cryptography uses a key-pair consisting of public and private keys. The public key encrypts data, while the private key decrypts it. The keys are mathematically related, but deriving one from the other is infeasible, ensuring secure communication and data exchange.

By examining recent advancements in authentication mechanisms and the related innovative potential of WebAuthn, we hope to illuminate the prospects of a passwordless future in digital security.

The findings in this report is not an attempt of getting rid of trivial concepts such as session hijacking, nor is it an empirical study of user's perception of the technology seen.

2 Background

In today's world of rapidly evolving technology, it is essential to have a strong foundation in the underlying concepts and protocols that drive modern systems. This background chapter aims to provide a comprehensive understanding of the key technologies and principles that are relevant to our report. By delving into these fundamental topics, we can better appreciate their significance and application in the context of the implementation, design, and architecture presented in the subsequent chapters.

We will provide background information on topics such as WebAuthn (Section 2.1), JSON Web Tokens (JWT. Section 2.4), JSON Web Encryption (JWE. Section 2.4), hashing through Argon2 (Section 2.3), and Advanced Encryption Standard (AES. Section 2.2).

2.1 WebAuthn

WebAuthn, short for Web Authentication, is a collaborative project between the FIDO Alliance and W3C that aims to implement a secure, robust key-based authentication system for the web, to strongly authenticate users [Wor21]. The concept relies on the use of a third-party device, called an Authenticator, which leverages asymmetric cryptography. These devices employ biometric or hardware-based mechanisms to provide secure and reliable means of authenticating a user.

Upon registration, the Authenticator Device (AD) generates a key-pair called a Passkey. This Passkey contains a CID uniquely generated for each registered key-pair [21a; 21b] on the Authenticator, per service registered. The unique generation of each key-pair offers the advantage of making it much more difficult for trackers to follow a user[1].

Further, if an attacker gains access to an individual's Passkey, they might compromise one specific service, whereas a traditional password could potentially compromise multiple services where password reuse occurs [Wan+18]. This suggests that WebAuthn could serve a stronger level of security, whereas a traditional password exposed in a leak will expose users who are reusing password across services and devices.

2.1.1 Registration

In order to register in a WebAuthn authentication system, the user (i.e., client, browser, phone, Etc.) issues a registration request to a WebAuthn implemented server, called a Relying Party (RP), asking to be registered [W3C21c]. The request contains a body with the relevant user identifier (UID. i.e., username, phone number, email, Etc.). The server responds by initiating a Registration Ceremony (RC) and generates a challenge, which is sent to the client [W3C21d].

The client then calls the browser-integrated WebAuthn API, requesting the Authenticator (i.e., phone, hardware authenticator device, Etc.) to create a new public key credential through the Client to Authenticator Protocol (CTAP2).[2] During this process, the Authenticator generates a new key-pair (public and private keys). The Authenticator then signs the challenge received from the server with the private key [W3C21b].

The newly created public key, signed challenge, and additional metadata are combined into a public key credential object, which the client sends back to the server.

[1] This is subject to the key-pair alone. A willing party could still track the user through their email or similar.

[2] The user is prompted to use their Authenticator to prove their presence, which can involve scanning a QR code, providing a fingerprint, or any other modality supported by the device.

The server verifies the authenticity of the signed challenge and the public key credential object. If successful, the server stores the user's public key and other relevant information (e.g., user identifier, credential ID) for future authentication.

Thus completing the RC, the process can be seen in Figure 1.

2.1.2 Authentication

When a user wishes to perform authentication (commonly referred to as `logging in`), much of the same procedure occurs. The user issues an authentication request to the RP, asking for authentication with the UID that the user used to sign up included in the request body. If the server can find an associated user with the UID in the persisted storage, the server responds by initiating and issuing an Authentication Ceremony (AC) containing a challenge.

The client calls the browser-integrated WebAuthn API, prompting the use of the Authenticator to validate and sign the challenge. Unlike the registration process, the Authenticator now yields a signature, which is forwarded to the RP along with Authenticator-specific data [W3C21a]. Upon receiving the data, the RP validates the signed challenge using the stored credentials. If the validation succeeds, the server authenticates the user.

The RP can then determine the next steps, such as deciding the authentication duration and establishing methods for persisting and validating the authentication.[3] If the authentication process is unsuccessful at any point in the ceremony, the ceremony is aborted and considered invalid.

This process can be seen in Figure 2.

[3] In the case of LessPM, the system sets an authentication duration of 15 minutes. It stores this information within an AES-256 encrypted 2.2 JSON Web Token. For more details about this process, refer to Section 2.4.

2.2 Advanced Encryption Standard with 256-bit

The Advanced Encryption Standard (AES) is a symmetric[4] key encryption algorithm. Since its inception in 1998, it has become the gold standard for encrypting various information across applications [Sch15; DR02], being adopted as the successor of the Data Encryption Standard (DES) by The National Institute of Standards and Technology (NIST) in 2001 [NIS00]. AES operates on fixed-sizes units of data referred to as `blocks` [ST01a], supporting keys of sizes 128-, 192-, and 256-bit [ST01b]. A Substitution-Permutation Network (SPN) structure forms the basis of the design, which achieves a high level of security through multiple rounds of processing by combining substitution and permutation [ST01c]. AES with 256-bit key length (hereafter referred to as AES-256 in the rest of the report), employs a 256-bit key and consists of 14 rounds of encryption, offering an advanced level of security compared to its counterparts with shorter key lengths and fewer rounds [ST01d]. In each round of encryption, AES-256 undergoes four primary transformations, operating on a 4×4 block, as seen in Figure 3.
The larger key-size in AES-256 provides an exponential increase in the number of possible keys, making it significantly more resilient to brute-force attacks and further solidifying its position as a robust encryption standard for safeguarding sensitive information.[6]

2.3 Hashing through Argon2

Argon2 is a hashing/key-derivation function developed by Alex Biryukov, Daniel Dinu, and Dmitry

[4] Symmetric in this context refers to the same key to encrypt and decrypt.

[5] The term `state` refers to an intermediate result that changes as the algorithm progress through its phases.

[6] The practical number of potential keys for an AES-256 implementation is 2^{256} possibilities. This gives us an approximation of 1.1579209×10^{77} options. The number is theoretical, as this is a worst-case scenario of options an attacker must go through to find the right key.

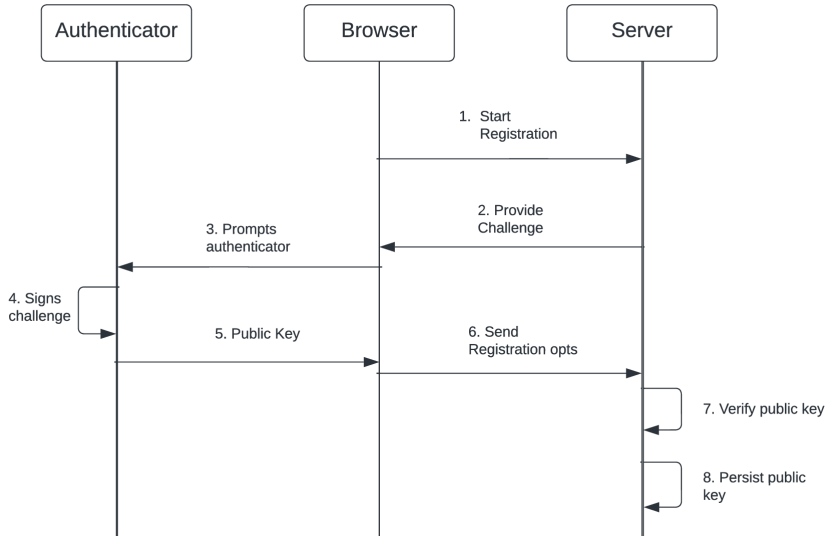

Figure 1: A diagram dipicting the registration process through WebAuthn and Authenticator Device.

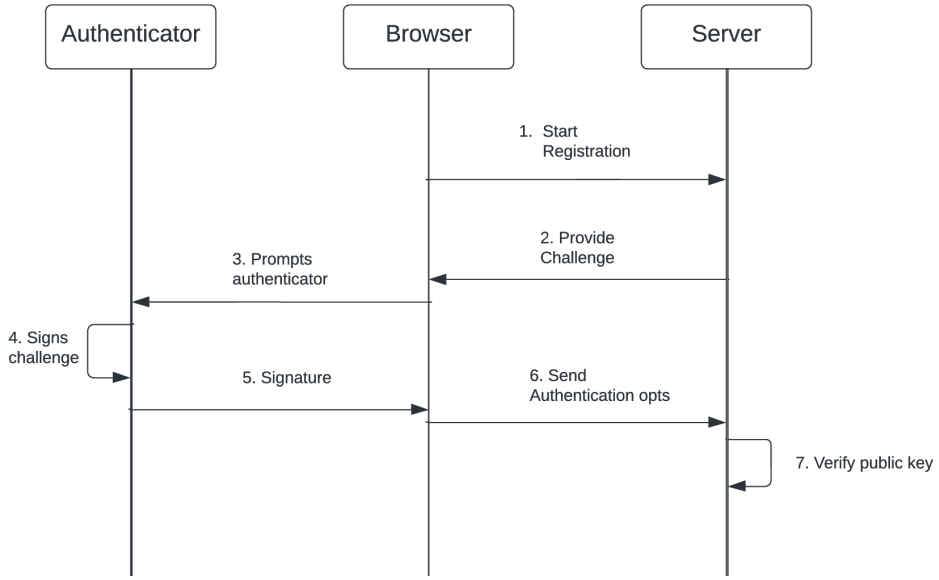

Figure 2: A diagram dipicting the authentication process through WebAuthn and Authenticator Device.

- **SubBytes** is a non-linear substitution step where each byte is replaced with another according to a predefined lookup table.
- **ShiftRows** cyclically shifts each row of the state over a certain number of steps. of the State over varying numbers of bytes while preserving their original values.
- **MixColumns** is a process that works on the columns of the state by combining the four bytes in each column through a mixing operation.
- **AddRoundedKey** involves combining a subkey with the state[5] by applying a bitwise XOR operation.

Figure 3: The steps AES takes when encrypting and decrypting data [ST01c].

Khovratovich [al17]. Argon2 aims to provide a highly customizable function tailored to the needs of distinct contexts [al17]. Additionally, the design offers resistance to both time-memory trade-off and side-channel attacks as a memory-hard function [al17].

The function fills large memory blocks with pseudorandom data derived from the input parameters, such as the password and salt.[7] The algorithm then processes the blocks non-linearly for a specified number of iterations [al17].

The function offers three configurations, depending on the environment where the function will run and what the risk and threat models are:

Argon2, as a memory-intensive hashing function, demands substantial computational resources from

- **Argon2d** is a faster configuration and uses data-depending memory access. This makes it suitable for cryptocurrencies and applications with little to no threat of side-channel timing attacks.[8]
- **Argon2i** uses data-independent memory access. This configuration is more suitable for password hashing and key-derivation functions.[9] The configuration is slower due to making more passes over the memory as the hashing progresses.
- **Argon2id** is a combination of the two, beginning with data-dependent memory access before transitioning to data-independent memory access after progressing halfway through the process.

Figure 4: The three configurations of Argon2 [al17].

attackers attempting dictionary attacks.[10] This characteristic significantly hampers the feasibility of cracking passwords using such attacks. The algorithm's customizability allows users to adjust its behaviour based on memory, parallelism, and iterations, catering to specific security requirements and performance needs.

As these configurations are crucial for computing the original hash, Argon2 provides robust resilience against brute-force and side-channel attacks [al17]. The resulting enhanced security makes Argon2 suitable for password storage and key-derivation in various applications and systems.

In 2015, Argon2 won the Password Hashing Competition [The].[11]

[7] A salt is a randomly generated sequence of characters, unique to each instance that gets hashed. Argon2's intension is to have a 128-bit salt for all applications but this can be sliced in half, if storage is a concern [al17].

[9] Side-channel timing attacks analyze execution time variations in cryptographic systems to reveal confidential data, exploiting differences in time caused by varying inputs, branching conditions, or memory access patterns.

[9] Due to the nature of prioritizing security, LessPM uses the third configuration. This is expanded upon in Section **??**.

[10] A dictionary attack is an approach where an attacker tries to find a hash by searching through a dictionary of precomputed hashes or generating hashes based on a dictionary commonly used by individuals or businesses.

[11] NIST's competition to find an encryption algorithm inspired the Password Hashing Competition, but it took place without NIST's endorsement.

2.4 JSON Web Token & JSON Web Encryption

Although JSON Web Tokens (JWTs) [JBS15] are not inherently encrypted, they still serve an essential purpose for some forms of secure data transfer. By combining JWTs with JSON Web Encryption (JWE) [IET15], secure intercommunication between two or more parties can be achieved.

2.4.1 JSON Web Token (RFC 7519)

JWTs were introduced as part of RFC 7519 in 2015 [JBS15]. It is a compact, URL-safe string intended to transfer data between two entities. They can be used as part of an authentication and authorization scheme in a web service, application, or API. The data in the string is intended as a payload and is referred to as a `claim` [JBS15].

A JWT typically consist of three parts: header, payload, and signature. The header and payload are serialized into JavaScript Object Notation (JSON) and then encoded using a Base64Url encoding to ensure a URL-safe format [JBS15].

The token contains an expiry timestamp, which, when decoded, is validated if the timestamp is not passed at the time of decoding. JWTs can be cryptographically signed using various algorithms like Hash-Based Message Authentication (HMAC), Rivest-Shamir-Adleman (RSA), or Elliptic-Curve Digital Signature Algorithm (ECDSA) ensuring the integrity and authenticity of the token [JBS15]. This prevents unknown authorities from constructing or hijacking existing tokens.[12]

The URL-safe format of a JWT is often performed through a Base64 encoding, which permits larger bits of data to be sent in a compressed, safe[13] format [JBS15]. JWTs are widely used for scenarios such as single sign-on (SSO), user authentication, and securing API endpoints by providing an efficient and stateless mechanism for transmitting information about the user's identity, permissions, and other relevant data [Kar18].

The process functions as follows:

1. The client completed an authentication request.
2. A token was constructed and created through a claim on the server. The claim could have been any data the server wished to use to authenticate the legitimacy of a future request.
3. The claim was signed with the desired algorithm. This could also have been a secret, stored on the server.
4. As part of the response to a request, the server appended the token.[14]
5. The client received the token and carried it upon the next performed request.
6. When the server received the token again, it validated the `exp` property of the JSON object and took action accordingly.

2.4.2 JSON Web Encryption (RFC 7516)

Complimenting the JWT standard are JWEs (RFC 7516). Introduced around the same time as JWTs, in May 2015, there are many parallels between the two, but the significant distinction between them is that JWEs are encrypted [IET15]. This encryption can happen through AES (see Section 2.2) and provides integrity and authenticity for the token. This prevents eavesdropping or tampering with the token during transit. Combined with JWT, JWE enhance the overall security of JWT-based implementations, making them more suitable for transferring sensitive data between intercommunicating entities on the Web.

[12] Hijacking a token could happen by a man-in-the-middle attack. This is done by a third-party individual listening and intercepting traffic in order to either read data or input their own in a client's request. This would allow an attacker to gain access to privileged information.

[13] Safe in this context should not be interchanged with secure. We reference safe as a way to transfer the data over the selected protocol, in most cases HTTP(S).

[14] A common place to embed these tokens is in the Authorization part of the HTTP header [JBS15].

3 Literature Review

Passwordless Authentication (PA) is a growing field of study within Computer Science, as traditional authentication methods like passwords are increasingly recognized as vulnerable to attacks such as phishing[15] and credential stuffing.[16] Passwords and sensitive information can also be a victim of successful brute-force attacks [Bon12] through data leakages by hacking or purchasing information on the dark Web.

Following NIST [Nat20; Pau17], authentication should consist of covering one of the three principles in Figure 5.

- **Something you know**: Such as a password, an answer to a personal question, or a Personal Identification Number (PIN).
- **Something you have**: A device that contains some token or cryptographically signed keys.
- **Something you are**: Biometrics of any sort or kind. Facial recognition, retina scan, fingerprint and similar.

Figure 5: The Three Principles of Password Security [Sch00; Nat20].

There are many approaches to passwordless authentication or a second step to authenticate with a common password.[17] In 2022, Parmar et al.[Par+22] described several attractive solutions, along with their advantages and drawbacks, for performing PA using the most common methods. The study discovered that PA commonly gets accepted as the most frictionless authentication system for User Interfaces (UI) [Par+22]. Biometrics was mentioned as one of the authentication methods, concluding that it captures universal human traits, encouraging differentiation from one another [Par+22]. The same study raises the caution surrounding the loss of authentication devices and how fingerprints can be compromised [Par+22].[18]

One promising approach is using the FIDO Alliance's collaborative work with W3C to create WebAuthn. WebAuthn permits users to authenticate through biometric information stored on a device in the user's possession (i.e. phone, computer) or a physical security key (i.e. YubiKey, Nitrokey, Etc.) [Wor21].

In [Hus22], Huseynov utilized a Web interface with WebAuthn to create credentials that users could use for a VPN. The client required a user to authenticate through the procedure of WebAuthn (see Section 2.1). On a successful request, the Remote Authentication Dial-In User Service (RADIUS) creates a temporary username and password, which would then be transferred as a response to the end-user, permitting them to copy and paste it into the necessary client, alternatively to construct a batch file which would establish the correct connection [Hus22]. The study suggested creating a solution for a VPN client which embedded some browser components [Hus22].

Gordin et al. [Gor21] implemented PA into an OpenStack environment using WebAuthn, which increases security and bypasses the risk of malefactors employing leaked passwords on other services.[19] The PA process considerably reduced the number of attacks towards a server because the server no longer has user authentication secrets [Gor21]. They, however, utilized a fingerprint as the primary biometrics, citing cost as a primary

[15] Phishing is a form of attack where a hacker tries to leverage Social Engineering to act as a trusted entity to dupe a victim to give away credentials by opening an email, instant message, or text message, then signing into a spoofed website, seeming legitimate [OWAnd].

[16] Credential stuffing refers to the practice of using automated tools to inject compromised or stolen username and password combinations into web login forms to gain unauthorized access to user accounts [Mulnd].

[17] Often referenced as Two-Factor Authentication or Multi-Factor Authentication.

[18] The security implication of using the core concept of FIDO2's WebAuthn is subject to storage in the system on Apple-specific devices [App23a]. On an Android device, the implementation is up to the manufacturer of the device, where Samsung has implemented a Physically Unclonable Function [Lee+21].

[19] See Section 2.1 for further explanation as to how this works.

factor to discourage the use of FIDO2 [Gor21].[20]

Statistia reported in 2020 that between 77–86% of smartphones now have a form of biometric scanner built into their device [Sav22]. Gorin, Et al. continue by mentioning that some individuals have trouble using their biometric scanner or getting it to work correctly on their device [Gor21], which can be a potential drawback for user adaptability.

According to a study conducted by Lyastani et al. in 2020 [Ghr+20], a significant portion of users found the usage of WebAuthn and Fido2 standard to be easy and secure, but with some concerns about losing access to their accounts or fear of others accessing their accounts [Ghr+20]. The study utilized a fingerprint Yubikey for the authentication process. Despite some reservations, the study found that overall, WebAuthn and Fido2 have good usability for passwordless authentication [Ghr+20].

The authors reported that users automatically associated the loss of the AD with losing access to the account [Ghr+20] – and vice versa – indicating that users are slightly unwilling to replace the initial principle of *"Something you know"* with the second *"Something you have"* and third *"Something you are"* principle. Additional research is necessary to educate users, increase trust and confidence in the technology, and address concerns about the potential loss of account access.[21]

Morii et al. [Mor+17] investigated the potential of FIDO as a viable PA solution in 2017 when the FIDO2 and WebAuthn standards were yet to be widely adopted. The authors noted that, at that time, only the Edge browser had implemented proper support for FIDO2 and WebAuthn [Mor+17]. Despite the limited browser support, the study demonstrated the feasibility of integrating PA into the well-established authentication system, Shibboleth [Shi22; Mor+17].[22]

As the technology evolved from FIDO to FIDO2.0, some security concerns persisted, such as session hijacking[23], which can compromise user accounts [Mor+17], highlighting the need to protect these.

Since the publication of Morii et al.'s research, browser support for FIDO2 and WebAuthn has significantly improved, with major browsers like Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge, and Vivaldi now offering support for these standards. This broader adoption has enabled the more widespread deployment of PA solutions, providing increased security and improved user experiences across various online services.

However, the ongoing evolution of security threats and the increasing sophistication of attackers highlights the need for continuous research and development in the field of passwordless authentication, ensuring that new methods and standards are both secure and user-friendly.

Having explored various studies and developments in the field of PA, it is evident that this area has been continuously evolving to provide secure, user-friendly solutions. However, the implementation of these solutions into stand-alone, real-world applications, such as password managers, independent of other login systems, is a critical aspect that requires thorough investigation.

In the following section, we will examine the methodology employed in this study to integrate a passwordless system into a password manager, taking into account the challenges and concerns identified in the literature. By doing so, we aim to contribute to the growing body of knowledge on passwordless authentication and its practical applications.

[20] Authentication was provided through the Keystone environment on the OpenStack platform.

[21] We believe that these concerns are mostly raised due to using a YubiKey and that using a phone-based authenticator would reveal other results.

[22] Shibboleth is a widely-used, open-source federated identity solution that enables secure single sign-on across multiple applications and organizations.

[23] Session hijacking refers to an attacker gaining unauthorized access to a user's authenticated session, often exploiting weaknesses in the handling of cookies, sessions, or JSON Web Tokens (JWT). See Section 2.4.

4 Methodology

Exploring the development and implementation of LessPM, a passwordless password manager, our focus will be on the key components, technologies, and steps that form the system's development process. A crucial part of the development and system was its security and robustness.

Our approach encompasses an explanation of the system architecture, the technologies and tools utilized and the development process to create the prototype. By providing a comprehensive account of the system development process, we aim to enable readers to understand the technical aspects of our work as well as to assess the validity and relevance of our findings.

4.1 Environment

To implement the system, we approached the development from a type-safety environment.

4.1.1 Server

We chose Rust as the programming language for our backend, which provided significant benefits regarding the software development environment.

Rust's emphasis on safety and performance allowed us to create a highly secure and efficient environment for our implementation without the risks typically associated with memory-related vulnerabilities [Riv19]. The built-in memory management and focus on concurrency ensures that the software runs smoothly, which is crucial [FLP85] when dealing with sensitive user data.

Additionally, Rust's surrounding ecosystem is at rapid growth [Kor], containing a vast library of high-quality crates[24], which enables expeditious development and easy integration of various functionality. Utilizing Rust's distinctive features, the passwordless password manager provides enhanced security and reliability in the context of user authentication [Riv19], showcasing the benefits of utilizing a modern programming language.

The backend ran an instance of an HTTPS server, serving as a wrapper for the sensitive data.[25] The Chromium developers mandate this constraint to guarantee that the pertinent API is invoked exclusively within a secure context [web16]. During development, we established a secure connection by using a self-signed certificate for the "`localhost`" domain.

4.1.2 Client

An essential part of the implementation was to create a viable client that could function as a visual entry point to the server. For the simplicity of the project, we chose React as a framework. React is a JavaScript library for building user interfaces, offering an efficient and flexible approach to web development.

As is customary when developing a system containing authentication and authorization, we took advantage of JWT/JWE (Section 2.4) to retain some information to authorize the client between requested resources. We strongly encrypted the Base64-encoded data passed between the server and client using AES-256 (Section 2.2).

We took advantage of React's ability to construct a single-page application with no routing capabilities, avoiding the possibility of utilizing any URL tampering or manipulation[26] to attempt privilege escalation or accessing data restricted from them.

Further, the server checked and verified the token before proceeding with any requests made to it. This information is updated and inspected between each re-render [Rea] of the client.

As a deliberate decision, we opted for MongoDB due to its NoSQL architecture, which facilitated the storage of Object-like structures [Mon21] including pass-

[24] `Crate` is the Rust-specific name for a package or library.

[25] We took advantage of the Authorization header during development, as specified in RFC 7519. However, the framework we used for the server required us to expose the usage of `Authorization header` to access it in the client. See Section 6 for further explanation.

[26] Malefactors can perform URL Manipulation, which involves modifying a URL to request resources that would otherwise be inaccessible to a user.

When initiating the project's development, we had a strong vision of creating a client that could seamlessly integrate with Chromium-based browsers as an extension.[a]

[a] In the client's project folder are traces of a manifest.json file and build scripts to have the client run in an extension.

However, we quickly discovered that the Rust cargo used to perform WebAuthn requests did not implement this support. We reported this issue upstream to the authors of the cargo, and we will continue to work with the authors to ensure a proper implementation of this functionality in the future.

4.2 WebAuthn

We used WebAuthn to perform passwordless authentication. As an open standard, WebAuthn aims to provide a key-based authentication scheme to strongly authenticate users. When users sign up for a new service, their device generates a new key-pair (See Section 2.1).

WebAuthn attempted to mitigate phishing, man-in-the-middle, and brute-force attacks through its intuitive design [Wor21]. By leveraging the increasing market of smartphones with biometrics [Sav22], WebAuthn becomes a natural extension in terms of authentication.

4.2.1 Registration

When a user tried registering in LessPM, we first check whether a user with a similar name existed. We would deny the registration if the user existed. If no user had that name within the system, we generated a unique ID using UUID V4 and started the necessary Registration Ceremony.

We generated a JWE during this process, which was signed with RSASSA-PSS using SHA-512 before getting encrypted with AES-256 (See Section 6). This claim is then attached to the request's `Authorization` header, along with the Creation options from WebAuthn (See Section 2.1).

The `expiration` time for this claim is set to one minute to allow the user some time to authenticate.[27] The claim expired after this minute, and the user would then have to restart the process. This expiration timer was a decision we made to emphasize security within LessPM further.

When the response returned to the user, LessPM used the browser's built-in WebAuthn API to prompt the user for their authentication (See Section 2.1). At this point, it is entirely up to the user to decide what device to use to authenticate. We used an Apple iPhone 13 Pro Max, a Samsung S21, and a Samsung Galaxy A52 to test authentication.[28]

We scanned the QR codes prompted through our phones, which initiated the key-pair generation on the device after a successful facial recognition. [29] The public key then gets transmitted to the browser and sent to the Relying Party through a new request.

Before the request reaches the Relying Party, an authentication middleware checks for the JWE sent earlier and denies the request with an `Unauthorized` HTTP status code if the request is invalid. [30] If WebAuthn can validate the key and metadata sent with the request, the user gets stored in the database, along with the UUID generated.

4.2.2 Authentication

Authentication worked similarly to the registration process. Upon receiving an incoming request, the server checked the database. The server immediately rejected the request if the user did not exist in the database.

Further, the server attempted an Authentication

[27] WebAuthn describes a timeout performed within the system. In our case, this timeout is one minute. However, we chose to add this extra step to secure the project further, and JWEs require an expiry time. See Section 2.4.

[28] Other alternatives included a YubiKey, NitroKey, Etc.

[29] There is a question of concern that photography can bypass facial recognition. Apple uses built-in sensors to scan depth, colours and a dot projector to create a 3D scan of a person's face. However, this approach prevents the use of photography to authenticate through their FaceID and TrueDepth technology [App23b].

[30] Validity in this context means not timed out, tampered with, or similar.

Ceremony by collecting the public key from the database. Upon validation[31] of the public key, the server creates a new JWE, which also receives an `expiry` of one minute for the user to authenticate (See Section 2.1), before sending that and the challenge in the response. We then prompt the user to authenticate with their original authenticator that they used in the registration step, having the authenticator sign the challenge before performing a new request to the server with the signed challenge. The server considered the user authenticated if the Relying Party accepted the signed challenge.

Finally, the user had a 15-minute timeframe to perform the necessary activity within LessPM. The specification for JWT does not specify upper- or lower bounds for this value [JBS15].

4.2.3 Password Creation & Retrieval

Passwords are sensitive in nature, so it seems only natural in a security context to enforce an extra level of authentication upon retrieving and creating one unique password.[32]

A user is presented with the following options when they initiate a password creation:

- **User Identifier**: An identification that the user wants to associate with the password entry they are storing. Such as a username, phone number, or email.
- **Website**: A URL or similar where the password belongs.
- **Password**: The user is prompted with the input to create a strong password automatically, choosing options such as numbers, special symbols, smaller or larger characters, and the length. As an option, the user was also permitted to construct their password but warned by a warning saying that this option is less secure.

As a final step before a password is created and stored, the user is prompted to reauthenticate with their AD. The password is created and stored in the database if this authentication process is successful.

Users needed to make a new authentication request when they wanted to retrieve the plaintext version of the password. This step is done in part to validate the user's presence[33] but also to gather the necessary bits for the key-reconstruction (See Section 4.5).

4.3 Cors

Cross-Origin Resource Sharing (CORS) must be configured correctly when the server and client are running separately on different ports.

When a web page tries to access a resource hosted on another domain, browsers perform an additional request to the server, called a "`preflight`". The preflight request determines whether the request that the web page is trying to make to the server is allowed. This request is done through the `OPTIONS` method in HTTP and contains some information about the origin, accepted Content-Type, and similar for the actual request.

The server responds to this with what methods and headers are allowed, denying the actual request from ever happening if the preflight is not successful.

We constructed a CORS `layer`[34] which contained the two domains for the server and client, permitted credentials[35] and then permitted the two HTTP methods `POST` and `GET`. We also ensured the `Content-Type, Authorization`, and `Cookie` headers are permitted.

Any other methods or headers should abort the request in the preflight.

[31] Validation in this context only means that the key stored in the database is valid.

[32] We retrieved a complete list of the user's passwords upon successful authentication. The hashed version of the password is stripped of the returned values to protect and enforce security.

[33] This approach is also used in password managers on phones to avoid situations where a user might have left their computer unlocked.

[34] In this context, we referred to a layer as a wrapper around all other requests.

[35] To pass the JWT token back and forth between the server

4.4 Cookie

JavaScript can access and manipulate `Cookies` [HXC18]. We utilized the browser's local cookie storage to attempt secure authentication between requests.[36] We attempted a couple of strategies listed below to fortify the cookie that LessPM set in the browser against a malefactor:

- **Strict SameSite**: This ensures that the cookie is safeguarded against Cross-Site Request Forgery (CSRF) attacks and remains restricted to its original origin domain.
- **Expires**: Once the system authenticated the user, it gave the cookie a Time-to-Live (TTL) mechanism similar to the JWE, which remained valid for only 15 minutes.
- **Secure**: We applied the `Secure` attribute to ensure that the cookie was only accessible through the HTTPS protocol. This protocol encrypts the data being sent back and forth between the client and the server, attempting to avoid eavesdroppers.
- **HttpOnly**: Setting HttpOnly tells the browser to make this cookie inaccessible through JavaScript. This property is important to mitigate session hijacking.

4.5 Password Encryption

A password can be stored and hashed using a `salt` in a typical authentication scheme. When trying to authenticate, the user would provide their UID and password. These values are collected from the database and checked with the random salt[37] that was generated when the user registered.

Securing the applicant's passwords made the process more complicated than the one described above. Furthermore, since the passwords in a password manager should be a randomly generated string unknown to the applicant, we cannot hash it and compare the input from the user.

We decided on the AES-256 (see Section 2.2), as a high form of encryption is paramount to securing the passwords. AES requires a 256-bit key to encrypt and decrypt the data. Since the CID generated by WebAuthn is unique and random for each application and device, this serves as a basis for constructing the key.

We based the key for each password on the following premise:

1. We take advantage of the fact that each CID is unique in every application (see Section 2.1). We therefore used 192-bits of the string that is the CID, converting it to integers. Since every CID is unique depending on the device, we generated padding to reach the remaining difference when the CID does not contain 192-bit.
2. We appended each key with a random 128-bit salt of integers and stored these bits with the entry in the database.
3. Then we add a 128-bit of pepper collected from the environment variable to finish it.

We used this 448-bit as the input for the key-derivation function, Argon2 (See Section 2.3). Upon hashing, each key is also subject to a 128-bit salt explicitly generated for the key.

[36] The cookie storage in a browser is subject to any vulnerabilities that can be performed on an SQLite database while having access to the computer where it is running.

[37] Salting is the process of adding a randomly generated string consisting of arbitrary characters to the password before creating a hash [See15].

4.6 Hashing

When we searched for a good key-derivation function, we first came across Password-Based Key Derivation Function 2 (PBKDF2). We saw PBKDF2 as a good solution for the project, but after researching the topic further, we ended up with Argon2.
Argon2 is regarded by some to be more secure than PBKDF2 due to its modern design considerations, including protection against side-channel attacks and memory-hardness, which make it more resistant to brute-forced and rainbow table attacks. Argon2 also offers customization of parameters for flexibility in adapting to changing hardware technologies and security requirements. PBKDF2 offers to set an amount of iterations to construct the hash and which pseudorandom function to use.

Since LessPM is required to run in as safe of an environment as possible, Argon2's configuration option is an excellent solution.

4.6.1 Configuring Argon2

Argon2's hashing output is dependable on configurations [al17].[38] Given that we emphasized security, we opted for the `Argon2id` configuration, which gave us equal protection against side-channel and brute-force attacks.

A part of Argon2's customizability is the offer to set the option for a required amount of memory to do the hashing.

These options would force a malefactor to use a specific amount of memory for each attempt to construct the hash. The only way a malefactor can get passed this requirement is to purchase more memory.[39]

For LessPM, we used 128 Megabytes of memory to construct the hash.[40] We went for the default suggestion of two iterations to complement the memory. To finalize the configuration, we added 8 degrees of parallelism since the system where we developed it consists of 8 cores. [41]

4.7 JWT & JWE

The JWT contained data we could use to validate an authenticated user between HTTP requests, which we encoded as a URL-safe base64 string (See Section 2.4). According to [JBS15], these should be attached to the authorization part of a request, along with a `Bearer` part. These tokens are cryptographically signed with RSASSA-PSS using SHA-512 with a key-pair stored on the server to avoid forgery.

For each step in registration 4.2.1, authentication 4.2.2, and password creation/retrieval 4.2.3, different tokens with different information are constructed, verified, and discarded.[42] We opted for this approach so that an exposed token could not serve as more than one entry point for one of the steps.

By default, these tokens are not encrypted. It is out of the question to unnecessarily expose any sensitive details about the server's inner workings. Therefore, we decided to incorporate a part inspired by JWE.[43] Before any token is attached to a response from the server, the token is encrypted using AES-256. The Base64 gets encrypted into a ciphertext and then turned into a new Base64 encoding. An example of this can be seen in Appendix A.1.

[38] Dependable in this context refers to each configuration that can possibly be constructed. An instance of Argon2 with 256 Megabytes of `memory` will not return the same hash as 255 Megabytes. The same is true for the amount of `iterations` and `parallelism`.

[39] As a side note, the increase in memory usage will scale as technology evolves and more memory becomes common.

[40] According to [al17], the reference implementation using Argon2d with 4Gb of memory and 8-degree parallelism, the hashing process should take 0.5 seconds on a CPU with 2Gz. However, we were unsuccessful in seeing anywhere close to similar results.

[41] The degree of parallelism is affected by how many cores a CPU contains [al17].

[42] The token for password creation/retrieval is the same as verifying that a user is signed in and authorized to create entries on behalf of the user.

[43] To properly implement JWE, metadata should be encoded into the token, such as what algorithm performs the encryption [IET15]. For further explanation, see Section 6.

We decoded the Base64, decrypted the ciphertext, and decoded the JWT-Base64 to reconstruct the claim when the server received a new request. We used this claim to gather details about the user who made the request and responded accurately to the request accordingly.

The following section

4.8 Security Analysis

The security of a passwordless password manager is of paramount importance to protect sensitive user infromation and prevent unauthorized access. In this subsection, we conduct a comprehensive security analysis, identifying potential attack vectors and outlining the defensive measures that have been implemented to mitigate tehse risks.

The defensive measures include industry-standard security protocols such as HTTPS, JWT/JWE, Argon2 hashing, salt and pepper techniques, CORS configuration, and secure cookie management. Additionally, different JWTs are used for registration, authentication, and password creation/retrieval to minimize the risk of exposure. These defensive measures collectively aim to provide a robust and secure architecture for LessPM, safeguarding against various threats and ensuring the confidentiality, integrity, and availability of user data.

- **HTTP**

  Weakness: Any server running HTTP (Hypertext Transfer Protocol) is passing data between a server and a connecting client. With HTTP, this data is unencrypted. A malefactor can eavesdrop this data, performing a man-in-the-middle attack, replacing information, reading tokens, perform header inections, Etc. with this unencrypted traffic.

  Defensive Mechanism: Upgrading the HTTP connection to HTTPS ensures that communication between the client and server is encrypted through Transport Layer Security (TLS). It provides confidentiality and integrity of data transmitted over the network, making it more difficult for attackers to intercept or tamper with sensitive information.

- **JSON Web Tokens (JWT)**

  Weakness: JWTs are signed with a secret specific to the server. However, if this secret is discovered or leaked, a malefactor could use this information to sign their own tokens and provide their own information. Since JWTs are signed and not encrypted, any malefactor who receives access to a token can decode the Base64 format and read the tokens information in clear text.

  Defensive Mechanism: Properly signining the tokens with a strong algorithm prevents information from being tampered with. An attacker can still read the information in cleartext but can not forge their own tokens. LessPM uses RSASSA-PSS with Sha-512 to sign and encode tokens, decoding them with the keypair when received. This ensures integrity and authenticity of the JWT, as the signature is verified by the server. RSASSA-PSS is a robust and secure signature algorithm that provides protection against various attacks, such as collision attacks and length extension attacks. To further impose a level of security, LessPM takes advantage of different tokens for registration, authentication, and authorization.

- **JSON Web Encryption (JWE**

  Weakness: JSON Web Encryption are JWTs signed and encrypted using an algorithm for signing and encryption. However, if this encryption algorithm is not potent or weak (such as a Caesar Cipher), an attacker can break the encryption or even encrypt their own tokens. JWEs are also subject to scenarios such as lack of key rotation, insecure length of the encryption key and improper implementation (See Section 6).

Defensive Mechanism: To combat these weaknesses, LessPM implements encryption partly inspired by the JWE standard. LessPM encrypts the token using AES-256, a strong encryption scheme capable of 2^{256} different keys. This ensures that the data is confidentially kept and protected from unauthorized access. While inspired, it is important to note that JWEs have not been properly implemented in LessPM (See Section 6).

- **Hashing Passwords & Keys**

  Weakness: Hashing is a one-way process to convert information (such as passwords) into a fixed-size string of characters, typically a fixed-length hash value. However, the hashing process is susceptible to a malefactor using pre-computed hash-values for large number of possible passwords, stored in a lookup table called a Rainbow table attack. Hashes are also vulnerable to dictionary attacks, and not being properly implemented without using a salt and preferable a pepper, or brute-force attacks.

  Defensive Mechanism LessPM takes advantage of the latest within public hashing functionality, through Argon2. Argon2 constructs a memory-hard and computationally expensive hash that provides protection against brute-force, rainbow table, dictionary and side-channel attacks by requiring a significant amount of computational resources and time to compute the hash. Finally, LessPM takes advantage of using both a random salt unique for each password and a pepper stored in the source code of 128-bit for both. This creates a higher threshold for a malefactor by requiring access to the database and the source code to be able to quickly compute a hash. The salt and pepper are added to the key of the AES-256

- **Password Authentication**

  Weakness A password is something a secret that a user knows. Secrets belonging to an individual will always be potentially accessible to a malefactor. There are multiple vectors that can be used, such as a user being careless and writing the password down on a piece of paper or storing the secret in an unencrypted file. Further, a password is required to be stored somewhere for a server to authenticate the user, preferably with an email for recovery. This creates two new vectors for a malefactor to exploit, the persisted storage on the server or the recovery process.

  Defensive Mechanism LessPM takes advantage of WebAuthn to avoid the usage of passwords to authenticate the user. The user is required to have two vectors for authentication; their authenticator device and the biometrics, which is the user itself. It is, however, important to mention that any authenticator device is vulnerable to being compromised. Should a device be compromised or infected with malicious software (Malware), it could be used to intercept or manipulate authentication requests.

- **Storing passwords in plaintext**

  Weakness: Storing a password can be done in plaintext in a database protected by a password. However, should an attacker get access to the database, all passwords and their plaintext are compromised. Storing the passwords in plaintext makes the information accessible to any individual with access to the database, which causes a privacy risk.

  Defensive Mechanism: LessPM encrypt all passwords with one of the strongest public symmetric encryption processes, AES-256. Each key for each password is unique, and consists of the 192-bit CID from WebAuthn generated when the user registered (See Section 4.5). This CID is unique to each service the user is registered with. Along the 192-bit CID, 128-bit is used from a salt and 128-bit used from a pepper. The salt is stored with the entry

in the database. Using both salt and pepper enhances the security of the derived key.

- **Cross-Site Scripting, Cross-Site Request Forgery & Unauthorized Access**

  Weakness: Any HTTP server improperly configured is subject Cross-Site Scripting (XSS) and CSRF. XSS is a process where a malefactor could inject a malicious script from a different origin, and execute said script in the context of a user's browser. This could lead to a situation where a malefactor could get access tokens or cookies, or interact with the webpage a user is viewing through the script. CSRF is a type of security vulnerability where a malefactor tricks a user to unknowingly make unauthorized requests on a trusted website, such as a bank or similar. This can lead to actions being performed on a user's behalf unintentionally, without the user's consent, leading to unauthorized access.

  Defensive Mechanism: CORS is a measure that enforces a strict policy for which domains and services are permitted to access certain resources on the server. LessPM takes advantage of CORS by permitting the server itself and the client associated domain to access resources on the server. This is a preventive measure put in place to allow the communication between the client and the serve, even though they are running on different ports, and potentially different domains. This ensures that only authorized clients can access the server's resources, preventing unauthorized cross-origin requests and protecting against cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks.

- **Cookies Weakness:** A cookie is vulnerable to many aspects of security. Improperly stored, a cookie can be accessed through JavaScript, sent to different domains, get hijacked, excessive expire time, or even poisoned.[44]

  Defensive Mechanism: LessPM only uses one cookie, which contains the encrypted JWT. The cookie is protected through built-in browser-features such as restricted to the same origin that the cookie came from, and cannot be sent anywhere else, further preventing XSS of sensitive information. The cookie is expired after 15 minutes, which is extensive amount of time for a user to have authorized access to their passwords, before the need to reauthenticate their identity. Upon creation, LessPM makes sure that the cookie becomes set to secure. This prevents the cookie from being sent over an insecure HTTP connection, limiting it to HTTPS. Finally, the cookie is HttpOnly, so that the cookie can't be access through JavaScript, reducing the attack vector of SS further.

5 Conclusion

This report describes the details of implementing a passwordless password manager, using WebAuthn to authenticate users. The aim of the project was to create a secure and efficient solutions for managing passwords that doesn't rely on the need for a traditional password.

We came up with an intuitive way to encrypt passwords, instead of the traditional hash. To achieve this, AES-256 was applied and Argon2 was used to construct a hash out of a key supplied through WebAuthn's Credential ID and a randomly generated salt for both key-derivation function and the AES-256 key, combined with a pepper for the latter.

The backend where WebAuthn and the password manager was running consisted of a HTTPS server, serving content through a self-signed certificate. Because the server and client were independent of

[44] Cookie Poisoning is the process where an attacker modifies the content of the cookie to inject malicious data or bypass security controls.

each other, we also configured CORS.

To keep a user authenticated between requests, we used an encrypted version of a JWT, inspired by the JWE RFC. This was achieved by storing the token in a fortified cookie in the client, being encrypted on the server before it was stored.

Despite its potential benefits, our report also highlights some limitations and challenges associated with passwordless authentication. These include the potential lack of adaptability by users and the possibility of losing the authenticator device.

Our findings suggest that future studies should explore ways to address the issue of device loss in the context of passwordless authentication. In particular, research should focus on developing methods for securely recovering access to accounts and data when a user's primary authentication device is lost or stolen. This could include the use of backup authentication methods, such as biometric verification or recovery codes, as well as the development of secure protocols for remotely revoking access to lost devices. By addressing this challenge, we can enhance the security and reliability of passwordless authentication systems.

6 Future work

Through implementing LessPM, we aimed to create a barebone implementation that could serve as a reliable Minimal Viable Product (MVP). However, we recognize that more work is needed to further enhance and compliment the product.

The related topics to further improve LessPM are listed below and briefly discussed as a way to highlight potential drawbacks of the current version.

- **Authorization Headers**
  Even if [JBS15] specifies the `Authorization` header as the appropriate place to append the header, in a system using WebAuthn it would be better to construct a cookie with similar settings as mentioned earlier in the report. This is of particular concern as the HTTP framework we used requires to specify exposure of the `Authorization` header in order for JavaScript to read it. Exposing it to the client entails exposing it to malefactors as well.

- **Hashing the stored password**
  We emphasized and recognized that only constructing a hash for the AES key leaves the stored password exposed **iff** AES gets broken or have an unknown zero-day failure. Adding some form of hashing for the password before encrypting it would serve an exceptional benefit. As of this writing, we are unsure how this hash would be properly implemented, given that we have nothing to construct the hash for the password. A potential suggestion would be to apply similar methods to how the encryption was performed.

- **Hardcoded AES for JWT**
  Each JWT is encrypted with the same AES-256 key. This serves as a great treat to the whole system and was implemented this way due to the lack of time during the implementation phase. In a perfect scenario, we would have applied the same technique for the JWT token as for the passwords.

- **Encrypted Passkey**
  The Passkey that gets stored on the user object could be beneficial to encrypt. While the Argon2 hash of the key serves as a great way to require an attacker to get access to the codebase as well as the database, the passkey could be encrypted with AES and the Credential ID serve as part of the key to decrypt it.

- **Attaching connecting IP to JWT**
  We would have liked to attach the connecting IP address to the JWT. Seeing that a JWT token is exposed to the client and then a form of session hijacking, attaching an IP address to the token serves as a first-line defence in order

to avoid exposure of unencrypted tokens.[45]

- **Properly implement JWE**
  During development, we thought that JWTs were encrypted and not just signed. We discovered the JWEs [IET15] late in the process and these were new to us. Due to lack of time, we therefore took the shortcut of just implementing the encryption process of JWE, not the remaining metadata. In the future, we would like to properly implement these and follow the standard.

- **Multiple Authenticators**
  In its current implementation, LessPM supports a single registered authenticator per username to maintain a focused security approach. During registration, the server checks the database for an existing username similar to the incoming one and aborts the registration ceremony if a match is found.

  While WebAuthn permits users to have multiple authenticators, limiting this feature in the initial iteration of LessPM helps ensure a more controlled security environment. As the system evolves, considering addition of support for multiple authenticators can be weighed against potential security risks and benefits.

- **Return Passwords with Authentication**
  As part of the authentication process, we could make sure that the password list is returned with the last request to authenticate. This would further emphasize security by not exposing passwords in a separate end-point.

  On the same note, it would be good to handle the decryption part of the JWE better, seeing that a misconfigured cookie sent to this end-point now returns an error. Yet another technical debt as a cause of lack of time. This is, however, not something that brings the system to a halt, rather an error that needs handling and nothing about what error it is exposed to the requesting party.

[45] This would not work on a mobile device connected to mobile data, seeing that the IP address switches between connections.

7 Acknowledgement

The implementation of this project was only possible with the dedicated and hardworking community surrounding Rust.

As a thank you, we would like to acknowledge the following cargos and their hardworking developers: jsonwebtoken, ring, webauthn-rs.

While there are other libraries to mention, these are the most prominent ones that serve as the basis of the project. All the libraries are well-respected within the community, with the former two having several million downloads and the latter well over 100.000.

ChatGPT Throughout the report, the authors utilized ChatGPT to thoroughly reiterate concepts used during development.

References

[21a] *Web Authentication: An API for accessing Public Key Credentials Level 2 - Credential ID.* `https://www.w3.org/TR/webauthn-2/#credential-id`. Accessed: 2023-03-25. World Wide Web Consortium, 2021.

[21b] *Web Authentication: An API for accessing Public Key Credentials Level 2 - PublicKeyCredential identifier slot.* `https://www.w3.org/TR/webauthn-2/#dom-publickeycredential-identifier-slot`. Accessed: 2023-03-25. World Wide Web Consortium, 2021.

[65] *CTSS Programmers Guide.* 2nd ed. MIT Press, 1965.

[al17] Alex Biryukov Et al. *Argon2: The memory-hard function for password hashing and other applications*. Tech. rep. Accessed on: March 25, 2023. University of Luxembourg, 2017. URL: https://github.com/P-H-C/phc-winner-argon2/blob/master/argon2-specs.pdf.

[App23a] Inc Apple. *Secure Enclave*. 2023. URL: https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web.

[App23b] Apple Inc. *How to use Control Center on your iPhone, iPad, and iPod touch*. https://support.apple.com/en-us/HT208108. Accessed: March 30, 2023. 2023.

[Bon12] Joseph Bonneau. "The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords". In: *2012 IEEE Symposium on Security and Privacy*. Last Accessed: 2023-03-28. 2012. DOI: 10.1109/SP.2012.49.

[Doo18] John F. Dooly. *History of Cryptography and Cryptanalysis: Codes, Ciphers, and Their Algorithms*. Springer, 2018, p. 5. ISBN: 978-3-319-90442-9.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag Berlin Heidelberg, 2002. ISBN: 978-3-540-42580-9.

[FID17] FIDO Alliance. *FIDO UAF Overview*. https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-overview-v1.1-id-20170202.html. Accessed: 2023-03-25. Feb. 2017.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[Ghr+20] S. Ghrobany Lyastani et al. "Is Fido2 the kingslayer of User authentication? A comparative usability study of FIDO2 Passwordless Authentication". In: *2020 51st IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2020, pp. 102–108. DOI: 10.1109/SP40000.2020.00047.

[Gor21] Ionel et al. Gordin. "Moving forward passwordless authentication: challenges and implementations for the private cloud". In: *2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. Last Accessed: 2023-03-28. IEEE. 2021. DOI: 10.1109/RoEduNet54112.2021.9638271.

[Hus22] Emin Huseynov. "Passwordless VPN Using FIDO2 Security Keys: Modern Authentication Security for Legacy VPN Systems". In: *2022 4th International Conference on Data Intelligence and Security (ICDIS)*. 2022, pp. 453–455. DOI: 10.1109/ICDIS55630.2022.00075.

[HXC18] Xincheng He, Lei Xu, and Chunliu Cha. "Malicious JavaScript Code Detection Based on Hybrid Analysis". In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)* (2018), pp. 1–5.

[IET15] IETF. *RFC 7516: JSON Web Encryption (JWE)*. https://www.rfc-editor.org/rfc/rfc7516.txt. Accessed: 2023-03-26. May 2015.

[JBS15] Michael B. Jones, John Bradley, and Nat Sakimura. *RFC 7519 - JSON Web Token (JWT)*. https://www.rfc-editor.org/rfc/rfc7519. Accessed: 2023-03-26. Internet Engineering Task Force (IETF), May 2015.

[Kar18] Chetan Karande. *Securing Node Applications: Protecting Against Malicious Attacks*. O'Reilly Media, Inc., 2018. ISBN: 9781491999644.

[Kor] Kornelski. *Lib.rs Stats*. https://lib.rs/stats. [Accessed: Mar 29, 2023].

[Lee+21] Yongki Lee et al. "Samsung Physically Unclonable Function (SAMPUF™) and its integration with Samsung Security System". In: *2021 IEEE Custom Integrated Circuits Conference (CICC)*. Last Accessed: 2023-03-28. Access through IEEE Explore. IEEE. 2021, pp. 1–4.

[Lev84] Steven Levy. *Hackers: Heroes of the Computer Revolution*. Sebastopol, CA: O'Reilly Media, 1984, pp. 85–102. ISBN: 978-1449388393.

[LH90] Karl J. Lieberherr and Ian M. Holland. "Assuring Good Style for Object-Oriented Programs". In: *IEEE Software* 7.5 (1990), pp. 38–48. DOI: 10.1109/52.35588.

[Mon21] MongoDB. *NoSQL Explained*. Last Accessed: 2023-03-29. 2021. URL: https://www.mongodb.com/nosql-explained.

[Mor+17] Michitomo Morii et al. "Research on Integrated Authentication Using Passwordless Authentication Method". In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. IEEE. Tokushima University, 2017, pp. 895–900. DOI: 10.1109/COMPSAC.2017.198.

[Mulnd] Neal Muller. *Credential Stuffing*. https://owasp.org/www-community/attacks/Credential_stuffing. Last Accessed: 2023-03-28. n.d.

[Nat20] National Institute of Standards and Technology. *Special Publication 800-171 Revision 2: Protecting Controlled Unclassified Information in Nonfederal Systems and Organizations*. Page 24, Chapter 3. 2020. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-171r2.pdf.

[NIS00] NIST. *Commerce Department Announces Winner of Global Information Security Competition*. Web page. Last Accessed: 2023-03-27. Oct. 2000. URL: https://www.nist.gov/news-events/news/2000/10/commerce-department-announces-winner-global-information-security.

[OWAnd] OWASP. *PHISHING IN DEPTH*. https://owasp.org/www-chapter-ghana/assets/slides/OWASP_Presentation_FINAL.pdf. Last Accessed: 2023-03-28. n.d.

[Par+22] Viral Parmar et al. "A Comprehensive Study on Passwordless Authentication". In: *2022 International Conference on Smart Computing and Data Science (ICSCDS)*. 2022. DOI: 10.1109/ICSCDS53736.2022.9760934.

[Pau17] James L. Fenton Paul A. Grassi Michael E. Garcia. *Digital Identity Guidelines: Authentication and Lifecycle Management*. Tech. rep. SP 800-63-3. National Institute of Standards and Technology, 2017. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-3.pdf.

[Pol23] Polybius. *The Histories*. Accessed on: March 23, 2023. 2023. URL: http://www.perseus.tufts.edu/hopper/text?doc=Perseus%3Atext%3A1999.

01.0234%3Abook%3D6%3Achapter%3D34.

[Rea] React developers. *React Component*. https://react.dev/reference/react/Component#setstate. Last Accessed 2023-03-23.

[Riv19] Elijah E. Rivera. *Preserving Memory Safety in Safe Rust during Interactions with Unsafe Languages*. Tech. rep. Defense Technical Information Center, 2019. URL: https://apps.dtic.mil/sti/trecms/pdf/AD1188941.pdf.

[Sav22] Justina Alexandra Sava. *Biometrics-enabled consumer device adoption in 2020*. Statista. Last Accessed: 2023-03-28. June 2022. URL: https://www.statista.com/statistics/1226096/biometrics-enabled-devices-by-region/?locale=en.

[Sch00] Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. Wiley, 2000.

[Sch15] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 20th Anniversary Edition. Wiley, 2015. ISBN: 978-1119096726.

[See15] Alok Sharma Seema Kharod Nidhi Sharma. "An Improved Hashing Based Password Security Scheme Using Salting and Differential Masking". In: *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*. IEEE, 2015, pp. 1–6. DOI: 10.1109/ICRITO.2015.7359225.

[Shi22] Shibboleth Consortium. *Shibboleth - Federated Identity Solutions*. Accessed: 2023-03-28. 2022. URL: https://www.shibboleth.net/.

[ST01a] National Institute of Standards and Technology. *Announcing the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. 3.1 Input and Output, p7-8. Last Accessed: 2023-03-26. NIST, Nov. 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[ST01b] National Institute of Standards and Technology. *Announcing the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. Introduction, p5. Last Accessed: 2023-03-26. NIST, Nov. 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[ST01c] National Institute of Standards and Technology. *Announcing the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. Algorithm Specification, p13-14. Last Accessed: 2023-03-26. NIST, Nov. 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[ST01d] National Institute of Standards and Technology. *Announcing the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. Appendix C, C3, p42-46. Last Accessed: 2023-03-26. NIST, Nov. 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[The] The Password Hashing Competition. *Password Hashing Competition*. https://password-hashing.net. Accessed on: March 25, 2023.

[W3C21a] W3C. *Web Authentication: An API for accessing Public Key Credentials*

- *Level 2. Section 6.1 Authenticator Data*. Accessed: 2023-03-26. 2021. URL: https://www.w3.org/TR/webauthn-2/#authenticator-data.

[W3C21b] W3C. *Web Authentication: An API for accessing Public Key Credentials Level 2*. https://www.w3.org/TR/webauthn-2/#sctn-registering-a-new-credential. Accessed on: March 25, 2023. 2021.

[W3C21c] W3C. *Web Authentication: An API for accessing Public Key Credentials Level 2*. https://www.w3.org/TR/webauthn-2/#webauthn-relying-party. Accessed on: March 25, 2023. 2021.

[W3C21d] W3C. *Web Authentication: An API for accessing Public Key Credentials Level 2*. https://www.w3.org/TR/webauthn-2/#registration-ceremony. Accessed on: March 25, 2023. 2021.

[Wan+18] Chun Wang et al. "The Next Domino to Fall: Empirical Analysis of User Passwords across Online Services". In: Mar. 2018. DOI: https://doi.org/10.1145/3176258.3176332. URL: https://people.cs.vt.edu/gangwang/pass.pdf.

[web16] web.dev. *Credential Management API*. Last Accessed: 2023-03-29. 2016. URL: https://web.dev/security-credential-management/.

[Wor21] World Wide Web Consortium. *Web Authentication: An API for accessing Public Key Credentials Level 2*. https://www.w3.org/TR/webauthn-2/. Accessed: 2023-03-25. Apr. 2021.

[Yan+00] Jianxin Yan et al. *The Memorability and Security of Passwords: Some Empirical Results*. Tech. rep. UCAM-CL-TR-500. University of Cambridge, Computer Laboratory, Sept. 2000. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-500.pdf.

A Appendix

A.1 Base64 and ciphertext

```
uAaQghfTS0jpMA1WaYozepQ4/TpN13Y6tlKSzXG0l+epVZK+
   vjvD8BwMIlWVxvTaV3mcFxL665qBNsFg//81hpU8I660lq/LAXsPcdfq2vr8YRa14+GH+
   Gtw6YlqLrDU3E8Rhb/
   IlAvJ8u5VN8pwV1SmBZLTwL7AyEWlp4GodSrX4NSl5grIFoVwRq7kXofVu1aUToD6KJcmo0X

BnwG0KWpEPUZFPd76KcA/QfXHnJHQsaR2jKWFJdRCpbtJAacAbssJk/
   bJjioo3AS1caEVZbNJctp9xgqVvgQJPyhmYtMLqdjq/
   SocUscTrLSPiR2X0g5sWByNIm6ses2SJ3dtMYYeOr7+qtVOcoX+U7w2+
   uLorawsCcXCmXRunEKd5jXiydwXZjzPoKHaT8hGwDB8CNSHxg/
   JXrezEJ5JKwq3Gio3xEyjD09Cfq5qLn9kENbcDZ/uRZK6+
   Swxcqg1DYhGngPJbbPkbOpqcXxdLutybYgdGpFN0zCQw3/LNbxYzOBeVhVsXM+
   GOVEAcKgYpnyCILwKKtFUyaRX2Q7IjdBbKP8NpG7RWZKFtRBVc4YVdWSIRpfek1/
   lFkq1JrvgK/6KjyyR+
   m6sb2RzUzxNO1V5uTkH2m8cBUwQBUqjiNEgVQDzTeaYIZqH0j2Is4cblRcCsjoFgX3Nvh4/
   OvgpgQ==
```

```
KHO
Vi3z8:Mv:RqU;

" U W y 6 ‘5<#{qakpj.O U7pWTEu*
pF^ VN (&E)jD=FE=)?A    v % B I n          *(qeIr}
        ?(fb*RlX"nz  ’wm1: N r S 6  ,     tnB5f  <  OFO  |   ^I+
*7LOB~j.-d
E,1r5
Fmljq|].rmMO-5X^Vl\   9Qp   |
*T  W  [(
nYUsudF _J&+,nodsS<M;Uy9oLJ  U M !=:}  ?:)
```


Figure 6: The Base64 encoding and relevant ciphertext after AES-256.