# Password Manager - Product Evaluation

23/08/2021

**Lukasz Baldyga**

**Computer Forensics and Security**

c3538194

# 1. Abstract

The development of a password manager with a focus on security that's right for the user but also recovery methods, such as Shamir's Secret Sharing Scheme, and an adapted implementation of Bitcoin's BIP39. It discusses the methodology and technologies used in the developmental process.

# Table of Contents

# 2. Introduction

The purpose of this this project is to create a password manager. There are many different password managers with different functions and protections in place, however, this password manager is designed to prevent advanced adversarial threats, and it keeps the data safe above all.

It is not unreasonable to assume any adversary that attempts to decrypt the password manager's encryption might not be willing to allow the user to walk away free, or worse - the adversary may cause the user to expire in a painful way; or - the user may want to terminate themselves given the alternative of torture. Due to this, this password manager allows the user to distribute a shared recovery scheme before being interrogated that allows other people to unlock and read the contents of the password manager. There is no way to tell how many shared recovery schemes have been made and if done correctly, this also means that any adversary would also not know who the pieces have been distributed to. A shared recovery scheme is a scheme where the user creates a number of pieces and an opening threshold where a secret (like a master key) can be derived. Both the threshold and pieces of the scheme can be set independently of each other. (Adi Shamir, 1979). Alternatively, another backup method exists that does not use the shared recovery scheme if the user trusts a sole entity or wants to back up the master key and keep it in a very secure place.

This password manager also differs from convention given the implemented authentication method. The currently implemented method allows multiple passwords to be used to unlock the password manager.

Another distinct feature of the password manager is that it implements "Identities", which are fundamentally a collection of accounts. The password manager is designed as a supplement for operating on the Tor network (Dingledine, Mathewson and Syverson, 2004). Having identities ensures that the user does not accidentally "link", accounts that they do not want to be linked together, allowing the user to assume an identity per Tor session. This allows for a much safer operating security when doing sensitive tasks.

# 3. Review of Literature

There exist many password managers that offer similar protection as the password manager created in this project. They all fundamentally rely on symmetric or asymmetric encryption. Offline password managers, like KeePassXC, primarily rely on AES (Nechvatal *et al.*, 1999) as the symmetric encryption cipher. The most popular password managers use AES with PBKDF2 (B. Kaliski and RSA Laboratories, 2000) (Dan Callahan and Mozilla, 2014) (LastPass, 2019).

## 3.1 The mainstream encryption approach for password managers

### 3.1.1 AES

The most well known and widely used, AES (also known as Advanced Encryption Standard and previously as Rijndael) (Nechvatal *et al.*, 1999) is the mainstream approach to symmetric encryption used by password managers. This is because because AES is the most well tested and a tried algorithm at the time. Not many password managers exist that use something else other than AES. The NSA uses AES (Philip Bulman, 2000) to encrypt their most sensitive data, so it would also make sense that a lot of encryption applications would use it as their primary encryption.

Unfortunately, AES is starting to show its age. AES can take keys up to 32 bytes. Usually only 16 byte keys are secure enough for AES, however, due to the fact that quantum computers can half the brute force time in block ciphers (Lane Wagner, 2021) by using Grover's algorithm (Grover, 1996), only using 32 byte keys is acceptable.

### 3.1.2 PBKDF2

The PBUDF2 (B. Kaliski and RSA Laboratories, 2000) key derivation function is a widely used function used to encrypt disk data (cryptsetup, 2020), as well as password managers (Dan Callahan and Mozilla, 2014) (Bitwarden, 2021). It is the algorithm recommended by NIST (Meltem Sönmez Turan *et al.*, 2010).

The function's cost can be updated and dynamically scaled depending on the usage. This means that as technology advances, the cost can be adjusted to be very expensive for the adversary, while keeping the user secure.

This fundamental flaw with this function is that it can be implemented as an ASIC and also accelerated by GPUs (Ondrej Mosnáˇcek, 2015). This is because of the low memory requirement by the algorithm.

# 3.2 Alternative encryption approach for password managers

Alternative approaches exist to encryption. Serpent (Ross Anderson, Eli Biham, and Lars Knudsen, 1998) and Blowfish (B. Schneier, 1994) encryption algorithms are examples of an alternative ciphers. There also exist new and better Key Derivation Functions, such as Scrypt (Colin Percival, 2009; Percival and Josefsson, 2016) and Argon2 (A. Biryukov *et al.*, 2021, p. 2) (Password-Hashing.net, 2019).

## 3.2.1 Ciphers

### *3.2.1.1 Serpent*

Serpent (Ross Anderson, Eli Biham, and Lars Knudsen, 1998) is a block cipher. It was a candidate for the Advanced Encryption Standard, and was ranked second to Rijndael. It was deemed to have a larger security margin than Rijndael (Nechvatal *et al.*, 1999).

Like any block cipher, Serpent can take keys up to 32 bytes and suffers from the same quantum attack as AES (Philip Bulman, 2000).

The most likely reasons the Serpent algorithm was not chosen in the competition is because it is significantly slower across all platforms (Nechvatal *et al.*, 1999) and the fact that it suffers from an attack when implemented in a smart cards (Chari *et al.*, 1999).

### *3.2.1.2 Blowfish*

The Blowfish algorithm (B. Schneier, 1994) is a block cipher. It can take keys up to 48 bytes in size. Like AES, Blowfish does not show evidence of being fully cracked and also produces an output that is indistinguishable from random. Blowfish has been tried and tested given its age.

The fact that Blowfish can take 56 byte keys means that it might be more secure than AES when Grover’s algorithm (Grover, 1996) is applied.

## 3.2.2 Key Derivation Functions

### *3.2.2.1 Argon2*

The winner of the Password Hashing competition in 2019 was the Argon2 algorithm (Password-Hashing.net, 2019). There exists a draft for the Argon2 function on the Internet Engineering Task Force (IETF)’s website (A. Biryukov *et al.*, 2021).

It allows for memory cost and time cost control independently, unlike Scrypt. This allows for greater control and also better ASIC and GPU resistance, which is the biggest threat to these algorithms.

The only problem that arises is that this is a very recent algorithm, much younger than Scrypt. This

means that there hasn't been much time for experts to give thorough scrutiny for this algorithm. There may be cryptanalysis that exists in the future for this specific algorithm. Regardless, this is also an algorithm worthy of consideration.

#### *3.2.2.2 Scrypt*

Scrypt (Percival and Josefsson, 2016) is a recent algorithm. It is a "memory hard" (Colin Percival, 2009) algorithm, meaning that it requires a lot of memory to compute in order to try and be hardware resistant. The algorithm has been designed with ASIC computation threats in mind, whereas PBKDF2 is not. In the case of this algorithm, it is possible to adjust its memory and time costs simultaneously; as computational cost rises, memory costs also rise.

However, since Scrypt is a recent algorithm it is so not as heavily scrutinised as PBKDF2 or Bcrypt. It might not be as stable as the tried and tested PBKDF2.

Like PBKDF2, it shares the same fundamental flaw: it is not ASIC or GPU resistant (anymore). The problem with adjusting the memory and time cost simultaneously is that you cannot fine tune the algorithm.

# 3.3 Recovery

## 3.3.1 Bitcoin BIP39

Bitcoin has it's own improvement protocol for backing up and storing the private key of the wallet. The proposal outlines a method of encoding bits of data into human readable words. This is relevant to the password manager as the password manager encrypts and decrypts using a master key. Serialising a master key in a similar way to Bitcoin's BIP39 (Marek Palatinus *et al.*, 2013) allows a method of backup that is better for humans to understand, rather than have it in a raw hexadecimal form.

## 3.3.2 Shamir's Secret Sharing Scheme

Shamir's scheme (Adi Shamir, 1979) solves the problem of requiring multiple people to actively contribute a piece of information in order to obtain a secret, other than each person having their own "locks" and "keys". This is very useful if the secret that needs to be shared is the master key of the password manager. The secret can then be serialised into a BIP and shared between desired participants.

# 4. Review of technologies

## 4.1 Programming Languages

### 4.1.1 Python 3

Python 3 is a very versatile language that is supported on virtually every platform, from embedded systems to mobile phones. It was the first language considered for the product. It has has a brilliant package manager, pip (The pip developers, 2021), which would be subtile for every need.

The biggest downside is that Python family of languages are not very subtile for very large projects, like a password manager with a front end, back end and external libraries. The dynamically typed language makes it difficult to develop applications that have a large scale.

### 4.1.2 JavaScript + Browser Implementation

The advantage of the JavaScript implementation would have made the password manager compatible with everything that ran the Chrome or Firefox browser, including mobile.

Unfortunately, browser libraries were simply was only capable enough of the most popular algorithms, namely PBKDF2 and AES. Implementations of Argon2 or Blowfish algorithms simply did not work in a modern browser.

Furthermore, any attack on the hosting service of the website would provide an attack surface capable of leaking and stealing user's passwords. This very simple exploitation would simply involve poisoning the JavaScript source code, leading to massive data breaches.

For these reasons, a browser only implementation had to be scrapped, however, this implementation was then used in the Electron implementation.

### 4.1.3 NodeJS + Electron Implementation

The advantage of using NodeJS is that it already contains a huge variety of packages, it also includes auditing and testing capabilities through the use of npm. NodeJS (also known simply as "Node" or "Node.JS") is "a platform built on Chrome's runtime" (Cantelon *et al.*, 2014). Ultimately, Node runs JavaScript on the V8 virtual machine that runs on Google Chrome, however, instead of displaying pages on the web, it runs programs for the server side.

#### *4.1.3.1 TypeScript*

TypeScript adds types to JavaScript. This helps with large scale projects as plain JavaScript can "jiggle" between types because it is a loosely typed language. TypeScript solves this problem by

enforcing types and makes sure that types stay consistent across multiple files. This reduces errors when working with many files. (Microsoft, 2021)

#### *4.1.3.2 Electron*

Electron is essentially a Chrome browser that runs on top of Node. Electron provides a pass through layer to NodeJS. This means that Node specific APIs can be used in a Chrome browser environment. Unfortunately, anything written for Electron is not directly compatible with browser. However, it is necessary as a bare-bones API the browser provides doesn't have the necessary packages

However, to add support for devices like Android, Electron can be ported to Android-JS, which is a NodeJS like library. Android-JS ensures that every npm package can be used with Node in an android application meaning minimal changes need to be made across versions.

# 4.2 Implementation of encryption related algorithms

## 4.2.1 Serpent

Libraries for the Serpent algorithm are non-existent or deprecated for NodeJS. Developing Serpent into a NodeJS package is possible but doing so would be hard and difficult. Not to mention that it would certainly beyond the scope of this project: to make a secure password manager. Unfortunately, due to these reasons, Serpent had to be excluded from the final product.

## 4.2.2 Asymmetric encryption

Methods of asymmetric cryptography would not be susceptible for this project's password manager. The password manager does not need to communicate with entities that they do not know. The data is stored locally and is never sent away. The requirement is for data to be secured **only** locally.

Furthermore, encrypting the data asymmetrically would add unneeded overhead and would be the ultimately the weakest link in the encryption for a capable attacker (for example: a government) with a quantum computer. This is because it is relatively trivial for quantum computers to factorise numbers into their prime factors (Lavor, Manssur and Portugal, 2003).

In short, there is no sense in using asymmetric cryptography to encrypt things locally, and only brings downsides to security. It's an unnecessary risk.

# 4.3 GUI design

Styling for Electron works in a similar way to styling for any other modern web browser. The web browsers use Cascading Style Sheets (also known as simply "CSS"). Through the combination of structure given by HTML and styling provided by CSS, it is possible to make a modern GUI for users to use. There are many ready made CSS frameworks for rapid development of websites or web

applications.

## 4.3.1 Bootstrap 5

Bootstrap 5 is a popular CSS framework (Bootstrap, 2021). It allows for rapid prototyping and has many examples and has alternative open source MIT licensed themes. This framework has been chosen above others because of its ease of use and popularity. Having a popular framework means a larger support from the community if there is a problem. Furthermore, using a framework over traditional plain old CSS for positioning and styling elements means that the project doesn't have to focus on developing CSS also.

### *4.3.1.1 Popper*

Popper was cut because it depends on contain many unfixed vulnerabilities found by the npm auditing tool at the start of development. Bootstrap depends on Popper for creating pop up elements, however, it works just as well without Popper.

# 4.4 GIT

GIT is a version control software, initially created by Linus Torvalds, that tracks changes in files. It is used mainly to help collaboration between developers. It is also designed with speed and data integrity in mind (Loeliger and McCullough, 2012). It is the most popular versioning control software to date (Stack Overflow, 2018). The tool has been chosen to be used in the project as it can provide a timeline and

## 4.4.1 GitHub

Further expending on GIT, the online GIT hosting service "GitHub" provides further collaboration tools. It allows comments and issues where users can submit issues or suggestion and get help with their issues. GitHub also provides an alternative hosting of the code in the cloud. This means that there is no single point of failure in terms of storage of the code and the project, meaning recovery is possible even after catastrophic failure.

# 5. Methodology and Design

## 5.1 Agile Development

Agile (Fowler, Highsmith, and others, 2001) is a product development methodology. It relies on the product being delivered in small increments as opposed to delivering one product and having one version.

The Agile approach to development was used in this project to deliver the product. This happened in two steps, firstly as GIT commits, most of which delivered small improvements to the product; and secondly, when significant changes were made or the product lost backwards compatibility, a new release was uploaded.

This approach was chosen as it was unclear how many requirements were possible to deliver under the new deadline, and resource problems made it even harder to predict the time available for development. This methodology allowed the project to deliver as many set objectives as possible without running out of time.

### 5.1.1 Stages

Every iteration of Agile development there are a set number of stages that have to be done. During any specific stage, more Issues are added, and assigned priorities. This helps with planning for each iteration.

The first stage is to identify what is lacking and design implementations. This was done through the use of GitHub Issues and also the criteria set out at the start of the project. The issues are also categorised as either low, medium or high priority, and every iteration closes a number of these issues, prioritising the issues with the high quality.

The next stages implements a number of improvements and features outlined in the Issues, and are slowly implemented in accordance to their priorities.

Testing happens during and after the implementation process. This is to ensure that the product is doing what it should be.

The final stage is the review stage. This stage focuses on adding more Issues, reflecting what has been done and what needs to be created.

### 5.1.2 Version Control

Using version control, like GIT (Loeliger and McCullough, 2012), works well with Agile development. It is possible to use features of GIT like branch merging to indicate where a release has finished, and more fundamentally, it's possible to make every commit a small iteration on improving the program, as outlined above.

### 5.1.3 Object Orientated Development

The idea of splitting a program into multiple smaller abstractions or components is not new. In this approach to development, every piece of a program has its own class and own responsibilities. This interacts well with Agile methodologies because every component can be built independently in a sprint.

## 5.2 Waterfall Development

Waterfall development (Petersen, Wohlin and Baca, 2009) is a linear, progressive model. This means that every stage of this methodology must be done in order and there is no going back to previous steps, meaning once a stage like designing is completed, there is not redesigning later on in the project.

This approach to development was not used as it was unclear what requirements would be possible to add in the amount of time that was available. Rather than risking a non-submission, given the difficulties in resources, either by tests failing or by not finishing the full implementation on time, it's better to take an iterative approach.

# 6. Implementation and Testing

This is a security development project. The product that is produced is a password manager for a user under high adversarial threat, like a government of an oppressive nation. It's designed to supplement the TOR network by separating identities. It also presents the user with the ability to share access to the password manager by using Shamir's Scheme, or a BIP39 inspired recovery.

## 6.1 Code Scanning

The most important feature for this project is that GitHub provides is code scanning for vulnerabilities via CodeQL (GitHub, 2021). This tool detects bad code defined in the CWE (Common Weakness Enumeration) database (Mitre, 2021). This tool is important because it can detect and alert the developer, showing the exact line of code where the vulnerability exists, illustrated in Figure 2: CodeQL on GitHub. This tool has been beneficial to the production of this product as it detected some weaknesses in the prototype of the password manager.

## 6.2 Code Testing

The product was heavily influenced by the presence of object self testing. The project has been built with feedback from automated, manual unit tests and automatic code quality scanners. During every release of the password manager, the manual unit tests that were available all had to pass in order for a release of the product to be released.

Later in the developmental stages, the testing was done automatically and internal unit test were slowly phased out. The unit tests can be seen below in Figure 1: Testing of the password manager's components.

## 6.3 Agile Iterations

### 6.3.1 Prototype and Release Candidate 1

The first implementation phase was to build a working prototype, this was done by the first version of the password manager on the 27$^{th}$ of June, as this was the deadline underlined by an email sent out regarding the submission date. This was later cleared up and changed. On that day, only the first and most crude version of the password manager was developed. In this unrefined implementation showed it was possible to make a product that meets all criteria.

### 6.3.2 Release Candidate 2 & 3

These iterations of the password manager contained improvements to the storage method and security.

They also added HMAC to the container to verify the master key is correctly decrypted and removed possible data exposures. Release Candidate 3 specifically adds performance upgrades by making code asynchronous.

### 6.3.3 Release Candidate 4

This iteration of the password manager has a lot of security changes, bug fixes and structural changes. The first security improvement was randomising the IV of the encryption so that an adversary cannot see where the changes to the data began. The second major change was that the login page and the password manager page were in the same HTML document. This essentially fixed the bug described in Figure 2: CodeQL on GitHub. The final biggest change was that the releases of the password manager no longer shipped in debug mode by disabling console access.

### 6.3.4 Release Candidate 5 – Quality of Life

This iteration of the password manager removed a lot of developmental overhead, refactored a lot of code and improved the user experience of the password manager.

Some of the user experience improvements were settings and themes were implemented, giving the ability for the password manager to have different colours and layouts and for user to save these preferences in the password manager. It also brought

The next changes focused on reducing the file sizes, refactoring code, streamlining dependencies. This changed removed around 1,500 files from the repository, reduced the size of `PasswordManager.ts` from around 700 lines to roughly 200 lines, making it easier to work on this file later in the project.

This change also brought shortcuts to the recovery pages and recovery generation. This means that the users no longer need to click to check the underline checkbox with the mouse, and can simply either copy and paste the word or add a star (*) character as they're typing it in.

### 6.3.5 Release Candidate 6 – Danger Close

This iteration of the password manager focused on protecting the user from themselves. This means that any potential way that data can be lost has been protected by a "Are you sure?"-like prompt, allowing the user to change their mind.

Other changes exist too, like visual improvements to every part of the password manager by implementing Bootstrap accordions. Accordions split elements into sections, improving the user experience.

This is the first release that is completely backwards compatible with the previous release.

### 6.3.6 Version 1.0.0 (scheduled)

This is the first fully featured password manager release that has not be released at the writing of the report. So far, this release version adds documentation to the code, further testing to the password manager, removed further large files, fixes previously not found bugs and other miscellaneous changes.

Most of the report writing happened in this version of the password manager.

This release is also completely backwards compatible (until Release 5).

# 7. Product Evaluation

## 7.1 Creating a cryptographic container

A cryptographic container contains all of the information that the password manager needs, including settings, identities, accounts, password, and possibly more in the future. The cryptographic container is responsible for encrypting and decrypting itself to reveal its encrypted contents.

### 7.1.1 Recovery without password

Recovery without a password is possible if a user generates a Word Recovery in the form of a BIP. This recovery can be used to recover and unlock the container without a password. This has been fully implemented into the password manager.

### 7.1.2 Distribute recovery using Shamir's Secret Sharing Scheme

Recovery using Shamir's scheme (Adi Shamir, 1979) means that the product is able to create schemes based on the parameters that the user gives. The password manager produces a number of BIPs that the user can write down and distribute. The password manager is able to reverse the BIPs into a master key that can be used to open the container. This is a fully implemented feature of the password manager.

### 7.1.3 Multiple passwords open container

The cryptographic container can be open using multiple password, or an external master key. This requirement has been met.

# 7.2 Password manager features

## 7.2.1 Add/Remove/Update passwords

Yes. The password manager possesses the ability to manipulate the password in any shape the user desires.

## 7.2.2 Exporting passwords

Not directly. It is possible with commands in the debug version of the program, however, it is impossible to currently export the passwords out of the password manager. This is a low priority feature that wasn't implemented in the Agile development.

## 7.2.3 Change encryption specific settings

Only before creation of the first container, but since this is a low priority issue, it was skipped due to Agile development. This feature will be available in a future release of the password manager.

## 7.2.4 Secure password generation

The program generates secure passwords based on user's preferences. See Figure 4: Password generator settings for a full list of password preferences.

# 7.3 Standard password manager features

The password manager can:

- Add Accounts.
- Delete created accounts.
- Edit created accounts.
- Search account information for specific strings and display the relevant results.
- Generate passwords in accordance to the user's preferences.
- Change theme in accordance to user's preferences.

# 7.4 Storage Efficiency

A test has been done to test the proof of concept of storing a lot of passwords in the password manager's format. A container with 5 slots was created. To this container 100 identities containing 128 bytes of additional random information were added. To each identity in the container, 100 accounts containing 192 bytes of data were added. This gives us a grand total of 10,000 accounts stored in a password manager. The encrypted data of the password manager resulted in the size of less than 4MB. The code ran to prove this is a unit test written for the password manager, Figure 4: Unit test used to test the limit of the password. The current implementation of Electron/Chrome officially allows only up to 5MB of local storage, however, there have been patches added specifically for Electron that allow unlimited storage. If this happens to be trouble in the future, the password manager will move away from storing in local storage.

# 7.5 Hosting the password manager

Hosting the password manager was not required as the password manager is an offline application. Initially, when the password manager was going to be a browser only application. This meant that hosting would be necessary to hold all of the source code. Since the password manager is now an offline application with all of the files bundled with it, hosting only the source code and build files is required. This is done by using GitHub to distribute the source code.

# 8. Project Evaluation

## 8.1 Difficulties

Time difficulties were considerable, especially because of the limited resources and unexpected health issues. Mitigation helped with the delivery of this project, however, time and resource complications during the end of the project caused a late delivery by 3 days.

Initially creating the password manager showed it was impossible to implement in the form that met all of the requirements in the browser. The alternative solution would have been to start from scratch and rewrite, however, it was possible to continue development using Electron. That also meant delays as experience in the technologies was low, and there were at lot of unknowns at the start of development.

## 8.2 Aims

Most of the aims have been met, as dicussed in the product evaluation, since this project is mostly about the password manager. The only requirements that were not met were simple quality of life improvements, like exporting the user data in a compatible way to other password managers.

The research aims have been mostly met, as alternative encryption methods have been found and implemented into the password manager.

# 9. Summery and Conclusions

Security is hard. Implementing encryption related algorithms into software is difficult and there were many challenges, especially making sure that no data was leaking from the password manager.

The biggest advisory would be to give more time to the end of development. The final stages of writing this paper took over 10 days to complete due to complications in resources and health issues.

Secondary advisory would be to do testing early on, and to do random tests at a huge scale. This ensures that any edge cases that cause bugs are eliminated early, rather than finding them at inconvenient times.

The biggest help was the fact that developing the first prototype helped to see if all the required external packages existsed. Since the packages required for encryption did not exist in the first password managersteered the project away from being a browser based password manager to an electron based password manager.

Further work on the password manager could be done by closing all of the GitHub Issues, allowing online syncing and adding further features that users request. Due to the modular object orienteted programming of the password manager, it’s very simple to use different parts for different projects. Since the encryption is securly implemented

# 10. Bibliography

A. Biryukov *et al.* (2021) ‘The memory-hard Argon2 password hash and proof-of-work function draft-irtf-cfrg-argon2-13’, 11 March. Available at: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-argon2 (Accessed: 12 June 2021).

Adi Shamir (1979) ‘How to Share a Secret’, *Massachusetts Institute of Technology*, 22(11), pp. 612–613.

B. Kaliski and RSA Laboratories (2000) ‘PKCS #5: Password-Based Cryptography Specification Version 2.0’, September. Available at: https://www.ietf.org/rfc/rfc2898.txt (Accessed: 13 June 2021).

B. Schneier (1994) ‘The Blowfish Encryption Algorithm’, *Dr. Dobb's Journal*, 19(4), pp. 38–40.

Bitwarden (2021) ‘Encryption’. Available at: https://bitwarden.com/help/article/what-encryption-is-used/ (Accessed: 12 June 2021).

Bootstrap (2021) *Bootstrap - The most popular HTML, CSS, and JS library in the world.* Available at: https://getbootstrap.com/ (Accessed: 22 August 2021).

Cantelon, M. *et al.* (2014) *Node. js in Action*. Manning Greenwich.

Chari, S. *et al.* (1999) ‘A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards’, in *In Second Advanced Encryption Standard (AES) Candidate Conference*, pp. 133–147.

Colin Percival (2009) ‘STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS’. Available at: https://www.tarsnap.com/scrypt/scrypt.pdf (Accessed: 13 June 2021).

cryptsetup (2020) ‘Cryptsetup and LUKS - open-source disk encryption’, *What the ...?*, 21 December. Available at: https://gitlab.com/cryptsetup/cryptsetup (Accessed: 14 January 2021).

Dan Callahan and Mozilla (2014) ‘Firefox Sync’s New Security Model’, *Mozilla Services*, 30 April. Available at: https://blog.mozilla.org/services/2014/04/30/firefox-syncs-new-security-model/ (Accessed: 12 June 2021).

Dingledine, R., Mathewson, N. and Syverson, P. (2004) ‘Tor: The Second-Generation Onion Router’, *Paul Syverson*, 13.

Fowler, M., Highsmith, J., and others (2001) ‘The agile manifesto’, *Software development*, 9(8), pp. 28–35.

GitHub (2021) *CodeQL documentation*. Available at: https://codeql.github.com/docs/ (Accessed: 22 August 2021).

Grover, L. K. (1996) ‘A Fast Quantum Mechanical Algorithm for Database Search’, in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. New York, NY, USA:

Association for Computing Machinery (STOC '96), pp. 212–219. doi: 10.1145/237814.237866.

Lane Wagner (2021) 'Is AES-256 Quantum Resistant?', *Qvault.io*, 10 June. Available at: https://qvault.io/cryptography/is-aes-256-quantum-resistant/ (Accessed: 14 June 2021).

LastPass (2019) 'LastPass Technical Whitepaper'. Available at: https://assets.cdngetgo.com/1d/ee/d051d8f743b08f83ee8f3449c15d/lastpass-technical-whitepaper.pdf (Accessed: 11 June 2021).

Lavor, C., Manssur, L. R. U. and Portugal, R. (2003) 'Shor's Algorithm for Factoring Large Integers', *arXiv e-prints*, p. quant-ph/0303175.

Loeliger, J. and McCullough, M. (2012) *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media. Available at: https://books.google.co.uk/books?id=aM7-Oxo3qdQC.

Marek Palatinus *et al.* (2013) *Mnemonic code for generating deterministic keys, bips/bip-0039.mediawiki at master · bitcoin/bips*. Available at: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki (Accessed: 22 August 2021).

Meltem Sönmez Turan *et al.* (2010) 'Recommendation for Password-Based Key Derivation Part 1: Storage Applications'. NIST. Available at: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf (Accessed: 12 June 2021).

Microsoft (2021) *TypeScrypt: Why does TypeScript exist?, TypeScript*. Available at: https://www.typescriptlang.org/why-create-typescript (Accessed: 22 August 2021).

Mitre (2021) *CWE - About - CWE Overview*. Available at: https://cwe.mitre.org/about/index.html (Accessed: 22 August 2021).

Nechvatal, J. *et al.* (1999) 'Status Report on the First Round of the Development of the Advanced Encryption Standard', *Journal of Research of the National Institute of Standards and Technology*. 1999/10/01 edn, 104(5), pp. 435–459. doi: 10.6028/jres.104.027.

Ondrej Mosnáˇcek (2015) *Key derivation functions and their GPU implementation*. Masaryk University, Faculty of Informatics. Available at: https://is.muni.cz/th/409879/fi_b/?lang=en.

Password-Hashing.net (2019) 'Password Hashing Competition', *Password Hashing Competition and our recommendation for hashing passwords: Argon2*, 25 April. Available at: https://www.password-hashing.net/ (Accessed: 11 June 2021).

Percival, C. and Josefsson, S. (2016) *The scrypt Password-Based Key Derivation Function*. RFC Editor (Request for Comments, 7914). doi: 10.17487/RFC7914.

Petersen, K., Wohlin, C. and Baca, D. (2009) 'The waterfall model in large-scale development', in *International Conference on Product-Focused Software Process Improvement*. Springer, pp. 386–400.

Philip Bulman (2000) ‘Commerce Department Announces Winner of Global Information Security Competition’, 2 October. Available at: https://www.nist.gov/news-events/news/2000/10/commerce-department-announces-winner-global-information-security (Accessed: 14 June 2021).

Ross Anderson, Eli Biham, and Lars Knudsen (1998) ‘Serpent: A Proposal for the Advanced Encryption Standard’. Available at: https://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf (Accessed: 14 June 2021).

Stack Overflow (2018) ‘Stack Overflow Devloper Survey 2018’, *Developer Survey Results 2018*, 19 March. Available at: https://web.archive.org/web/20190530142357/https://insights.stackoverflow.com/survey/2018/#work-_-version-control (Accessed: 17 August 2021).

The pip developers (2021) *pip*, *pip documentation v21.2.4*. Available at: https://pip.pypa.io/en/stable/ (Accessed: 22 August 2021).

# 11. Appendices


*Figure 1: Testing of the password manager's components*

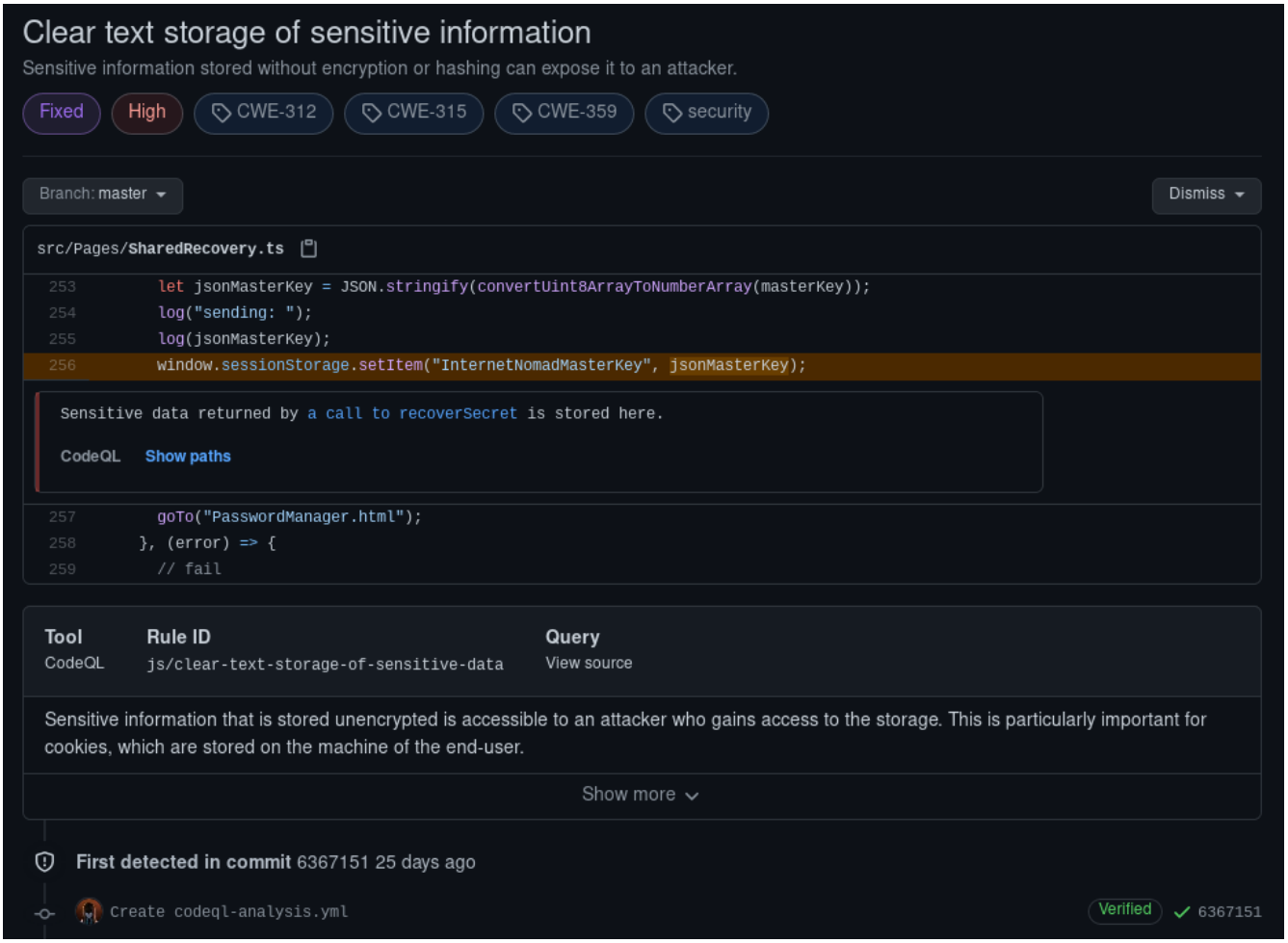


*Figure 2: CodeQL on GitHub*

```
async function Container_test_Blow_Argon2_Intensive_1() {
      let container = new Container();

      let password = randomCharacters(32);

      let identityName = randomCharacters(64);
      let identityData = JSON.stringify({
            "accounts": [],
            "identityDesc": randomCharacters(64),
            "identityName": identityName,
            });

      let algorithm = "Blow" as EncryptionType;
      let masterKey = getRandomBytes(algorithmBytes(algorithm));
      let containerIv = getRandomBytes(algorithmIvBytes(algorithm));
```

```
let testingIdentity = new Identity(identityData);

let identities: Identity[] = [];
// create 100 identities
for (let i = 0; i < 100; i++) {
      // create identity
      let iData = JSON.stringify(
            {
            "accounts": [],
            "identityDesc": randomCharacters(64),
            "identityName": randomCharacters(64),
            }
      );

      let identity = new Identity(iData);

      // create 100 accounts
      for (let i = 0; i < 100; i++) {
            let accountData = {
                  "website": randomCharacters(64),
                  "password": randomCharacters(64),
                  "login": randomCharacters(64),
                  "extra": new Extra().getJSON(),
            };

            identity.accounts.push(new Account(accountData));
      }

      identities.push(identity);
}

identities.push(testingIdentity);

container.identities = identities;
container.settings = new Settings();
container.iv = containerIv;
container.encryptionType = algorithm;
container.dataHash = encrypt(algorithm, masterKey, containerIv,
hash(masterKey));

// create slot
let iterations = 10;
let kdf: KeyDerivationFunction = "Argon2";
```

```
        let memory = 2 ** 15;
        await container.addSlot(randomCharacters(64) + "a", algorithm, iterations,
        kdf, memory, masterKey);
        await container.addSlot(randomCharacters(64) + "b", algorithm, iterations,
        kdf, memory, masterKey);
        await container.addSlot(randomCharacters(64) + "c", algorithm, iterations,
        kdf, memory, masterKey);
        await container.addSlot(password, algorithm, iterations, kdf, memory,
        masterKey);
        await container.addSlot("password", algorithm, iterations, kdf, memory,
        masterKey);
        await container.lock();

        await container.unlock(password);

        let containerJSON = container.getJSON()
        log(containerJSON);
        log(containerJSON.length);

        container.save();

        return container.identities == null ? false :
        container.identities[100].identityName == identityName;
}
```

*Figure 3: Unit test used to test the limit of the password*

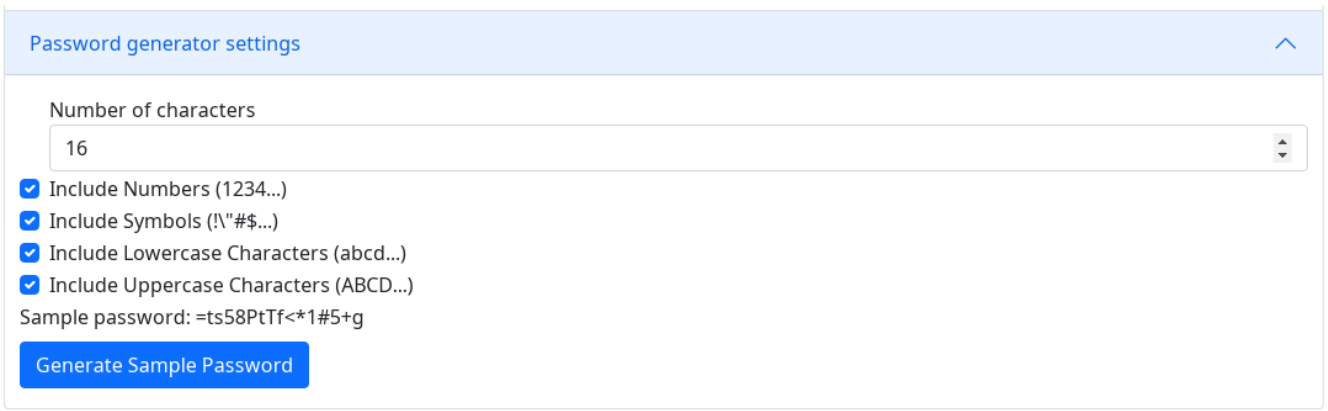


*Figure 4: Password generator settings*

# 12. Glossary

**BIP** – A human readable word representation of serialised binary data, similar in output to Bitcoin's BIP39 (Marek Palatinus *et al.*, 2013).