Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

Encrypting and Protecting Browser-Based Password Manager Metadata

Author: *Christos Protopapas (cp2n19@soton.ac.uk)*
Project Supervisor: *Ahmad Atamli (aa1m20@soton.ac.uk)*
Second Examiner: *Bahar Rastegari (br1e18@soton.ac.uk)*

A report submitted for the award of Computer Science Bsc

October 2021

Contents

Contents 1

1 Abstract 2

2 Statement of Originality 3

3 Acknowledgements 4

4 Literature Review 5
- 4.1 Introduction . . . 5
- 4.2 Password Manager Analysis . . . 5
- 4.3 App and Extension Based Password Managers . . . 5
- 4.4 Browser-Based Password Managers . . . 5
  - 4.4.1 Google Chrome . . . 5
  - 4.4.2 Google Chrome's Weaknesses . . . 6
  - 4.4.3 Mozilla Firefox . . . 7
  - 4.4.4 Mozilla Firefox's Weaknesses . . . 7
- 4.5 Differences, Weaknesses and Vulnerabilities . . . 8
- 4.6 Threat Model . . . 8

5 Problem Description and Goals 9
- 5.1 Problem Description . . . 9
- 5.2 Problem Scope and Goals . . . 9

6 Approach 10
- 6.1 Overview . . . 10
- 6.2 Tools and Libraries . . . 10
  - 6.2.1 Overview . . . 10
  - 6.2.2 Javascript, npm and CryptoJS . . . 10
  - 6.2.3 Webpack and SQL . . . 11
- 6.3 VirtualBox Windows 10 VM . . . 11
- 6.4 Development phase and Problems Encountered . . . 11
- 6.5 Design Decisions . . . 11
  - 6.5.1 Chrome Extension API . . . 12
  - 6.5.2 Advanced Encryption Standard - AES . . . 13
  - 6.5.3 Cipher Block Chaining - CBC . . . 13
  - 6.5.4 Password Based Key Derivation Function 2 - PBKDF2 . . . 14

7 Testing and End Product 15
- 7.1 End Product Overview . . . 15
- 7.2 Overall Testing of Extension . . . 15
- 7.3 Current Limitations and Potential Future Improvements . . . 16

8 Project Management 17

9 Conclusion and Suggestions 18

10 References 19

11 Appendices 22

1 Abstract

In our ever-expanding cyber world of today, the need to protect ourselves when traversing the Internet increases exponentially. Most of our data is locked behind accounts with passwords, and one of the most convenient software available to handle and protect them are password managers, that lock our credentials behind a single, master password, easing cognitive burden in having to remember them all. Every major Internet browser comes loaded with its' own variation of a password manager; the ones known as "browser-based". These ones, however, have a fair share of weakness and shortcomings compared to their extension and app-based ones, something that puts their users, as browser-based ones are easier to use, at a vulnerable position compared to the rest. These include, but are not limited to, outdated encryption schemes, storage of the encrypted passwords, and the various metadata generated by the password managers, which is the main focus of the research. This paper explores said weaknesses of those password managers, works on trying to bolster their security and suggests other general improvements.

2 Statement of Originality

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

ECS Statement of Originality Template, updated August 2018, Alex Weddell aiofficer@ecs.soton.ac.uk

3 Acknowledgements

I want to deeply thank my project supervisor, Dr. Ahmad Atamli, for his invaluable advice and help throughout the academic year, which helped this project take direction and form.

4 Literature Review

4.1 Introduction

I was inspired to come up with this idea after reading headline after headline of cyberattacks stealing user credentials. I researched into password policies and their storage, leading me to password managers and cryptography. Many academic papers have analyzed the security of password managers in the past - most of them have pointed out the existence of plaintext metadata[1,2,3,4,5], but none of them have gone into depth regarding it or proposed ways to protect it. As such, this paper will go into sufficient depth for this matter, explaining the inner structure of browser-based password managers, discussing the possible exploits one can achieve if they possess the aforementioned metadata, the importance of protecting it, and as well as attempting to solve the issue.

4.2 Password Manager Analysis

This work would be incomplete without mentioning the various benefits and functionality of password managers. These pieces of software offer ease-of-mind to its' user, in both the sense that they do not have to remember every username and password for every website, and also in the sense that their credentials are well protected under the barrier of encryption[1,2,5,6,22]. Most password managers offer some more accessibility features, like auto-saving new credentials when detected, or suggesting new ones when in need, auto-filling login boxes on sites where the user has saved their information, and some of them even keep track of various data breaches that might have happened and warn the users accordingly on whether or not they should update their password because of it. These pieces of software come in various forms, most notably three: app-based password managers, extension-based password managers, and the main focus of this paper, browser-based password managers[5,6,27,28].

4.3 App and Extension Based Password Managers

Firstly, it is worth noting that most extension based password managers are often simply an alternative, more lightweight version of an app-based one, offering the same base functionality straight through the browser, although sometimes one version might have features the other lacks, and vice versa. They usually include most of the features descibed in the section above, but it is worth noting that app-based ones cannot interact with the browser as they have no connection to it - hence they cannot know to autofill or autosave credentials, or suggest new ones. Both app and extension based PMs save their entries in a database almost always hosted in a secure cloud service, encrypted and protected by the provider of the password manager software. The encryption happens locally, and then it is send to your credential database, also called a "vault" for storage through secure channels.[5,28,34]

4.4 Browser-Based Password Managers

Browser-Based password managers, being integral parts of the browser itself, offer the best interactivity with the websites and other online applications and portals a user might have an account in, and hence the relevant credentials. They offer the options to save new, or update stored credentials, auto-fill functionality, and automatic password generation when in need of a new password for new accounts. All said credentials have their passwords encrypted and stored in a local database file within the browser's local installation directory, which can be synced to the cloud to be accessed from anywhere, if a user wishes to do so[2,5,6,7]. For the purposes of this paper, we have looked into two of the most popular web browsers, namely Google Chrome and Mozilla Firefox, have analysed their inner workings and procedures, and identified weaknesses in both of them, several of them shared by both.

4.4.1 Google Chrome

Chrome offers the whole assortment of password manager features and capabilities, as described in the above section, with the addition that it can also warn you when one of your stored passwords has potentially been compromised. Said credentials are stored in a SQLite database file named "Login Data", at the default location of "LocalAppData\Google\Chrome\User Data\Default\" [10]. Chrome also supports syncing your settings and data to the cloud by logging in with your Google Account. In this case another file similar in structure is created, named "Login Data for Account", which is the local copy of your credentials, retrieved

from Google's cloud storage. If any of the two, or both, files are removed, new ones are generated to take their place, with "Login Data" being empty, and "Login Data for Account" simply fetching another copy from the cloud. Encryption of passwords isn't handled by Chrome, but instead by the operating system, which in the case of Windows makes use of the "CryptProtectData" function built in the OS[30,31]. That results in ciphertext only decipherable by the system that encrypted it in the first place. However, although the password encryption is secure, the database also stores a great deal of metadata relevant to the aforementioned credentials. Those include several UNIX timestamps, like the date the entry was made, when it was last updated and/or used, the URL to the website that they belong to, the names of the fields to which the credentials are meant to go, and the username used for login in plaintext, as only the password is encrypted. It is worth mentioning that some websites use an email address to login instead of a username - something that might expose users to greater risk and other attack vectors, should a malicious entity get access to the database file.

4.4.2 Google Chrome's Weaknesses

Firstly, the fact that usernames and email addresses remain unencrypted and left as plaintext in the database is a potential severe security concern[15]. Anyone with this info can plan and execute other attacks targeting said credentials owner, even if they don't have knowledge of their password. The database is also easy to query: starting with ".schema", you can instantly find out how the database operates. Following this with a ".tables", you find out about all the tables present in it, and finally a simple SQL query such as "SELECT * FROM logins;" (with 'logins' being the table where all credential entries are stored) returns every entry in it and all associated metadata [10,26].

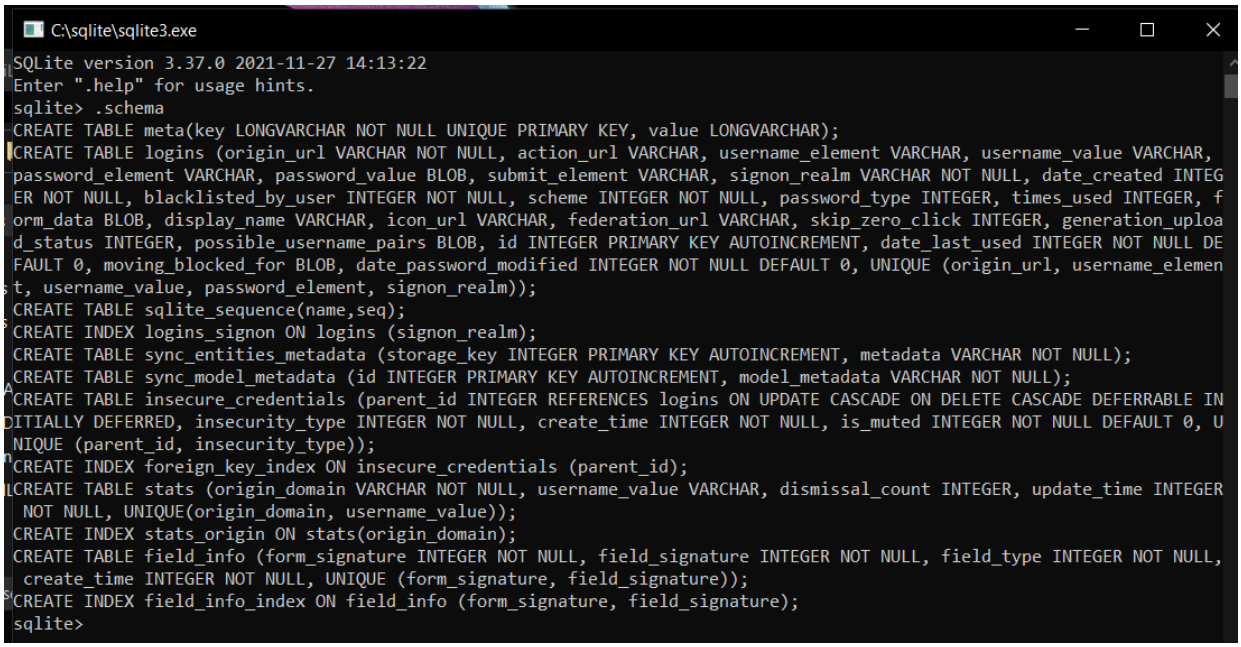

Figure 1: Schema of Chrome's 'Login Data' file ('Login Data for Account' shares the same schema) - Result of running ".schema" in SQLite3

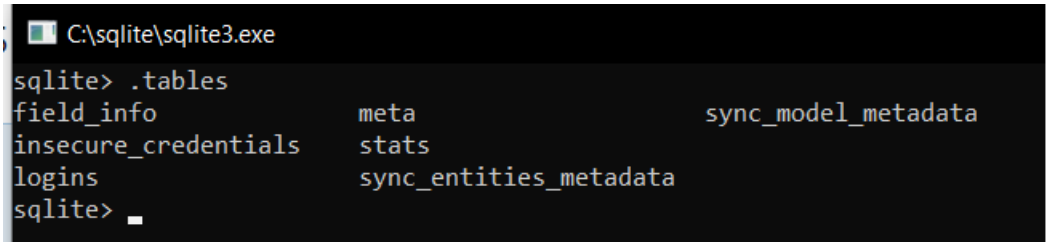

Figure 2: List of tables in "Login Data" - Result of running ".tables" in SQLite3

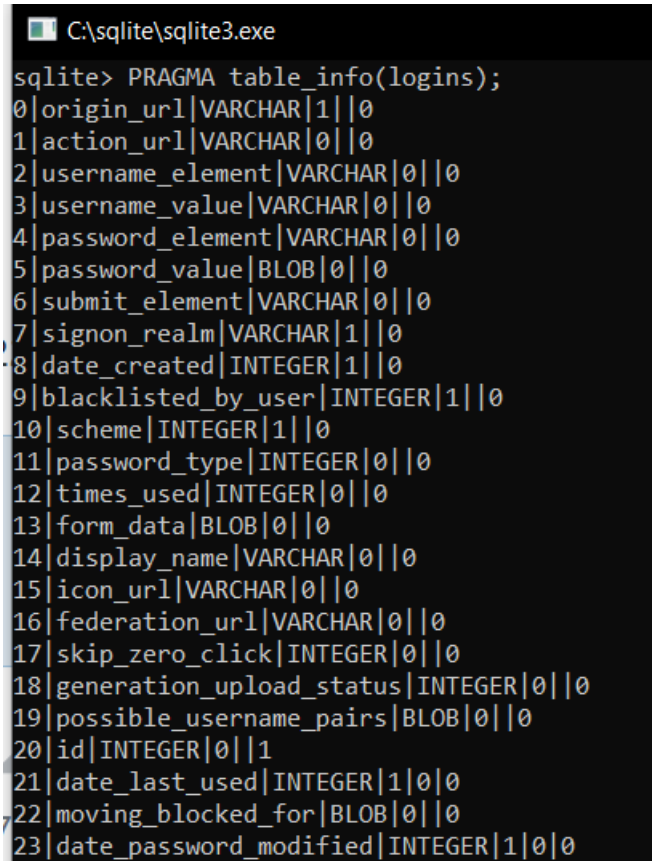

Figure 3: All metadata columns present in 'logins' date in "Login Data" - Result of running "PRAGMA table_info(logins) in SQLite3"

4.4.3 Mozilla Firefox

Firefox employs a similar approach to storing encrypted passwords. Firefox allows a user to choose a master password for their "password vault" for additional security, although not enforced in any way, encrypted with 3DES-CBC cipher. They are also stored in an SQLite database named "key4.db" in "AppData\Roaming\Mozilla\Firefox\Profiles\5nlwe76v.default-release" by default along with the encryption key and salt, as well in a JSON file named "logins.json" located in the same directory. Both files also contain a great deal of unencrypted metadata much like Google Chrome does. If a master password is not set, then firefox uses the empty string to encrypt the username and passwords only. Otherwise the master password is hashed with a single pass of the SHA-1 algorithm and stored in the "key4.db". The "logins.json" file remains unencrypted (except the credentials) regardless if a master password is set or not[8,9,11,12].

4.4.4 Mozilla Firefox's Weaknesses

What is notable is the odd decision for Firefox to use 3DES-CBC for the password encryption. The algorithm has been superceded and has long been considered obsolete and the security it provides minimal, considering today's computational capabilities. In addition the fact that it uses a default empty string for encryption on top of the outdated algorithm, makes it an even easier target. In the case of a master password being set, it it passed with a single hash of SHA-1 and stored in key4.db, where anyone can access it. However

none of the above affect the JSON file, where it is accessable and readable regardless of encryption or the existence of a master password[10,11,12,13].

4.5 Differences, Weaknesses and Vulnerabilities

The biggest and most notable difference between the two is the fact that firefox stores the credentials in two seperate files. Firefox's "key4.db" file can be made difficult to access if a user decides to use a master password for his vault, with which the entire file is encrypted with - otherwise, if not, firefox uses a default empty string to encrypt only the passwords in it and is much less secure than Chrome's encryption procedure. In fact, firefox's encryption has already been broken and is very easy for anyone to do so with scripts found online, like "firepwd"[12,13,14]. It is also worth noting that the JSON file, containing copies of the database entries, can still be easily accessible, regardless of whether a master password has been set, having its' contents in an easy-to-follow structure, effectively rendering that layer of security obsolete. In contrast to Chrome however, Firefox also encrypts the username required for login, not only the password, which does help with security [5,6]. The main focus of this paper is the vulnerabilities and attack vectors present due to the metadata present in plaintext in those files. Assuming a malicious entity were to get ahold of any of those databases could derive a variety of information from them, even without going through the process of trying to decipher the passwords. For example, they could compare the hashes of the passwords a user used for websites and see where they used the same password - a bad practise that unfortunately most people still do [6,7,27]. In the case of a data breach in one of the sites, the attacker could simply see to which sites you used the same password and access your account effortlessly, since both the websites URL and username employed (in the case of Chrome) are present in plain-text.

4.6 Threat Model

The existence of this browser extension can help mitigate or prevent certain attacks from happening. As noted above, auto filling hidden fields by a password manager simply because they are aptly named is an unwanted and dangerous functionality of the software – also called a "Field Type Mismatch Attack" [3]. Encrypting the name of the fields it auto fills ensures that should one get the login database; they won't be able to execute such an attack [7]. One would argue that disabling autofill is a valid workaround, however you open yourself to other attack vectors, such as a clickjacking attack. A properly crafted clickjacking attack can have you manually inputting your password in a field you cannot see and is placed just on top of the field you would normally input your credentials – in a sense, "the user's cursor is 'hijacked'", hence the name "clickjacking" [21]. My proposed browser extension tries to merge the best of both worlds, by allowing a user access to both the ease of use and the security features of autofill functionality, while also covering for its shortcomings. Moreover, the outdated encryption algorithms currently still present in browser-based password managers are a cause of concern, as can be easily overpowered by the sheer performance of modern computers, and ingenuity of attackers – a perfectly valid example would be "Firepwd", a GNU v2 licensed python script capable of decrypting the entire Firefox database file nigh instantaneously [12,13,14,15]. Of course, this raises several concerns – anyone with access to the database can simply get ahold of said script and effortlessly access everything in it.

5 Problem Description and Goals

5.1 Problem Description

Given what we described above, it is clear that the present metadata, although helpful for the browser to provide its' services, can open the users to other avenues of attack. For example, both databases contain the names of the fields the browser should autofill when and if it encounters them. Someone with the skill and knowledge to do so, if they know the names of those fields can easily craft a clickjacking attack - as no password manager yet is able to tell the difference between a legitimate field and a malicious one, it would simply fill it and give an attack our credentials. It is a paradoxical approach that a password manager would use advanced algorithms and protocols to protect a password, yet leave all other information associated with it in plaintext for anyone to be able to extort.

5.2 Problem Scope and Goals

The goal of this paper is develop a solution in the form of a browser extension that can help protect the vulnerable metadata generated. An extension for Chrome that ideally encrypts the login credential database upon suspension of the application, and decrypts it back again when Chrome launches again and requires to use and operate it. As the database is locked and inaccessible as long as Chrome is actively running, then encrypting it when it open will help secure that attack vector of someone managing to steal the file, such that they cannot make use of it in any way. I plan to achieve that by making the browser extension to interact directly with the browser itself, in contrast to the websites a user visits like the majority of extensions. It is also noteworthy to mention that while there is plethora of extensions that work on encrypting the information a browser sends, there is barely anything at all that helps protect the information a browser saves. This only justifies and enhances the need for this project, and by its end, we would have created a browser extension that does exactly that, with code that would be easy to translate and import as an extension to other available internet browsers.

6 Approach

6.1 Overview

Firstly, we need to establish the groundwork and find the obvious limitations that may restrict our development phase. Both Chrome's and Firefox's extensions must be written in the base "web languages", namely HTML, CSS, and JavaScript, and may contain other additional files like text files or images - however other programming languages or third party libraries are generally not supported, or must be converted in some way into one of the three mentioned above, something that i describe how it was done in the next section. In addition, both browsers have their own specific protocol that must be followed to make a working extension in the form of a manifest. The manifest contains information about all files that need to be included in the extension bundle, the service workers to be active alongside the scripts that have access to, and a collection of all the permissions that should be granted for it to work, and also act as a protective measure, by blocking the extension in the case of unexpected behavior and causing security issues.[19,20,23,24]

6.2 Tools and Libraries

6.2.1 Overview

The entire development phase and all coding were done on my local machine running Windows 10 on the WebStorm IDE. It offers many tools and functionality to smooth out development, test and debug, and manage the project and its' structure. In addition, it offers excellent integration with GitLab, on which the project is hosted and version controlled, with the link to the repository found in this report's appendices. Testing was done on a virtual machine for precautionary purposes, also running Windows 10, and some third-party libraries mentioned below were used in the extension's development.[26,32,33]

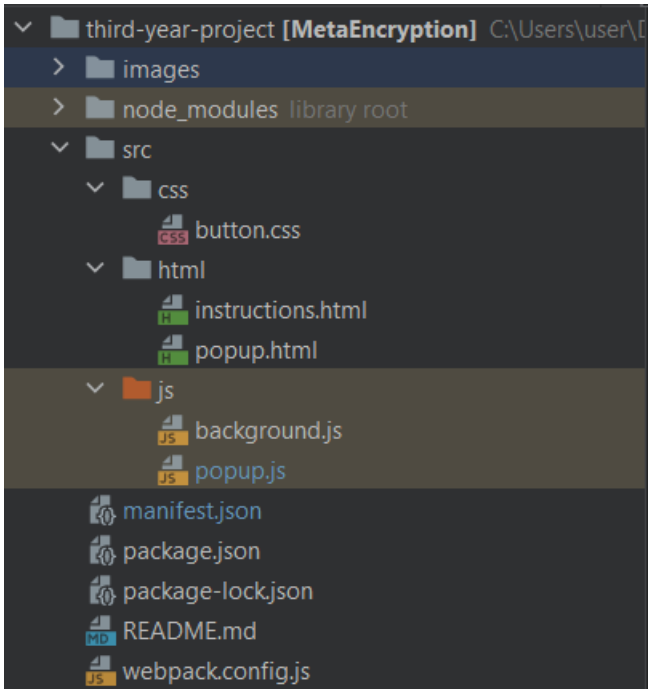

Figure 4: Project File Structure

6.2.2 Javascript, npm and CryptoJS

As mentioned in section 6.1, the languages used were HTML, CSS and the majority of code being JavaScript. To make use of third-party libraries as well as a module bundler, I built the project around and using npm (Node Package Manager) in order to easily install, use and manage said dependencies. One of them would be the most integral part of this extension, namely "Crypto-JS" a cryptography specific library that comes with many widely used algorithms built in, that can be easily implemented into our code. Those include ciphers like DES, 3DES, AES among others, hashing algorithms like SHA-1,2 and 3, along with the abilty to run them in different modes of operation like ECB, CBC, or CTR. As mentioned above however, browsers

do not support anything but pure vanilla javascript, and that is where the Webpack module bundler comes in[23,24,25,32,35].

6.2.3 Webpack and SQL

Webpack is a static module bundler, usable through npm, that allows transformation and compilation of all our code into multiple, or even a single file, that can be read and executed by the browser by analyzing our code and building a dependency graph. It is highly configurable, and all settings can be specified in the "webpack.config.js" file, that is included in the code repository. However for the purposes of our extension, simply defining two entry points, namely for the files "background.js" and "popup.js" and output two seperate files was more than sufficient. Last but not least, in order to monitor the activity and progress, and to also understand in depth the inner workings of both browsers and their databases, we needed SQLite3 in order to access and query the file, and view its' schema. This was also needed to check the file integrity after decryption during development, to ensure that our code does not corrupt the file[26,36].

6.3 VirtualBox Windows 10 VM

As a testing and research environment i made use of Oracle's VirtualBox with a fresh install of Windows 10. This was done for several reasons: for one, i needed to observe how Chrome (my target browser for this project) would behave and react to sudden changes in it's internal file structure, for example, the database being suddenly removed from directory while Chrome is actively running. That would assist in figuring out what functionality is possible to implement in the extension. Secondly, the virtual machine served as a sandbox environment to account for the unlikely chance the extension behaved erratically or the encryption/decryption process corrupted the files (further details regarding testing follow in section 7.1). [33]

6.4 Development phase and Problems Encountered

As this was my first time developing a browser extension, much of the above were not known to me, like third party library incompatibility. This however led me to find Webpack and several other approaches that will definitely help me on the aspect of web development. One other problem that i had when i started development and testing, was the database used to be decrypted into a text file rather than SQLite database file. As the ciphertext was saved as a base64 encoded string, it was saved as a text file, and in decrypting it would result in the same file type. I dealt with that problem by specifing a ".db" extension when decrypting but then this lead me to discover that Chrome would completely ignore the decrypted database. This was due to the fact that by default Chrome appends the ".txt" extension when downloading files to files that do not have any. The reason why we needed a file without an extension in the first place was because I had to match the file name and structure that Chrome expects to find: a *generic file* with the name "Login Data". That problem was solved by specifing the MIME type of the file when creating the URL to download, making it into *"application/octet-stream"*. Yet another problem that caused a design alteration of my extension was auto-generating those files in the exact specific format it needs them if it doesn't detect them, and also auto-syncing in the case of a user logging in with their Google account and syncing their information and settings to the cloud. Chrome would search for the file and if it didn't detect it, it would create a brand new, empty database in its place, ignoring the decrypted one that is accessible. I suspect the reason behind it is the fact that the decrypted database has the ".db" extension, instead of no extension like the original one. To work around this, i had to alter the decryption method to create a base64 encoded string in order to force a generation of a generic file, similar to the one Chrome creates[31,32,20,19].

6.5 Design Decisions

In developing this extension, I had to make several decisions about the structure, the storage of the key and initialization vector, as well as the encryption method to use. As stated earlier, Firefox employs 3DES with a 128-bit key, and Chrome employs AES with yet again a 128-bit key, both running on CBC mode. Our extension encrypts the database using PBKDF2 derived 256-key, using the AES-CBC. This decision was made because of the security this algorithm and key size provide, as well as the efficiency of said algorithm. It is worth noting that while CBC mode is not parallelizable, as one piece of ciphertext depends on the previous one, the size of the database amounts only to several kilobytes, a size small enough for current

computer speeds to process in milliseconds. I decided to utilize the chrome's "storage" API, in order to check for the existence of a key and initialization vector upon installation and start-up of the extension, and their storage upon creation. If any of the two aren't found, then new random ones are generated. To generate an IV I make use of 'cryptojs.Wordarray.random(16)' to get a 16-byte/128-bit random vector. In the case of the encryption key, I first generate a random salt using 'cryptojs.lib.WordArray.random(128/8)', and then use that salt to derive from a predetermined string a 256-bit key using PBKDF2, set to run 1000 iterations, an amount that balances security and performance. The key and IV are then accessed and used for encryption and decryption of the database file through usage of the AES-256 algorithm, running on the default CBC mode[16,29,18,31].

6.5.1 Chrome Extension API

The first step into creating a Chrome extension for cryptography would be to utilise the API given to us by Google. We declare the "manifest" JSON file, giving the entry point for our extension, and declaring all permissions, resources, and service workers it will need to function. This also serves as a safety precaution so in case the extension behaves unexpectedly it does not cause damage or attempt to access resources it shouldn't[20].

```
{
  "name": "MetaEncryption",
  "description": "Protect your login data",
  "version": "1.0",
  "manifest_version": 3,
  "background": {
    "service_worker": "src/js/background-packed.js"
  },
  "permissions": ["storage","activeTab", "scripting", "downloads"],
  "web_accessible_resources": [
    {
      "resources": ["src/html/instructions.html"],
      "matches": ["<all_urls>"]
    }
  ],
  "action": {
    "default_popup": "src/html/popup.html",
    "default_icon": {
      "16": "/images/icon16.png",
      "32": "/images/icon32.png",
      "48": "/images/icon48.png",
      "128": "/images/icon128.png"
    }
  },
  "icons": {
    "16": "/images/icon16.png",
    "32": "/images/icon32.png",
    "48": "/images/icon48.png",
    "128": "/images/icon128.png"
  }
}
```

Figure 5: Snippet of manifest.json file of said extension

In Figure 1, the entirety of our manifest.json file can be seen, but in my opinion the most important parts are the "permissions" and "web_accessible_resources". The permissions declare that:

- *"storage"* allows us to utilise Chrome's internal storage to store things like variables, either locally or sync with their cloud service - in this case we locally store our encryption key and initialisation vector.
- *"activeTab"* allow the extension to work on and/or monitor and change the active tab. Permissions can be set to allow for access to all tabs but they are not necessary, and more permissions than needed should not be used.

- *"scripting"* is probably the most important of the four - allows the extension to execute code on the tabs it has permission to. Here is where the manifest's security capabilities are shown best, as for example, if an attacker injects scripts in an extension that does not have the permission to execute code, they will not be run, protecting the end user.

- *"downloads"* is what gives an extension permission to monitor, start or cancel downloads. We need this permission in order to save the file we encrypt/decrypt. An alternative would be to utilise the "File System Access API" available for web applications, but the built-for-chrome API works just as seamlessly with the browser.

6.5.2 Advanced Encryption Standard - AES

The Advanced Encryption Standard (AES) is a symmetric key block cipher, first published in 1997, succeeded the now deprecated Data Encryption Standard (DES) as the national standard by NIST in 2001. It can operate with 3 different key sizes, 128, 192 and 256 bit keys, and offers more than sufficient complexity and performance to be the algorithm to employ for this project. This cipher is also capable of operating in several different modes, such as Electronic Code Book mode (ECB), Cipher Block Chaining (CBC) or Counter Mode (CTR). The most modern approach would be to run the algorithm in CTR, however implementing CBC would be far simpler and still provides adequate security. The algorithm operates by performing a series of transformations on the block, for between 9 to 11 rounds depending on key length. This transformations include byte substitution utilising an S-Box. Afterwards there are 2 closely linked transformations, namely "Shifting Rows" and "Mixing Columns". The rows are shifted an X number of bytes depending on row number, and columns are mixed by utilising a modulo multiplication in Rijndael Galois Field by a given matrix. These two steps are the primary source of diffusion in the algorithm. Then the final step consists of adding in what is known as the Round Key, a round-unique key derived each time from the user-supplied encryption key[29,16,17].

6.5.3 Cipher Block Chaining - CBC

Cipher Block Chaining splits a message into blocks of 128-bits, XORs the first block with an initialization vector, and then proceeds to encrypt the result to produce the first block of ciphertext. It then XORs that first ciphertext with the next 128-bit block and encrypts the result to get the second block of ciphertext – the process continues and padding is added as needed to the blocks until the whole message is encrypted. For decryption of the ciphertext, we simply follow the steps back to produce the original message, as this is a symmetric key cipher[18].

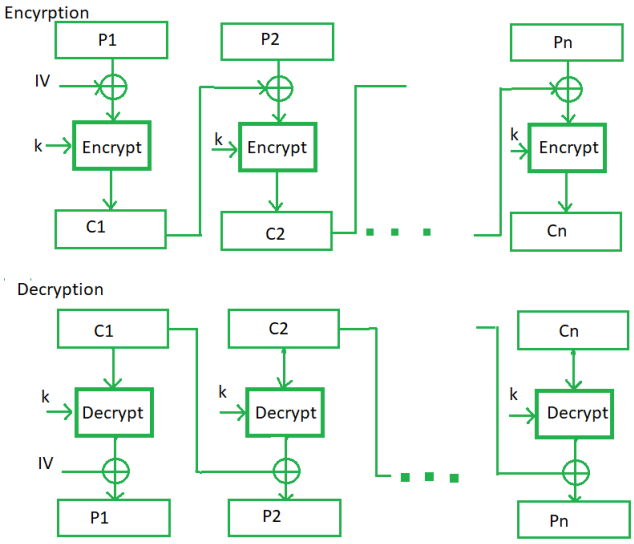

Figure 6: Illustration of how CBC mode works
Where P = Plaintext, C = Ciphertext, IV = Initialisation Vector, k = Encryption Key
source: `https://www.geeksforgeeks.org/block-cipher-modes-of-operation/`

6.5.4 Password Based Key Derivation Function 2 - PBKDF2

The key used through out this procedure is derived with the usage of Password Based Key Derivation Function (PBKDF2). First we need to generate a random salt, and append it to a keyword. Then we continuously hash the combination of the two and their resulting hashes over a predetermined amount of iterations. On occasion, we might also add what is known as "pepper", a single random character, that might be an upper or lower case character, number, or symbol. The pepper is not stored and every combination of the pepper is hashed instead. That is feasible because of the cheap computational cost of the hashing algorithms. Although many are available, our current implementation of PBKDF2 makes use of the Secure Hash Algorithm 1 (SHA-1), with added salt over 1000 iterations[31].

```
function generateKey(p){
    var salt = cryptojs.lib.WordArray.random(128/8);
    return cryptojs.PBKDF2(p, salt, { keySize: 256/32, iterations: 1000 });
};
```

Figure 7: Generating random salt, and then using PBKDF2 to generate a 256-bit key (where parameter 'p' is user-given password

```
vector = cryptojs.lib.WordArray.random(16);
```

Figure 8: Generating a random, 16-byte, initialisation vector

7 Testing and End Product

7.1 End Product Overview

The development phase resulted in a simple yet practical extension that can complete the majority of it's goals (limitations encountered are discussed further in the report)

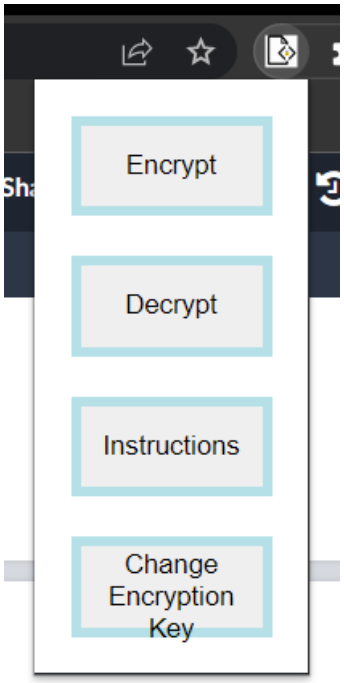

Figure 9: Extension end product

The four buttons present operate as such:

- ***Encrypt*** opens a file selection dialog that allows you to select the Login Data (and "Login Data for Account" if applicable) to encrypt using a 256-bit key derived from either a predetermined default or a user-inputed key, using AES-CBC. Then opens a file save dialog for the user to choose where to download the encrypted file, which must then be placed in Chrome's working directory. Note that because of the file type of the database, the encryption is made to always output a generic file, so encryption of other types will still be possible however the way to use them afterwards will differ.
- ***Decrypt*** works in the same way as encrypt does, by selecting the encrypted file, decrypting it and allowing you to choose where to place the password vault, that must then be placed yet again in Chrome's working directory for the browser to detect and use it.
- ***Instructions*** opens a new tab with a bundled HTML file, namely "instructions.html" that contain all the information mentioned above, a how-to guide, and points where a user must take caution.
- ***Change Encryption Key*** allows a user to input their own password from which to derive a key using PBKDF2 - as it is also mentioned in the Instructions tab, it is most important to do so just after installation of the extension, because by default it comes with a predetermined string to derive a key from - if left as is it introduces an unnecessary weak point by as then keys for multiple users will be derived by the same string.

7.2 Overall Testing of Extension

I conducted some tests on both Chrome's functionality and processes, as well as the capabilities of my extension on a Virtual Machine running Windows 10. That was to account for the case of the extension malfunctioning and corrupting the database, effectively causing loss of data, which while originally empty, was filled with dummy data. For every base functionality of the extension done, there would be functionality testing on the browser by loading it into it "unpacked", and only when ensured that it worked correctly and

consistently would I push the changed into the Git repository. In addition, the assumption and precautions taken for data corruption payed off - between trying to decrypt back to a generic file and trying to read the Base64 encoded data from a text file, several revisions of my code would cause the encrypted file to be corrupted during decryption, effectively becoming unreadable. I caused that to happen by mistakenly attempting to convert to UTF8 encoding from a WordArray, but binary data contain sequences of bytes that do not correspond to characters in UTF8. This however was fixed by setting the MIME type of our data as "application/octet-stream" when forming the blob of data to be formed into a file.

7.3 Current Limitations and Potential Future Improvements

My original idea of this extension was to encrypt and decrypt the database on demand, depending on whether or not Chrome would need access to it at any given moment. I was unable to achieve this due to lack of information on how Chrome recognizes the need to use its password manager features and the associated database. In addition, I am heavily restricted on file system access due to the existing browser protocols in place, so although the process can and does work, it cannot be fully automated. Moreover, the default key used to encrypt the file has random features that make it unique for each user, however a part of it is derived from a string identical to all distributions of the extension (unless a user changes it) - with further research into cryptography that can definitely be changed and improved into something even more random and entirely unique. It is worth noting that no policies are enforced for the user selected encryption password, so there exists a possibility of a weak key being used. Said key is stored using Chrome's "storage" API, in local storage, and not synced to a cloud for obvious reasons. That storage however, as stated by Google, is not encrypted or protected itself. So although, difficult to access, is currently the most vulnerable area of this project, as a successful attack aimed at it may extract the key - potentially implementing the "CryptProtectData" function to protect the key while in storage can safeguard against this, as the function is generally considered very solid for encryption. Moreover, different, better algorithms can be swapped into the extension - for example we can change the 1000 iterations of SHA-1 used in PBKDF2 for a different, more robust hashing algorithm, or change the CBC mode currently used in AES for a more secure mode, like (Galois) Counter Mode, that also offers an authentication code to ensure no one tampered with the database as well as encrypting it.

8 Project Management

The Project followed the timeline presented in my progress report. The first semester of the academic year was dedicated to literature review, identifying the problem and presenting possible solutions for it, while the second one was spent entirely on the production and testing of the browser extension. The Gantt chart that follows shows the seperation of the major tasks that had to be completed, and the chronological order in which they were completed, as well as the amount of time spent in each:

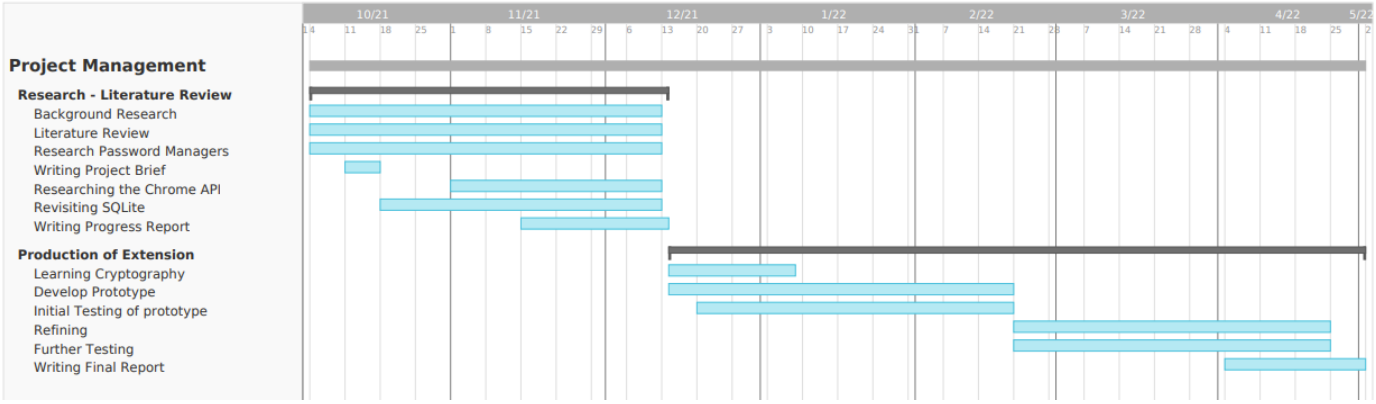

Figure 10: Gantt Chart of Project Management, split into the 2 terms, each one seperated into the various major tasks

9 Conclusion and Suggestions

Overall, this project aimed at protecting the exposed metadata left by browser based password managers, by encrypting the database as a whole. The process couldn't be fully automated due to current security standards and protocols set in place regarding browsers and the internet standards - regardless, the encryption and decryption works as intended and the file is in Chrome-readable format. Although the original vision of this project could not be fully realised, i believe the idea itself and research done sets a basis for further development - possibly this extension can become an integral part of a browser so it can perform to it's fullest extent.

10 References

1. C. Luevanos, J. Elizarraras, K. Hirschi and J. Yeh, "Analysis on the Security and Use of Password Managers," 2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2017, pp. 17-24, doi: 10.1109/PDCAT.2017.00013.
`https://ieeexplore.ieee.org/document/8326801`

2. On the Security of Password Manager Database Formats
Paolo Gasti, Kasper B. Rasmussen, September 2021
`https://link.springer.com/chapter/10.1007/978-3-642-33167-1_44`

3. Fill in the Blanks: Empirical Analysis of the Privacy Threats of Browser Form Autofill
Xu Lin, Panagiotis Ilia, Jason Polakis, November 2020
`https://www.cs.uic.edu/~browser-autofill/`

4. Revisiting Security Vulnerabilities in Commercial Password Managers
Michael Carr, Siamak F. Shahandashti, March 2020
`https://arxiv.org/abs/2003.01985`

5. That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers
Sean Oesch, Scott Ruot, August 2020
`https://www.usenix.org/conference/usenixsecurity20/presentation/oesch`

6. Password Managers: Attacks and Defenses
David Silver, Suman Jana, Dan Boneh, Eric Chen, Collin Jackson, August 2020
`https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/silver`

7. Automated Password Extraction Attack on Modern Password Managers
Raul Gonzalez, Eric Y. Chen, Collin Jackson, September 2013
`https://arxiv.org/abs/1309.1416`

8. Firefox Official Forums – Default Protection of Saved Logins
`https://support.mozilla.org/en-US/questions/1210914`

9. Firefox Source and Documentation
`https://searchfox.org/mozilla-release/source/security/nss/doc/html/pk12util.html`

10. RaiderSec Article – How your Browsers Store your Password (and Why you shouldn't let them)
`http://raidersec.blogspot.com/2013/06/how-browsers-store-your-passwords-and.html#firefox`

11. Firefox Official Forums – Password manager Encryption Type
`https://support.mozilla.org/bm/questions/1249831`

12. Firewpd: A Firefox key.db decryption tool
`https://github.com/lclevy/firepwd`

13. Import and Export passwords for Firefox Quantum
`https://github.com/louisabraham/ffpass`

14. StackExchange - How are Mozilla Firefox passwords encrypted?
`https://security.stackexchange.com/questions/215881/how-are-mozilla-firefox-passwords-encryp`

15. All your browser-saved passwords could belong to us: a security analysis and a cloud-based new design
Rui Zhao, Chuan Yue, February 2013
`https://dl.acm.org/doi/pdf/10.1145/2435349.2435397`

16. Advanced Encryption Standard
`https://telluur.com/utwente/master/SyS%20-%20System%20Security/2018/Aanvullende%20docs/AES.pdf`

17. A Study of Encryption Algorithms (RSA, DES, 3DES and AES) for Information Security
Gurpreet Singh, Supriya, April 2013
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.403.5601&rep=rep1&type=pdf

18. Cypher Block Chaining
https://www.techopedia.com/definition/11162/cipher-block-chaining-cbc

19. Firefox Browser Extensions Guide and Documentation
https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Your_first_WebExtensi

20. Chrome Browser Extensions Guide and Documentation
https://developer.chrome.com/docs/extensions/mv3/getstarted/

21. Clickjacking Attacks
https://owasp.org/www-community/attacks/Clickjacking

22. The Emperor's New Password Manager: Security analysis of Web-based Password Managers
Zhiwei Li, Warren He, Devdatta Akhawe, Dawn Song, August 2014
https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/li_zhiwei

23. JavaScript
https://developer.mozilla.org/en-US/docs/Web/JavaScript

24. JavaScript
https://en.wikipedia.org/wiki/JavaScript

25. LinkedIn Cryptography Courses
https://www.linkedin.com/learning/topics/cryptography

26. SQLite
https://www.sqlite.org/index.html

27. Zxcvbn: Low budget password strength estimation
Daniel Lowe Wheeler, August 2016
https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler

28. An Analysis of Modern Password Manager Security and Usage on Desktop and Mobile Devices
Timothy Oesch, August 2021
https://trace.tennessee.edu/utk_graddiss/6670/

29. Advanced Encryption Standard
https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf
https://www.rfc-editor.org/rfc/rfc3826.html

30. How does Google Chrome store passwords?
https://superuser.com/questions/146742/how-does-google-chrome-store-passwords

31. Password Based Key Derivation Function 2
https://www.ietf.org/rfc/rfc2898.txt

32. CryptoJS Documentation
https://cryptojs.gitbook.io/docs/

33. Oracle VirtualBox
https://www.virtualbox.org/

34. How do password managers work?
https://cybernews.com/best-password-managers/how-do-password-managers-work/

35. Webstorm IDE
https://www.jetbrains.com/webstorm/

36. Webpack
https://webpack.js.org/

11 Appendices

University of Southampton GitLab Repository
`https://git.soton.ac.uk/cp2n19/third-year-project`