# Security Analysis of Browser Auto-fill and Password Managers

Sharon Lin
sharonl@mit.edu

Shendo Maccow
cmaccow@mit.edu

Mayukha Vadari
mvadari@mit.edu

Avital Baral
abaral@mit.edu

May 11, 2020

## 1 Introduction

Auto-fill for passwords and other sensitive information have become a ubiquitous feature of modern browsers. This feature usually prompts the user to save their credentials when they visit a particular site. The browser will then fill in the fields with their credentials when they revisit, saving them time. They also allow for password storage and management, saving the user from having to memorize or otherwise recall the many passwords they use online without needing to repeat passwords. For security-minded people, implementations of this feature intuitively feel dangerous to opt into, given that the ease of a user retrieving their password may translate to an ease for an attacker to retrieve their passwords, yet on the other hand password managers such as LastPass are touted as safe and convenient tools for managing the countless username/password pairs we are prompted to create and maintain online. [1]

This project surveys the status of known vulnerability and security exploits facing password managers for the Chrome and Firefox browsers, as well as password manager applications like LastPass, comparing and contrasting their approach to security with that of the browser auto-fill features. We first explore the official security policies and assumptions underlying the browser auto-fill features in Google Chrome and Mozilla Firefox, followed by collecting and testing a set of known exploits of the Chrome and Firefox browser auto-fill feature and the LastPass password management application. Finally, we'll conclude with a series of observations and recommendations on the use of browser auto-fill features and password managers. [2]

## 2 Platform Overview

### 2.1 Chrome

Chrome's built-in password manager tracks login secrets on all visited sites. When users log into a site, they are prompted to save their credentials for later use. The next time they visit the site, their stored credentials will already be filled in. These credentials can be synced across different devices, as long as the user is logged into their Google account on Chrome. The password manager also checks reputations of new websites that users visit to ensure that they are legitimate, and checks whether your passwords have been involved in any breaches. Users can view their usernames in plaintext, and their passwords in plaintext if they enter their OS password.

### 2.2 Firefox

Firefox has an internal password manager called Lockwise that allows users to store credentials associated with their Mozilla account. If they log in on different machines or download the app and log in, they can access their passwords even if they are not on the browser on their home machine. The application allows them to copy and paste their usernames and passwords in plaintext.

### 2.3 LastPass

LastPass is an application that allows users to store passwords and other confidential information, using a username and master password. For browser applications, under default settings, a user signed into LastPass is able to view their passwords in plaintext in a browser. They can also view their credentials on the LastPass mobile app.

## 3 Security Policy

### 3.1 Objectives

The purpose of this section is to document the use and operation of browser-based auto-fill and password managers by defining the users, roles, access levels, and other policies surrounding the security of the products.

As for the operation of the application, the expected usage is that the user would add credentials for each site they log into. They would then be able to log in again the next time they return without needing to manually recall the username and password for the site. The frequency of the authentication may be user-determined (the fields can be automatically filled or require a master password to access the database with the passwords for filling in the fields), but the application should ideally clarify to the user the risks involved in staying logged in to the master application for extended periods of time.

The user should only have access to their own credentials, and should be the only person with access to their credentials. If another user is using their computer, there should be a way to disable the application and prevent other users from accessing their credentials.

### 3.2 Definitions

A password manager allows users to manage their usernames and passwords for different sites. This can be something that is natively a part of a browser, or a separate application.

Auto-fill is a feature where a password manager or something similar automatically fills in credentials or other stored information when a user is logging in or entering in other information online, such as credit card details or addresses.

### 3.3 User Roles and Access Levels

The access controls of the applications should ensure confidentiality of the user, data integrity of usernames and passwords, and authentication for users to access their data.

For both password managers and auto-fill features, users should be able to view, add, and modify usernames and passwords after authentication. They should not be able to view, add, or modify other users' usernames and passwords or their own usernames and passwords without authentication. The limits of the authentication should ideally be information theoretic, and computational at the very least.

In addition, the company behind the password manager should not be able to view, add or modify any user's passwords, and ideally usernames as well.

### 3.4 Authentication

Both the auto-fill and password manager should be required to authenticate a user before they access any of their usernames or passwords. This includes instances where the browser may be auto-filling a field on behalf of the user; the browser should be required to authenticate the user before revealing their sensitive data onscreen. It should not reveal the information that it is authenticating to any parties.

### 3.5 Information Gathering and Usage

Both apps should not be gathering any information about their users. If any aspects of their security policy are changed, the app should notify all users. Additionally, the app should require that users read and agree to a statement about the usage and gathering of their data.

## 4 Security Analysis

### 4.1 Chrome

#### 4.1.1 Security Policy Assumptions

The security engineers for Chrome made several assumptions about the security policy of their application. In particular, according to Justin Schuh, the "only strong permission boundary for your password storage is the OS user account" [3] [4]. Thus, Chrome uses encrypted storage provided by the OS to protect passwords for a locked account, while assuming that boundaries within the OS user account aren't reliable. Contrasting with other systems, Chrome believes that using a master password would provide users with a false sense of security and encourage risky behavior. [4] Thus, only OS user access protection is used for the primary protection of passwords.

#### 4.1.2 Security Implementation

Internally, Chrome saves the sign-on secrets in an internal database known as 'Web data' in the user profile folder, or 'Login Data' for newer version of Chrome. The SQLite database contains tables storing different data including auto-complete, search keyword, i37logins, and login secrets. Most of the login table contains information about sign-on secrets including website URL, username, and password. Apart from the password, which is stored in encrypted format, the other information is stored in plaintext. [5]

When syncing across the Cloud, Chrome will by default use the Google Account password, but there is the option to use a special syncing passphrase. When syncing locally, Chrome will attempt to use whatever local password vault exists. [6]

For Windows, Chrome uses Windows Data Protection API and the CryptProtectData function, built into Windows, to encrypt passwords. [4] This is a secure function implemented using a triple-DES algorithm, creating user-specific keys to encrypt the data. It can also be decrypted as long as the user is logged into the same account that encrypted it. The CryptUnprotectData is its counterpart function, which decrypts the data. [7]

For Mac and Linux, the encryption scheme is AES-128 CBC with a constant salt and constant iteration. The decryption key is a PBKDF2 key generated with a salt b'saltysalt', with key length 16, an IV of 16 bytes of space. On the Mac, the password is in the Keychain under Chrome Safe Storage - the password can be accessed using the excellent Keyring package or with bash using 'security find-generic-password -w -s "Chrome Safe Storage". On Linux, the password is peanuts and the number of iterations is 1. [5]

**4.1.2.1 Communication with Google** For the purpose of this section, Google refers to the company and Chrome refers to anything Chrome stores locally on the user's machine.

Some information about any form the user fills out, such as the fields and the form structure, is sent to Google, to help Google improve their auto-fill feature. [6]

Google helps users identify when their information has been a part of a data breach. When Google discovers a username-password pair in a data breach, they store a hashed and encrypted copy of the information. When a user signs into a website, Chrome will send to Google a hashed copy of their username and password, encrypting using a key

known only to Chrome and not to Google. To determine if a user's username and password has appeared in a breach, Google uses a technique called private set intersection and blinding, which allows Google to compare the user's encrypted username and password to its collection of encrypted usernames and passwords. Chrome also sends a 3-byte SHA256 prefix of the username to make computation more efficient. [8]

#### 4.1.3 Successful Attacks

For Mac and Linux, we were able to import the Crypto.Cipher.AES module to use for decrypting the ciphertext associated with the file containing the passwords. Since the key is stored in the database file, we are able to pass this key, the IV (16 spaces), and decrypt the CBC Mode AES ciphertext. This does require being logged into the same Keyring account that encrypted the data.

For Windows, we similarly used the win32crypt module, passing in the encrypted ciphertext to decode. This was done using the open-source Lazagne Project. It required no logins, and could even be performed from a different user account on the same computer, provided it is launched with admin privileges. [9]

#### 4.1.4 Unsuccessful Attacks

One attack was unsuccessful because downloading the Executable that supposedly would find the passwords was prohibited by the Windows operation system. This helps validate the security policy that the operating system will protect Chrome.

### 4.2 Firefox

#### 4.2.1 Security Policy Assumptions

It is assumed that a user who loses their master password is compromised. However, we want to see if there are methods for breaching security without use of the master password.

#### 4.2.2 Security Implementation

In the past, Firefox has made several significant changes in the way that their password management system is implemented. Users saving browser passwords without a master password would have been theoretically protected from attacked with access to their computer, if not for the key to the 'logins.json' file being found in the 'key3.db' file. The master password is hashed by adding a salt and applying the SHA-1 algorithm. When the user enters the master password, the software compares the hash with the master password's hash. Apart from the weakness of SHA-1, Firefox only had one iteration of the hash, permitting brute force attacks. [10] [11]

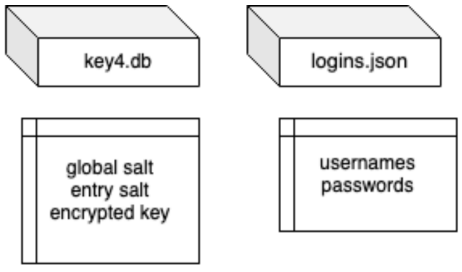


Figure 1: This shows how the databases that are used for storing the global salt, per entry salt, and encrypted key, as well as the encrypted credentials for Firefox.

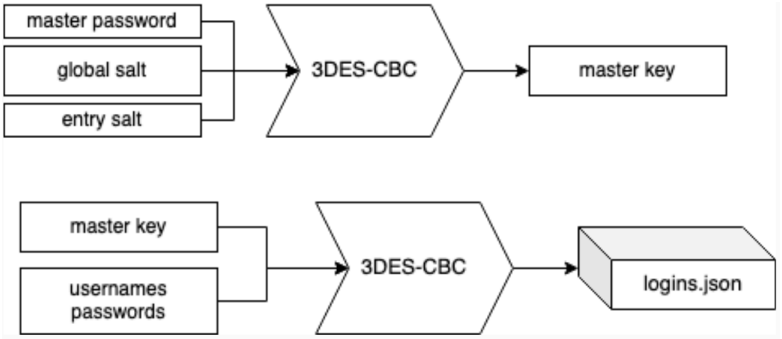


Figure 2: This shows how Firefox obtains a master key from a master password using 3DES-CBC and the process for using the master key to encrypt the login credentials.

Currently, passwords are accessed through the Firefox Lockwise application at 'about:logins' and are viewable in plaintext so long as the user is signed into their master account through the browser. Lockwise doesn't keep the master password, however.

Address, email, and telephone auto-fill data is available through 'auto-fill-profiles.json' located in a separate folder from the Firefox application. Likewise, password data is available at 'logins.json' (encrypted with 3DES) and 'key4.db'. In the key file, the global salt is stored in 'item' and the entry salt and encrypted key in 'item2' (in DER encoding). The master key is encrypted using 3DES-CBC with a key and IV derived from the master password, global salt, and per entry salt. The encryption of usernames and passwords is also done using 3DES-CBC with an IV in the metadata of each item. The best attack on this database would be a brute force on the master password. The password's entropy is bounded to 112 bits by 3DEX, thus making the algorithm weak by modern standards (though it would computationally take several decades to brute-force the key). [12]

#### 4.2.3 Successful Attacks

We made use of the Network Security Services (NSS) modules, an open source software conforming to PKCS#11 standard, providing hash functions, big number calculations, and cryptographic algorithms. [13]

For Mac, we were able to use the NSS library to decrypt the base64 entries in the database, taken from the credentials stored in the Firefox profile in .csv format. We check for a master password, although many profiles are stored without protection from a master password, thus allowing anyone with access to the profile to decrypt the credentials.

For Windows, we were able to use the Python ctypes library for accessing DLLs, including the 'nss3.dll'. We can then use the PK11SDR_Decrypt function for decrypted entries in the passwords.csv database found with the Mozilla profiles. For databases without master passwords set, the decryption can automatically occur without the need for brute force. [14]

For older versions of Firefox, we also used John the Ripper to crack the key3.db master password in the case that the master password was set. The login data is stored in signons.sqlite with base64 encoding, 3DES in CBC mode encryption, and standard block padding. The decryption key is kept in the key3.db file, with entries encrypted by the master password. For verifying the master password, the password check entry is decrypted and compared against the fixed string "check-password\\x00 \\xoo". From the database file, the global-salt (which is stored in plaintext) and the data in the file are passed to john for cracking. [15]

## 4.3 LastPass

### 4.3.1 Security Policy Assumptions

Any user who has access to the username and master password is the owner of the account. LastPass is designed so that any insecurity in either the transfer or storage of data is harmless because all data transferred or stored is encrypted. LastPass also believes that AES-256, PBKDF2, and SHA-256, are secure as their encryption model uses those algorithms. [16]

### 4.3.2 Security Implementation

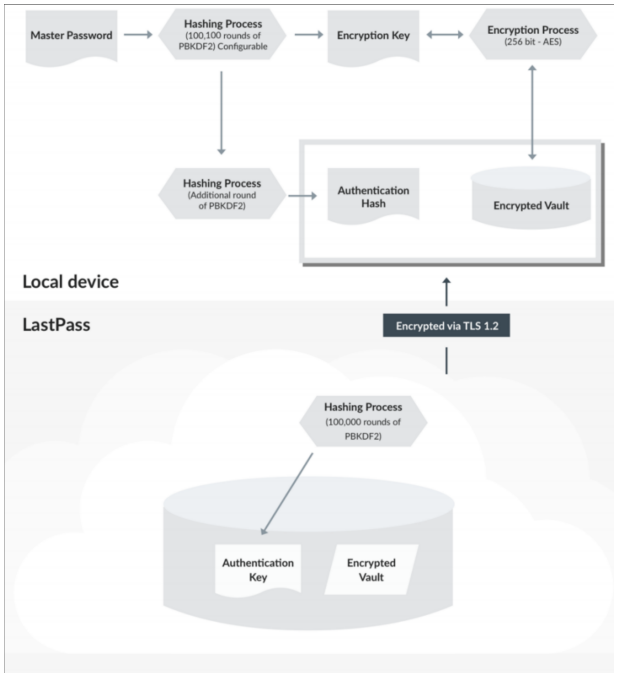


Figure 3: This shows how LastPass encrypts credentials on its platform.[16]

#### 4.3.2.1 Security Policy

Last Pass' promise of security is largely attributed to its local-only encryption model, also known as "host-proof hosting". Local-only encryption means that only the user will be able to decrypt and use the data that they are storing within last pass. This guarantees the security of both data being sent to last pass, and the security of data stored in last pass databases. Because only encrypted data is being sent to last pass, any data that is intercepted in transit to last pass will already be encrypted, and any data that can be stolen from last pass databases will already be encrypted. [16]

#### 4.3.2.2 Encryption Details

LastPass uses PBKDF2 to generate user specific encryption keys. PBKDF2 takes in 5 variables: a hash function, a password, a salt, a number of iterations to run through, and the desired output key length. In LastPass' version SHA-256 is used as the hash function because, although it is a slower hashing algorithm, it provides more protection against brute force attacks. LastPass also defaults to 100,100 iterations, although this can

be changed by the user, and a key length of 256. The Password is the user's master password and the salt is the user's username.

A user's data is encrypted by using the Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode with the encryption key generated from their username and master password. This data is then paired with an authentication hash which is generated by taking the encryption key and running one more round of PBKDF2. The authentication hash and encrypted data are then both sent to last pass via TLS 1.2 in order to mitigate the risks of downgrade attacks and misconfiguration. Content Security Policy headers provide further protection from injection attacks, such as cross-site scripting.

Once the user's data has been received by LastPass, the authentication hash that was sent is then run through an additional 100,100 rounds of PBKDF2 in order to ensure that both the data that is stored by a user locally and the data that is stored online in LastPass servers are protected. If the output after these final rounds of hashing matches the stored authentication hash for the user, then the user's vault unlocked and is updated with the encrypted data sent by the user, or sent to the user if they were requesting access to their vault rather than updating it. [16]

# 5 Recommendations

Based on our survey of the current relative security of these features, we wanted to recommend some changes to the way users interact with the Chrome and Firefox browser auto-fill features.

In general, due to the more secure encryption scheme used in LastPass's platform, their password manager is preferable to relying on either of the browser based auto-fill features. However, using a password manager at all is still preferable to not having any password manager.

## 5.1 Chrome

Although this has already been recommended numerous times, turning auto-complete off is necessary for preventing forms on websites not associated with a particular set of credentials from accessing those credentials. This can very easily lead to phishing attacks and thus should be avoided.

In addition, it is recommended that the Keyring account used for encrypting the Chrome passwords is not always logged on, and that it is protected with a strong password to prevent brute force attacks from accessing the credentials stored on the Keyring.

## 5.2 Firefox

When storing passwords on Firefox, it is recommended that the user uses a Master Password to provide an additional layer of security to the stored credentials.

## 5.3 LastPass

When storing passwords on LastPass, it is recommended to turn on the option requiring LastPass to prompt the user with the master password every time the user wishes to use credentials. This prevents someone from obtaining the user's usernames and passwords even if they obtain access to the machine, and also helps the user remember their master password better.

# 6 Conclusion

In this project, we examined the security policies and guarantees of the Chrome and Firefox browser password auto-fill features (for the latest stable versions of Chrome and Firefox, namely 81.0.4044 for Chrome and 75.0 for Firefox at the time of writing) as well as those of the password manager LastPass, comparing and contrasting the security assumptions made by the developers of these password management systems and the implementations of the systems based on these assumptions. We tested these systems on Mac (version 10.14.5), Linux (Ubuntu 20.04 LTS), and Windows 10 operating systems.

While Chrome's development team considers OS security the only actual strong permission boundary for password storage, they have still implemented password encryption, making use of the Keychain and the Windows Data Protection API for Mac/Linux and Windows. Meanwhile, Firefox, considers its master password the strong permission boundary for password storage and LastPass likewise utilizes its username and master password to assert the authenticity of the owner of an account.

We catalogued some successful as well as unsuccessful breaches of the Chrome and Firefox features on Mac, Linux, and Windows operating systems. With Chrome, we were able to use an encryption and decryption module to decrypt the ciphertext associated with the file containing the passwords. With Firefox, we were similarly about to use the NSS modules present on all three operating systems to decrypt the master key and login data, as well as use John the Ripper to crack weaker master passwords.

Finally, we issue recommendations for users of all three applications, specifically recommending that auto-complete features be disabled, a master password used for securing passwords if the option is available, and passwords managers be used if one is not already securing their credentials in an encrypted manager. We highly recommend that users use LastPass over the browser-based password managers, as LastPass has a much more secure encryption scheme.

# 7 Acknowledgements

We would like to thank the Prof. Ron Rivest and Prof. Yael Kalai, as well as the 6.857 TAs Adrian, Christos, and Jacob for their help in refining the scope and topic of the project.

# References

[1] M. Pinola, "Which Password Manager Is The Most Secure?," *LifeHacker*, 2012.

[2] B. Strine, "Is saving passwords in Chrome as safe as using LastPass if you leave it signed in?," *Stack Exchange*, 2013.

[3] J. Yarow, "Google Has A Major Security Flaw In Chrome That Gives People Easy Access To Your Passwords," *Business Insider*, 2013.

[4] tylerl, "Is saving passwords in Chrome as safe as using LastPass if you leave it signed in?," *Stack Exchange*, 2017.

[5] P. Chheda, "Chrome Password Grabber," *GitHub*, 2019.

[6] "Google Chrome Privacy Whitepaper," *Google Chrome*, 2020.

[7] darkArp, "Chromepass - Hacking Chrome Saved Passwords," *GitHub*, 2019.

[8] P. e. a. Nepper, "Better password protections in Chrome - How it works," *Google Security Blog*, 2019.

[9] AlessandroZ, "The LaZagne Project," *GitHub*, 2020.

[10] A. Hersean, "How are Mozilla Firefox passwords encrypted?," *Stack Exchange*, 2019.

[11] R. Alves, "Firefox Decrypt," *GitHub*, 2019.

[12] L. Abraham, "ffpass - Import and Export passwords for Firefox Quantum," *GitHub*, 2019.

[13] "An Overview of NSS Internals," *MDN Web Docs*, 2019.

[14] A. Sharma, "Firefox Passwords," *GitHub*, 2019.

[15] Skactor, "John the Ripper," *GitHub*, 2019.

[16] LastPass, "LastPass Technical Whitepaper," *LastPass Enterprise*, 2018.