Persistence of Passwords in Bitwarden's Browser Extension: Unnecessary Retention and Solutions

Rafael Alexandre da Silva Prates

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Alexandra Sofia Ferreira Mendes

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa

June 2022

FCT
Fundação
para a Ciência
e a Tecnologia

Acknowledgments

I would like to thank my parents for their encouragement, caring and support over all these years, for always being there for me. I would also like to thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years. I would like to thank my godmother as well, for being a big pillar of support and help during the times where I felt down and could not keep going.

To my friends, every single one of you has played a part in this, not only in helping me grow as a person but also supporting me and being there for me in the good and bad times of life. For going above and beyond to help me purely because you cared and wanted me to succeed.

I would also like to acknowledge my dissertation supervisors Prof. João F. Ferreira and Prof. Alexandra Mendes for their insight, support and sharing of knowledge that has made this Thesis possible.

To each and every one of you – Thank you.

This work was partially funded by the PassCert project, a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019.

Abstract

Password-based authentication is still the dominant form of authentication on the web, yet users do not adopt password managers for fear of them being insecure, unreliable and other reasons. In this project we modify a password manager to try to comply with certain data security properties as a way to increase adoption of this type of software that has been increasing in importance.

Taking BitWarden's Google Chrome extension as our chosen password manager, we define password manager states and data security properties regarding the master password that we would like to comply with, perform tests and analyse password retention problems in the application. While the BitWarden extension interacts with many layers, we decided to only change the application layer, as a way to understand how much can be done by the developers of these types of applications.

We then introduce our modified extensions that try to solve the issues presented before and introduce a testing framework that is able to automatically interact with the extension through the graphical user interface to replicate the use case chosen. While our solution does not completely solve the issue, we were able to reduce the problem slightly.

Keywords

Password Managers; Password retention; Data security; BitWarden; Google Chrome

Resumo

A autenticação usando senhas ainda é a principal forma de autenticação na internet, porém existem utilizadores que não adoptam gestores de senhas por medo de que sejam inseguros, desconfiáveis, entre outros. Neste projeto, nós modificámos um gestor de senhas para que conforme com algumas propriedades de segurança definidas, para tentar aumentar a adoção deste tipo de software que se tem tornado cada vez mais relevante com o passar do tempo.

Usando a extensão do Bitwarden para o Google Chrome como o nosso gestor de senhas, nós definimos estados do gestor de senhas e propriedades de segurança de dados relacionados à senha mestre que gostaríamos de satisfazer, corremos testes e analisámos problemas de retenção de senhas na aplicação. Embora a extensão BitWarden interaja com muitas camadas, nós decidimos modificar apenas a camada da aplicação, para compreender o quanto pode ser feito pelos programadores deste tipo de software.

Seguidamente, apresentamos as nossas extensões modificadas que tentam resolver os problemas discutidos anteriormente e introduzimos uma estrutura de testes que é capaz de interagir com a extensão automaticamente através da interface gráfica do utilizador para replicar um caso de uso escolhido. Embora a nossa solução não resolva o problema completamente, fomos capazes de reduzi-lo.

Palavras Chave

Gestor de senhas; Retenção de senhas; Segurança de dados; BitWarden; Google Chrome

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

PM	Password Manager
RAM	Random Access Memory
MP	Master Password
GC	Garbage Collector
OS	Operating System
DOM	Document Object Model
JS	JavaScript
Chrome	Google Chrome
TOS	Terms of Service
GUI	Graphical User Interface
PV	Password Vault

1

Introduction

Contents

1.1 Introduction

The need for more passwords has been increasing over the years as password-based authentication is still the dominant form of authentication on the web [1] and as users sign up to more and more services, they require more and more passwords. Unfortunately, because the number of passwords a typical user needs to remember is increasing, users tend to reuse passwords across multiple services or small variations of the same one and reuse previously leaked passwords [2]. Many users tend to follow these practices to avoid the cognitive burden of recalling different passwords [3, 4] especially when passwords that are difficult for an attacker to guess are also hard to memorize for the user. This is compounded by the fact that users see the high amount of effort required to get just a low percentage of passwords in data breaches and think the ease of use of password reuse outweighs the risk.

Password Managers (PMs) come as a solution to this problem by inheriting the responsibility of remembering passwords from the user to the software. This in turn allows the user to associate stronger passwords, that are hard to memorize, but even harder to attack, to the services they use boosting the user's security and avoiding the unsafe practices of password reuse, all while offering usability features such as automatic strong password generation resilient and auto-completion of log in forms to name a few. PMs also help protect the user against data breaches. By using a strong unique password for each service, even if one service gets its data breached the rest will be safe since there is no password reuse. This also lowers the amount of work a user has to do whenever a data breach occurs, since it is easier to change one password (the one that was breached), than to change all passwords if the user reused the same password or variations of it on all their services. And while having strong unique passwords is not exclusive to using a PM, it certainly makes it easier to do so.

And this is important as the number of data breaches are on the rise [5], leaving many users vulnerable to attacks.

However, PMs are not immune to attacks and a portion of potential users do not feel that using a PM would provide greater security [6] for their secrets, some think they are insecure [7–10] and some of them outright distrust [8, 9] PMs as a whole. Because of these reasons and others, these users refuse to adopt PMs.

In this project, we decided to focus our attention to a particular type of attack that can be used to steal sensitive data from a computer's memory. These attacks involve making a copy of the device's Random Access Memory (RAM) either through a cold boot attack [11] or through vulnerabilities that might exist such as HeartBleed [12], MeltDown [13], Specter [14], etc...

In the context of a PM application, it is expected that secrets, such as passwords or cryptographic material derived from passwords, will have to be in the memory of the process at some point in time throughout the use of the application. If an attacker is able to extract the memory at the right time, they will be able to extract and steal that information. As such, it is critical that these secrets are deleted from memory as soon as they are no longer necessary. Although there are different types of sensitive data an application can hold, we will only focus on user passwords.

Our study will take BitWarden [15] as our PM, more specifically the Google Chrome (Chrome)'s [16] extension of BitWarden as the case of study. We will be focusing on logging into the PM with a Master Password (MP) and by dumping the memory of the BitWarden extension process, analyse when the MP is in memory as well as when it should not be according to some desired data security properties. We then implement solutions at the application level to try and eliminate this secret in memory when it is no longer needed.

This project is part of the PassCert research project,[1] a CMU-Portugal exploratory project that will build an open-source, proof-of-concept password manager that through the use of formal verification, is guaranteed to satisfy properties on data storage and password generation.

1.2 Organization of the Document

This thesis is organized as follows:

Chapter 2 contains the background work, desired memory security properties, BitWarden and its' login process and password retention risks. In chapter 3, we explain our threat model, perform tests on the BitWarden Chrome extension, analyse the memory dumps created, compile a list of results and observations as well as a list of problems with the extension's code. In chapter 4, we introduce our modified extensions and explain their changes compared to the original BitWarden extension. In chapter 5 we present our testing and analysis framework and compare the results between the original BitWarden extension and our modified extensions. And finally, we present our conclusions and future work in chapter 6.

[1] PassCert is a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019

2

Background work

Contents

This chapter presents the relevant background work needed to understand the following chapters. We will be introducing what a PM is in general, desired data security properites of a PM, BitWarden's login process, the different layers of abstraction between the application code and the machine and insecure code practices that can retain passwords longer than necessary.

2.1 Password Managers

2.1.1 What are they?

The main feature of PMs consists in storing username information and the correspondent password into what is known as the Password Vault (PV). The PV is then responsible for storing the user's information in an encrypted fashion to ensure that the user's secrets are kept hidden from attackers.

Since there are various PMs available online, some offer more features such as, password generation, auto-completion of login forms, cloud storage of the PV, cross-device synchronization and the list goes on but in the end the basic concept remains the same. The PV may also store other relevant information like website data, namely URL and icons to name a few, creation date, fill count and other. This information will be referred to as metadata since it is not vital for logging into an account.

2.1.2 How are passwords stored

Password storage is done by creating a password vault, either on the local system or on the cloud. When a PV is stored locally, they are usually an encrypted file residing on the local system's disk. When stored on the cloud, it also encrypted in some fashion, with the details differing from provider to provider. The algorithms used to encrypt a PV file differs from PM to PM and most require a MP to reverse the encryption and be able to access it.

2.1.3 What is stored on the password vault?

A password vault contains entries. There are as many entries as the user wants or needs, but the regular behaviour is having an entry for each different account. Each entry stores the necessary information for logging into the account with the minimum usually being the username and password.

In addition, it is not unusual to store metadata along with the entry on the password vault. This includes but is not limited to:

1. Website metadata such as URL, Icon and name.

2. Password metadata such as creation time, modification time, last use time, fill count and expiration date.

3. User settings namely notes and autofill settings.

Unfortunately not every PM guarantees encryption of all metadata. For example, KeePassX and KeePassXC both encrypt all metadata [17]. Extension-based password managers encrypt most metadata, but all have at least one item they do not. Browser-based managers that rely on the operating system to encrypt their vaults protect the relevant metadata too. Those that do not rely on the operating system have a significant amount of unencrypted metadata.

2.1.4 How are password vaults encrypted?

In general, most app-based and extension-based encrypt their vaults using a MP. Requirements for the MP vary between applications with some not requiring one at all. Browser-based systems (except Firefox) rely on the operating system instead to help them encrypt the password vault and as such implementations vary.

2.2 Desired Data Security Properties of a Password Manager

Since this project is done in the context of the PassCert project, BitWarden was chosen as the base PM. The PassCert project is an effort to create a proof-of-concept PM that through the use of formal verification, guarantees properties on data storage and password generation [18]. which is the BitWarden extension for chrome.

With the basic function of a PM in mind, we will define 2 states in which the application is running: (a) not running; (b) locked (pre-login); (c) unlocked (and running); and (d) locked (session terminated) [19] .

We will not analyse the PM in its not running state (when Chrome is not opened or the extension is disabled) as we have decided to focus on the security of the MP in the login process of BitWarden.

2.2.1 Locked (pre-login)

We define BitWarden to be in the ”locked (session terminated)” when the vault has not been unlocked in the current session. In this state we concede that the MP is stored in the memory of the program and is visible to potential attackers as the application need the MP to perform the tasks necessary to authenticate the user.

2.2.2 Unlocked (and running)

We define the BitWarden PM to be in the ”Unlocked (and running)” state once the BitWarden vault is unlocked. To reach this state, the user must successfully authenticate by entering a valid e-mail and MP.

In this state, the MP is no longer necessary to be in memory as we will discuss in section 2.3.1 and as such, should no longer be in memory.

2.2.3 Locked (and running)

We define the BitWarden PM to be in the ”Locked (and running)” state when (a) the vault is locked manually; or (b) when the session is terminated. For this project however, we only considered (b) in our testing. Continuing from the unlocked (and running) state, the MP should not be in memory as well.

2.3 BitWarden

BitWarden is a PM available for different platforms, desktop, mobile, browser extensions and even online. As expected from a PM, it manages account information like usernames and passwords in a vault.

2.3.1 BitWarden’s Login Authentication

Figure 2.1 is an overview of how BitWarden’s login authentication works. When the user provides the e-mail address and MP, BitWarden uses Password-Based Key Derivation Function 2 (PBKDF2) [20] with a 100,000 iteration rounds to stretch the MP, using the e-mail address as salt. The generated value is a 256 bit Master Key. This Master Key is used in conjugation with the MP as salt to create the Master Password Hash which is sent to the BitWarden server upon account creation and login, and used to authenticate the user account. Once the Master Key and the Master Password Hash have been generated, the MP is no longer required to be in memory, as the application has everything it needs to authenticate the user.

BitWarden mentions that in their client application they do not store the MP locally or in memory and that they do their best to ensure that any data that may be in the application to function is only held in memory for as long as needed [21]. Ideally, the BitWarden extension for Chrome would follow these principles.

2.4 Security issues of keeping secrets in memory for longer than necessary

Keeping secrets longer than necessary is dangerous, as it is quite possible for passwords to remain in memory even after an hour after the program was terminated [22]. An attacker that can obtain a memory dump of the computer could potentially obtain secrets this way, use them to breach the user’s vault, steal all the information in it and proceed to breach accounts for other services that were stored in the vault.

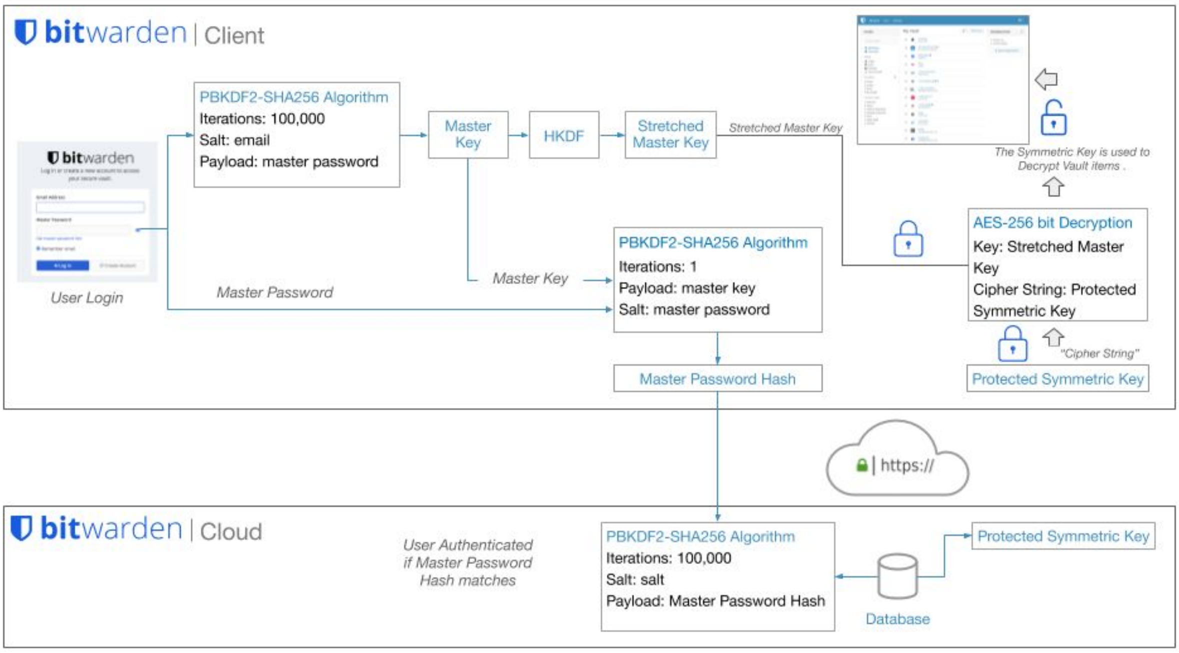

Figure 2.1: Control flow of logging into BitWarden

2.5 Layers of abstraction

The application we are studying runs on top of several other layers of abstraction. Namely the application is written in Typescript with the Angular framework. This Typescript code is then transpiled to JavaScript (JS) using Angular's transpiler. The JS resulted from this transpilation is then interpreted by browser (in our case, Chrome) using the browser's built-in JS interpreter. In Chrome's case, it uses the V8 JS engine [23] and the Blink [24] rendering engine. These two communicate with each other to interpret and display the page. Since Chrome is written in C++, it interfaces with the standard C++ runtime libraries and any other libraries necessary by Chrome's code. Lastly, Chrome also has to interface with the Operating System (OS).

Our project focuses only the application layer as it would be unfeasible to modify everything in the pipeline.

2.6 Password retention risks

2.6.1 Application sent to background

Modern OS's implement Virtual Memory which, amongst other things, can allow for secondary memory to act as main memory. This allows applications to swap memory in RAM to the disk when system

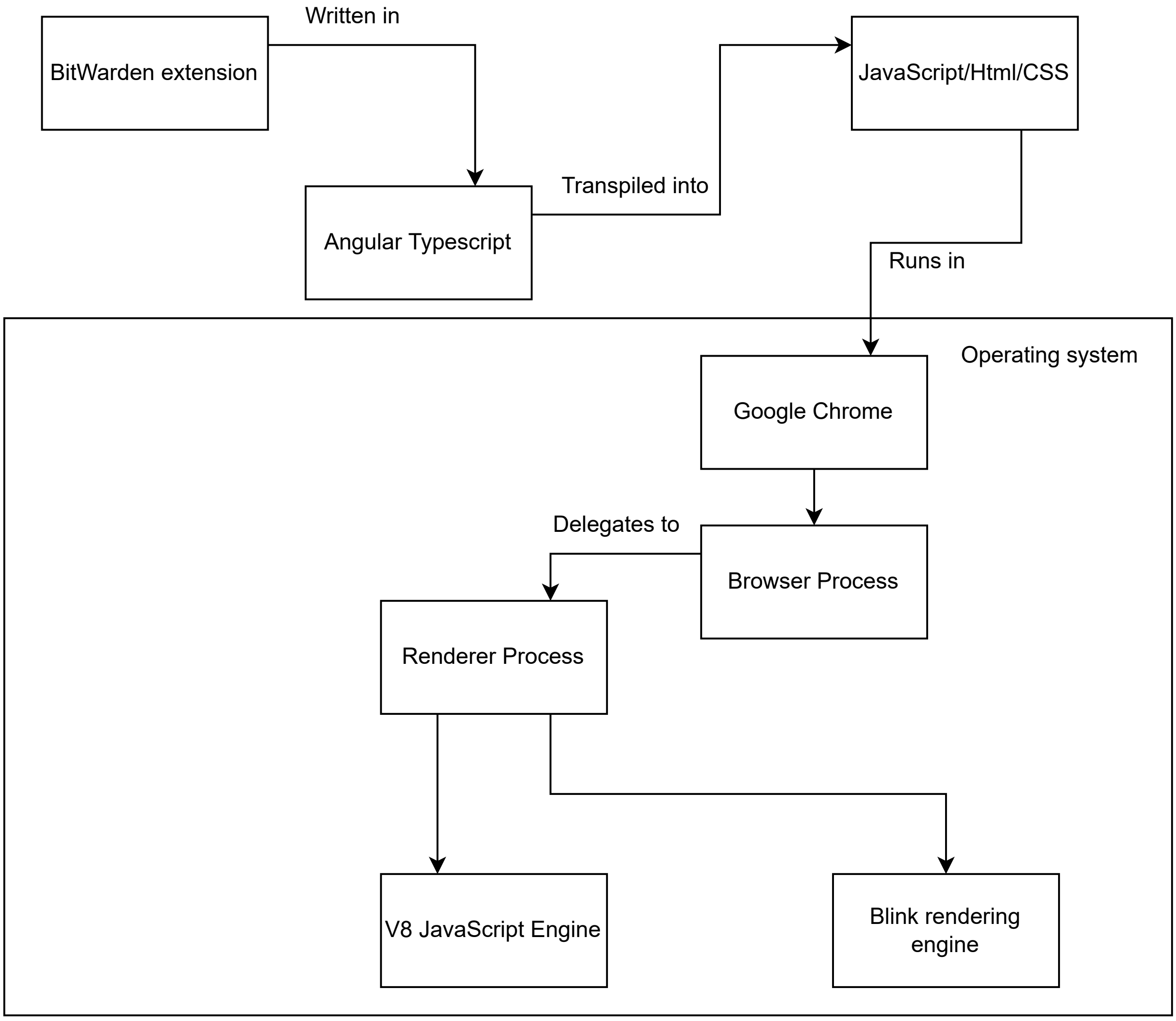

Figure 2.2: The different layers of abstraction

resources are low. This poses a significant risk as a secret could be the memory of a process when its memory gets written into the disk, prolonging the amount of time a secret is exposed. An attacker could then steal the secondary memory storage medium (a solid-state drive, hard-disk drive, etc...) and analyse the paging files to extract secrets that were written to the medium through virtual memory.

2.6.2 Crash dumps

If the computer crashes while the application contains a secret in its memory those secrets could be written to the crash dump file, exposing them. For example, Windows [25] and Ubuntu [26] (if the kernel crash dump utility is installed) dump the contents of RAM at the point of crash into a file.

2.6.3 Delayed garbage collection

The memory of JS applications are managed by a Garbage Collector (GC). When a piece of memory is no longer referenced by any variable it will remain in memory until the GC decides to reuse it. The amount of time that it takes until that piece is reused by the GC is undefined. It can be minutes or even hours [22], depending on the algorithm used and the system's resources and workload. This comes into play when we take immutable types in JS into consideration. The primitive type String in JS is immutable and thus can not be overwritten manually by the developers, leaving the deletion up to the GC when it eventually reuses the piece of memory that contained the String object and overwrites it with some other value. This is a known problem when using immutable data types for sensitive information, such that even Java's Cryptographic Architecture recommends using mutable data structure types [27] for passwords and secret data.

2.6.4 Different layers of abstraction and lack of secure API

Since the BitWarden extension runs on top of other layers, any communication between these layers has the possibility of creating buffer copies of sensitive data and having that data retained for a period of time, outside of the control of the application itself. Without any options of a secure API between layers, the application developers have no control over how the layers that the application interacts with treat the data.

2.6.5 Function Arguments Copies

Whenever a primitive type is passed to a JS function, a new copy of it is made [28]. This presents a problem because if a secret is passed to a function using a String data type, then a new immutable copy of the secret is created hence increasing the amount of copies of it in memory. Numerous copies of the secrets in memory means that it is harder for the GC to delete and reuse those memory pieces in a timely manner, as discussed in subsection 2.6.3.

3

Password retention on the BitWarden extension

Contents

3.1 Threat model

Our work assumes an attacker has access to a snapshot of the memory of the computer through a memory disclosure attack or through a memory dump of the system. This could be from a crash dump file, like we discussed in section 2.6.2 or through a cold boot attack [11].

3.2 Memory Dump Analysis

We performed several tests on the BitWarden extension for the Chrome browser by dumping the memory of the Chrome process responsible for running the BitWarden Extension. These memory dumps were done several times at different points in time, throughout the use of the extension.

3.2.1 Test steps

The points of time in which we perform a memory dump will be referred to as a ”test step” in the testing procedure. As such, the following is an overview when the process’ memory is dumped:

- **Step 0** - After typing the e-mail in the e-mail field, but before typing the MP into the password box;
- **Step 1** - After half the MP was typed into the password box;
- **Step 2** - After the MP was fulled typed;
- **Step 3** - After logging in and unlocking the BitWarden Vault;
- **Step 4** - After simulating a task;
- **Step 5** - After terminating the BitWarden session.

A more in-depth look at the testing procedure will be presented in chapter 5.

3.3 Results and observations

In the memory dumps, we refer to both a partial MP and a full MP. A full MP is the set of characters that compose the entirety of the MP, whether they be encoded in 8-bit (UTF-8/ASCII) or 16-bit (UTF-16) encoding. Given a full MP p of length i, a partial MP is defined by the subset $p_j, j \in \{\frac{i}{2}, ..., i-1\}$. This applies to both UTF-8/ASCII encodings and UTF-16 encoding.

In fig. 3.1, we can see the results of the testing procedure performed on the original BitWarden extension. In Step 0, there are no occurrences of the MP in memory. This is to be expected, since nothing of MP has been typed yet. Also, the memory given to the process is zeroed out by Linux’s virtual

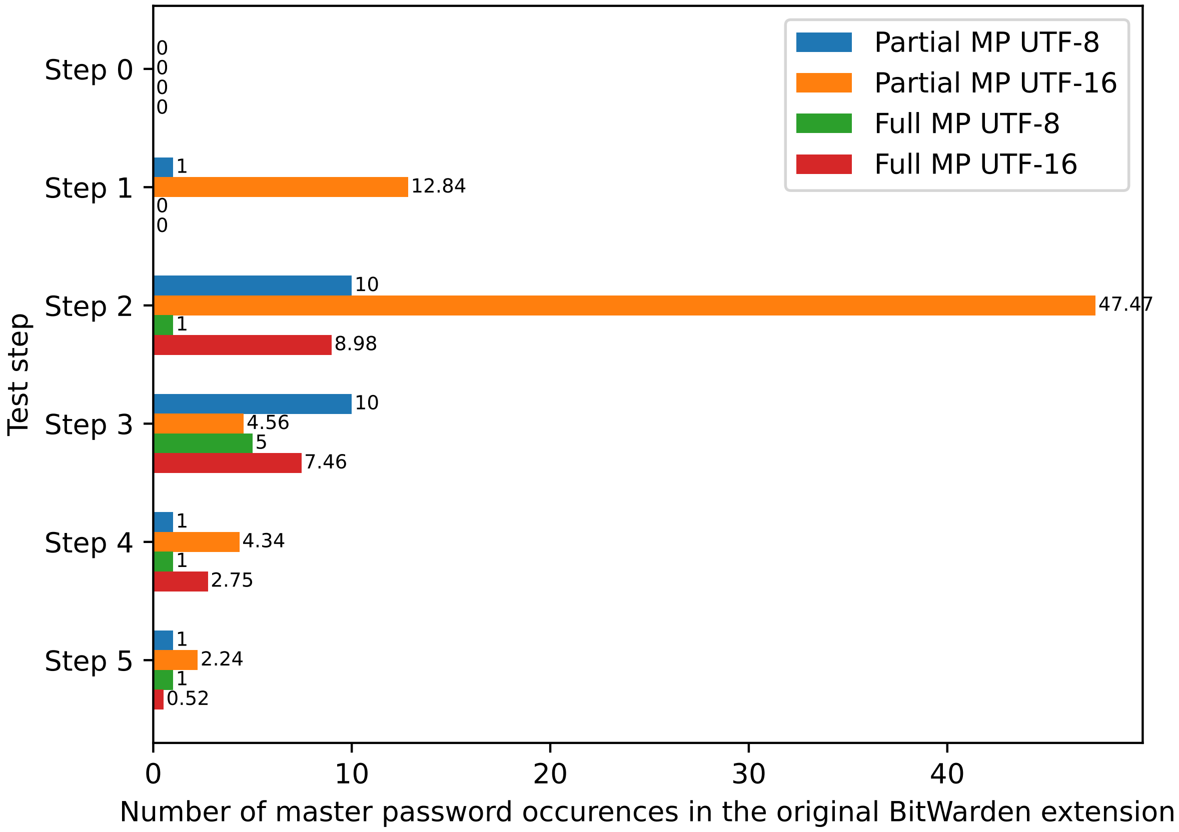

Figure 3.1: Number of partial and full master password occurrences in memory in the original BitWarden extension by test step

memory manager before making it available [29] to the process. This means that even if the process was given an address of memory that previously stored the MP from a previous test, it would not interfere with the results of the current test, since the memory give would be wiped beforehand. In step 1 and 2, as the user is typing the MP we see the number of occurrences rise. We mainly see occurrences of the partial and full MP encoded in UTF-16. This is because Chrome's JS engine, the V8 JavaScript engine, implements the ECMAScript standard which states that the primitive string type is to be encoded in UTF-16 [30].

As mentioned in section 2.2, it would be ideal if after the vault is unlocked (step 3), the MP would no longer be accessible in memory. However, our findings show that is not the case. There are still occurrences of the full MP and occurrences of the partial MP after logging in and unlocking the vault. Even after simulating a simple task, with the intention of increasing the system's load and resource usage to promote memory clean up by the GC, not all occurrences were cleared. Terminating the session and logging out decreases the occurrences but does not completely erase everything. This means the MP can be obtained from memory, for an indefinite amount of time as we mentioned in section 2.6.3, even after the the user closes the BitWarden vault.

3.3.1 Observation #1: Prefixes of the Master Password in Memory

The underlying data structure responsible for storing the MP as it is being typed, is an immutable string. Since we can not modify immutable data types, a new object has to be created. So given a password p with length i, when a new character c is added a new string object is created with the result of $p_i + c$. Likewise, when a character is removed, a new object is created with the value of p_{i-1}. The previous memory addresses are no longer referred to and cleanup is left up to the GC system. As shown in fig. 3.1, we were able to identify memory addresses that contained prefixes of the full MP.

3.3.2 Observation #2: Prefixes stay longer in memory if the input is left untouched

As discussed in subsection 3.3.1, new string objects are created whenever the user types or deletes characters while typing the MP. In our testing, we discovered that if the MP is typed relatively quickly without much delay between keystrokes (around a second or so), the amount of prefixes in memory would decrease compared to when the user types a portion of the MP, leaves it untouched for more than a second or two and then continues typing the rest of the MP.

3.3.3 Observation #3: Unlocking the vault does not clear the master password

After unlocking the vault, the MP is no longer required to be in memory as mentioned in section 2.3.1, but continues to be, even after the user locks the vault and terminates the session.

3.4 List of problems

While analysing the BitWarden's extension source code of the login process we compiled a list of problems within it. Lee et al. [31] found similar issues in Android applications and were able to eliminate leftover passwords from memory by solving them.

3.4.1 Problem #1: Use of immutable data types to hold the master password

In the login component of BitWarden, the MP is stored in an immutable String. Like we mentioned in section 2.6.3, application has no way to erase the content of that String, leaving the deletion of the secret up to the GC.

Listing 3.1: How the master password is stored in the login component of BitWarden

```
masterPassword: string = '';
```

3.4.2 Problem #2: No zeroization of the master password

Due to the problem mentioned above, BitWarden does not go through the effort to try and zero out the content of the variable that holds the MP.

3.4.3 Problem #3: Use of String to communicate master password from interface to component

In an Angular application, the application can access the information in a native element of the Document Object Model (DOM) through a Control Value Accessor [32]. In our case, BitWarden interacts with the input field responsible for the MP in the login page to receive the MP written by the user into the login component to perform the login process. Unfortunately, Angular only provides the DefaultValueAccessor [33] in input fields of type text. This accessor provides whatever is written into the input field to the application, as a String type.

Since we decided for this project to focus only on changes that could be done at the application layer, we have to use what Angular provides. However, in section 4.2 we present how we tried to mitigate this problem.

4

Our solution

Contents

In this section, we will describe the fixes we implemented at an application-level in the BitWarden Chrome Extension to try and reduce the amount of MP copies found after logging into the BitWarden Vault.

Two versions of the program were created with one fundamental difference: one uses a separate Angular component that deals with the input of the MP and communicates with the login parent component, while the other version has the login component deal with the MP input directly, without using a separate Angular component. This way, we can compare if the Angular communication between components is a factor that could lead to password leakage in our program. Each of these versions also has a variation where we inlined a function in a critical area to see how it would affect the amount of occurrences of the MP in the memory of the process.

This gives us four different versions in total: (a) Child component; (b) Child component - inlined; (c) No child component; and (d) No child component - inlined.

4.1 Common changes between all versions

The login component performs a few steps with the MP to complete the login process. First, it checks if the MP is not empty and then delegates the login to the authentication service, passing the e-mail and MP of the user. The authentication service then passes the MP and other information to the crypto service for (a) making the Master Key (makeKey); and (b) making the Master Password Hash (hashPassword). Once the Master Password Hash is complete, it uses it and the e-mail to perform the login and the login component receives a form response and proceeds with the rest of the login process.

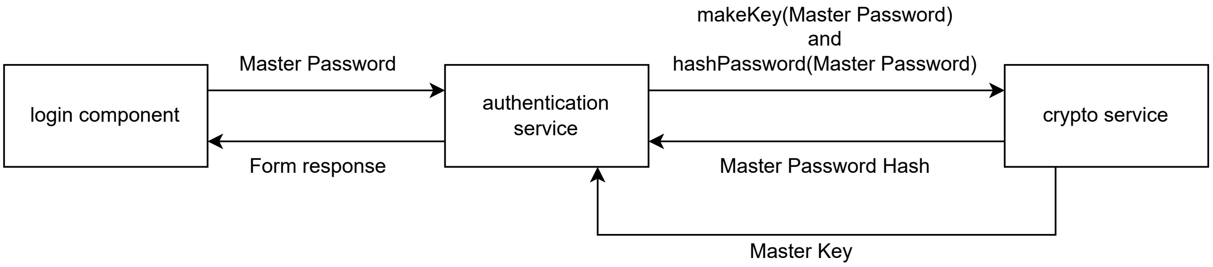

Figure 4.1: Flowchart of how the master password is used in the login component

The first change we implemented was to change the immutable variable type of the MP (string) in the LoginComponent to a mutable data structure.

Listing 4.1: The original masterPassword variable in the login component

```
    export class LoginComponent {
```

```
    masterPassword: string = '';
}
```

Listing 4.2: Our modified masterPassword variable, changed to an ArrayBuffer

```
export class LoginComponent {
masterPasswordBuffer : ArrayBuffer;
}
```

In our case, we decided to use a JS ArrayBuffer [34] for a few reasons. First of all, it is a mutable data type fixing the problems specified in Problem #1 (section 3.4.1) and giving us an opportunity to address Problem #2 (section 3.4.2) as well. Secondly, BitWarden's login code flow makes a few calls to services of the application with the MP, those being the authentication and cryptographic service as discussed above. All of these function calls already have support for the ArrayBuffer data structure. This allowed us to change the function parameters of the functions calls to accept an ArrayBuffer data type instead of the previously used string. Since we are no longer passing a primitive JS type (string) to several functions, but instead a reference type (ArrayBuffer), JS will instead pass the argument (in this case the MP) by sharing (call by sharing). This avoids making extra copies of the MP when it is passed down to a function. This addresses Problem #3 discussed in section 3.4.3.

Listing 4.3: The original login function of the authentication service

```
logIn: (email: string, masterPassword: string) => Promise<AuthResult>;
```

Listing 4.4: Our modified version of the login function

```
logInWithArrayBuffer: (email: string, masterPassword: ArrayBuffer) =>
    Promise<AuthResult>;
```

We replicated these changes in the crypto service as well.

Listing 4.5: The original crypto functions

```
makeKey: (password: string, salt: string, kdf: KdfType, kdfIterations:
    number) => Promise<SymmetricCryptoKey>;
    hashPassword: (password: string, key: SymmetricCryptoKey, hashPurpose?:
        HashPurpose) => Promise<string>;
```

Listing 4.6: Our modified version of the crypto functions

```
    makeKeyWithArrayBuffer: (masterPassword: ArrayBuffer, email: string,
        kdf: KdfType, kdfIterations: number) => Promise<SymmetricCryptoKey>;
    hashPasswordWithArrayBuffer: (masterPassword: ArrayBuffer, key:
        SymmetricCryptoKey, hashPurpose?: HashPurpose) => Promise<string>;
```

After BitWarden performs the login process, Problem #2 is addressed as the MP is no longer required to be in memory. To do so, the ArrayBuffer containing the MP is manually overwritten before the variable is set to null to help it being marked for garbage collection. Because JS's memory management is automatic it will either be reused throughout the execution of the program or freed up and used elsewhere, like another process running on the computer. This process might take a while, but since we cleared the buffer previously, the MP contained is no longer there.

Listing 4.7: Login component: clearing the master password buffer after the login has been completed

```
    (...)
    this.formPromise = this.authService.logInWithArrayBuffer(this.email,
        this.masterPasswordBuffer);
    const response = await this.formPromise;

    let masterPasswordBufferView = new Uint8Array(this.masterPasswordBuffer)
        ;
    this.clearArrayBuffer(masterPasswordBufferView);
```

The next change we had to implement was due to how Angular/JavaScript reads the MP from the DOM. The two different approaches (both the one using a child component and one without using one) are explained below.

4.2 Implementation with a login child component - Child component

To try and receive the MP in a more secure manner, we devised a new component responsible solely on bridging the communication between the native input element in the DOM and the login Angular Form. To do this, we implemented Angular's Control Value Accessor interface on the component to change the default behaviour. Later on, the login component of Angular receives the ArrayBuffer with the full MP from this child component and proceeds to perform the login.

4.2.1 MasterPasswordCustomInputComponent

This component is responsible for listening to input changes from the password input field coming from the DOM. Our approach consists of receiving the input through a function, and storing it in an ArrayBuffer. We also take the care to zero out the data of the previous ArrayBuffer, which contained the previous password input.

Listing 4.8: The function responsible for receiving the input and storing it in our modified master password buffer

```
onInsideChange(receivedString: string) {

let inputArrayView = new Uint8Array(this.modifiedinput);
inputArrayView.fill(1);

this.modifiedinput = Utils.fromUtf8ToArray(receivedString).buffer;
this.onChange(this.modifiedinput);
}
```

Unfortunately, the application is forced to receive the input from the DOM as a string, as it is the only text format that Angular supports. Like we said previously, this creates a copy of the input and it is also immutable, leaving the deletion up to the GC. To mitigate further damage, we do not store the input in the function and only use it when we absolutely must, to transform the input information to the ArrayBuffer.

After the user types the entire MP and clicks the login button, the child component sends the ArrayBuffer with the full MP to the login component. The login component then performs the login process as usual.

Listing 4.9: The child component submitting the full MP to the login component

```
  submitToParent() {
    this.sendPasswordToParentEvent.emit(this.modifiedinput);
  }
```

4.3 Implementation with a login child component - inlined

In listing 4.8, we use a function *(Utils.fromUtf8ToArray)* to convert the string input to an ArrayBuffer. This means we are passing a string into the function which creates an additional copy of the argument as discussed in section 2.6.5. Since this function is called every time the input field changes (whenever a new character is added or deleted), this could lead to potentially even more copies of the input being created.

We made an extension, called the child component inlined, where we inlined the *Utils.fromUtf8ToArray* function to see if it would make a significant difference in the amount of password leakage. The change is described in listing 4.10.

Listing 4.10: The onInsideChange function with the Utils.fromUtf8ToArray function inlined

```
  onInsideChange(receivedString: string) {

    let inputArrayView = new Uint8Array(this.modifiedinput);
    inputArrayView.fill(1);

    if (Utils.isNode || Utils.isNativeScript) {
      this.modifiedinput = new Uint8Array(Buffer.from(receivedString,
        'utf8')).buffer;
    } else {
     const strUtf8 = unescape(encodeURIComponent(receivedString));
     const arr = new Uint8Array(strUtf8.length);
     for (let i = 0; i < strUtf8.length; i++) {
       arr[i] = strUtf8.charCodeAt(i);
     }
     this.modifiedinput = arr.buffer;
    }

    this.onChange(this.modifiedinput);
    }
```

4.4 Implementation without a login child component

As a way to understand if the communication between Angular components could produce extra copies of the MP in memory, we also devised a version without an extra component, where the login component deals with the input of the MP. In this version, the Control Value Accessor interface was directly implemented in the login component.

Listing 4.11: Implementation of the Control Value Accessor in the login component

```
export class LoginComponent {

(...)
    onInsideChange(receivedString: string) {

    let inputArrayView = new Uint8Array(this.modifiedinput);
    inputArrayView.fill(1);

    this.modifiedinput = Utils.fromUtf8ToArray(receivedString).buffer;
    this.onChange(this.modifiedinput);
    }

(...)
}
```

4.5 Implementation without a login child component - inlined

Similarly to what we did in the child component inlined extension (listing 4.10), we simply inlined the *Utils.fromUtf8ToArray* function.

5

Evaluation

Contents

5.1 Preparing the testing environment

A Vagrantbox was created that runs Linux Ubuntu containing Chrome, the official BitWarden extension for Chrome, our modified extensions, all the required packages for running the Python scripts and all the required packages to create a local BitWarden server. Additionally, it also sets up the local BitWarden server and starts it in preparation for the testing procedure. We originally performed tests using BitWarden's servers, however, once we automated the testing procedure we started performing the tests on our own locally hosted BitWarden server for several reasons: (a) the traffic and usage patterns performed in the testing procedure violate BitWarden's Terms of Service (TOS); (b) logging into BitWarden does not require captchas to be solved when the server is locally hosted which simplifies our testing procedure; (c) no need to create a BitWarden account in BitWarden's hosted servers; and (d) the testing procedure is no longer dependent on the availability of BitWarden's services.

Table 5.1 shows the relevant software and versions used in our testing.

Name	Version
Linux Ubuntu	20.04.4 LTS
Oracle VM VirtualBox	6.1.32r149290
Google Chrome	9.0.4844.51-1
BitWarden Google Chrome Extension	1.55.0, 8 Dec 2021
PyAutoGui	0.9.53

Table 5.1: Versions of the software used in the testing procedure

5.2 Automatic testing using Python and PyAutoGUI

To check the differences in password leakage between the normal BitWarden extension and our modified extensions, we devised a script using Python and PyAutoGui [35] to automatically perform a testing procedure to replicate a use case of the application. Automating the testing procedure made testing the different extensions easier and made the tests across the different extensions more consistent with each other, giving us more confidence over the results. We also have the advantage of being able to perform a large number of tests per extension this way, giving us more statistical relevance. Lastly, replicating how a user interacts with the Graphical User Interface (GUI) of the extension made the test closer to in behaviour to how a user perform a task in a use case in the BitWarden extension, as well as replicating any type of memory pattern that might happen from such behaviour, leading us to more accurate results.

The script uses PyAutoGui to perform several GUI interactions with the OS, Chrome and the BitWarden extensions (both the official one and our modified ones). The following is a summary of what the program does:

- Resets Chrome's settings to a default known state

- Opens Chrome and points the BitWarden extension to use the locally hosted server
- Closes Chrome (this is to avoid having multiple processes with the BitWarden name which breaks our script)
- Opens Chrome again
- Clicks on the extensions icon and then the BitWarden extension
- Clicks on the log in button and types the e-mail in the e-mail field
- **Test step 0:** Performs a memory dump before writing the MP in the login page
- **Test step 1:** Types half of the MP and performs a memory dump
- **Test step 2:** Types the second half of the MP (now complete) and performs a memory dump
- **Test step 3:** Unlocks the vault and performs a memory dump once it is open
- To simulate a task, the script opens up a new tab, and plays a video on Vimeo [36] for around a minute
- **Test step 4:** After a minute has passed, it performs a memory dump again
- Clicks on the extensions icon and then the BitWarden extension
- Goes to the settings tab
- **Test step 5:** Finally, it terminates the session on the BitWarden extension, performs a memory dump and proceeds to close Chrome

5.3 Analysing the created memory dumps

To facilitate the analysis of the memory dumps, a Python script was also made, responsible for going through all the memory dumps created at the different steps of the testing procedure. It opens the memory dumps in binary mode and simply reads the memory into the program and scans the memory dump for four different things:

1. The first half of the MP in 8-bit encoding (UTF-8/ASCII) and 16-bit encoding (UTF-16)
2. The full MP in 8-bit encoding and 16-bit encoding as well

To calculate the number of partial MP occurrences, we count the amount of times the first half of the MP has appeared in memory and subtract it with the number of times the full MP was in memory as well, since the partial MP is a prefix of the full MP.

It then creates a CSV file with the results, ordered by test number. The file contains the number of occurrences of the partial and full MP, in both encodings, at the different steps of the testing procedure.

5.4 Results

What test step corresponds to which phase of the testing procedure is in section 5.2. The total number of occurrences of the full MP was calculated by summing the occurrences of the full MP in both UTF-8/ASCII encodings and UTF-16 encoding. Likewise, the same process was done for calculating the total number of partial MP occurrences, by summing the partial MP occurrences in both encodings described previously.

The raw analysis of the memory dumps obtained from the tests of all of our extensions are presented in appendix B.

5.4.1 Full master password

The results of our extensions against the original BitWarden extension are shown in fig. 5.1.

In test step 0 and 1, the full MP is not in memory, as it has not been fully typed yet. In test step 2, when we finish typing the full MP, our extensions have more copies of the full MP in memory. However as we said in section 2.2, we concede that the MP can be in memory while the user has not performed the login process. In test step 3, when we perform the login and unlock the vault, our extensions are able to reduce the copies present in memory when compared to the original extension. We see this trend in later steps for the other extensions as well, except in test step 4 and 5, where the no child component extension is slightly worse when compared to the original one.

The child component inlined extension is the one that performed the best right after step 3, having less full MP occurrences than the rest, but performed slightly worse than others in step 4 and 5.

5.4.2 Partial master password

The results of our extensions against the original BitWarden extension are shown in fig. 5.2

In test step 0, we see no references as the MP has not been typed yet. In test step 1, when the first half of the MP has been typed, we see similar results to the original BitWarden extensions. The rest of the test steps almost mimic the results obtained in section 5.4.1. We have an increase in occurrences in step 2 but after logging in and unlocking the vault in step 3, we see a reduction of copies in memory over

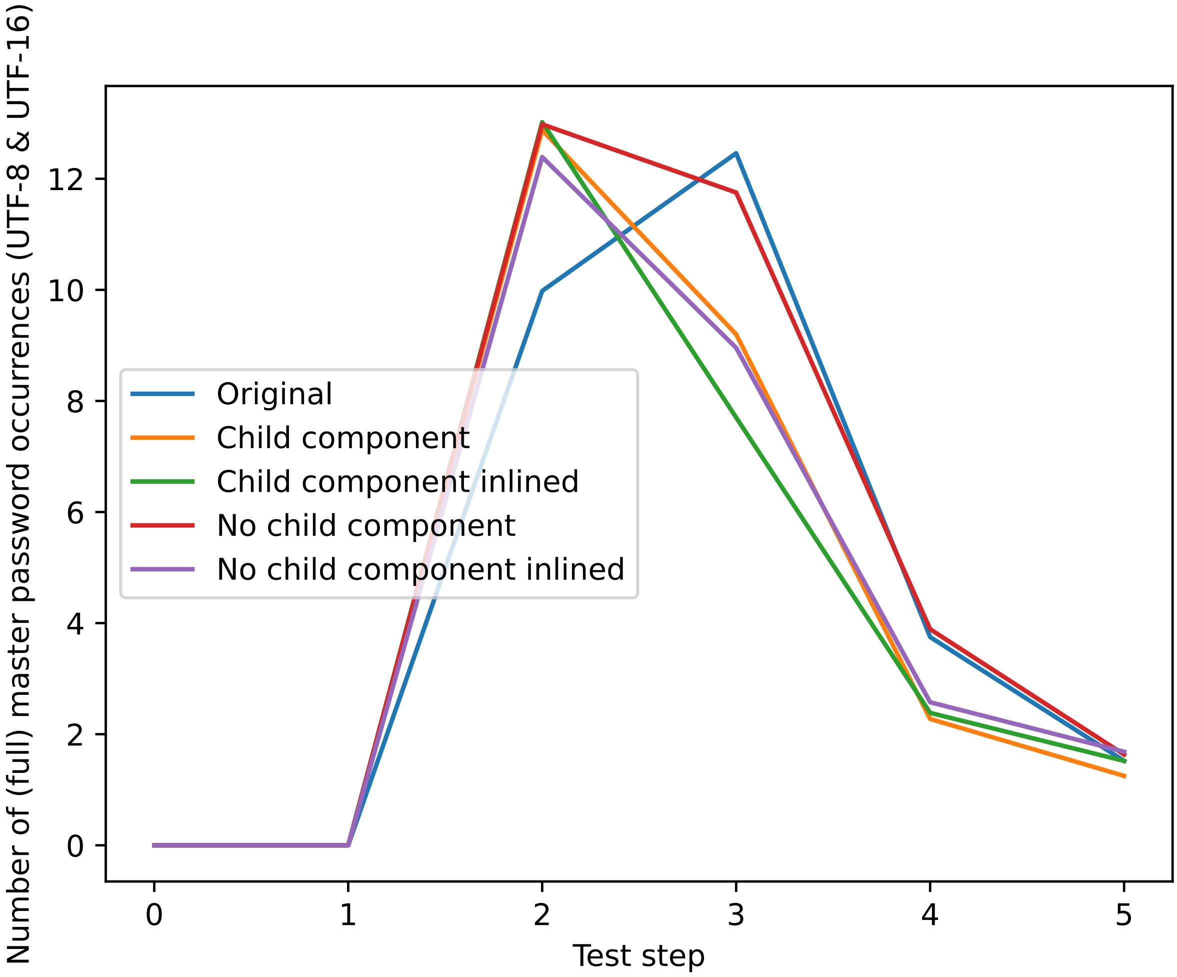

Figure 5.1: The number of occurrences of the full MP in memory per test step

all extensions, except the no child component extension. It stands to note that the no child component extension performs worse when compared to the original BitWarden extensions across the board, when it comes to the partial MP.

The child component and child component inlined extensions performed very similarly, with the child component inlined extension being slightly better in steps 4 and 5 compared to the child component extension.

5.4.3 Child component and no child component

Our no child component extension compares worse than the rest of our extensions and even BitWarden's original extension in some cases. We find this behaviour odd, as the results between the no child inlined component (where the difference between it and the no child component is simply a function being

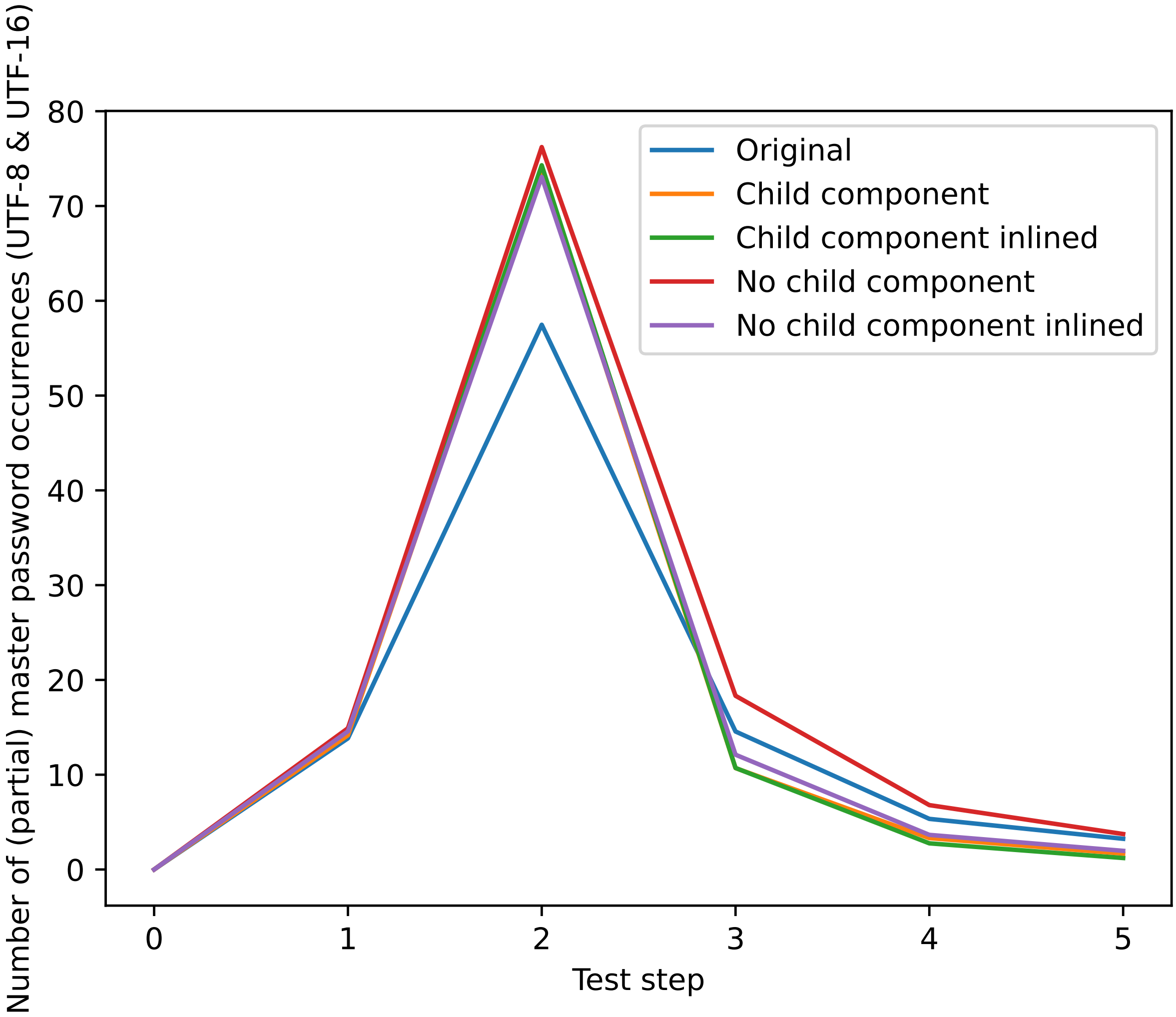

Figure 5.2: The number of occurrences of the partial MP in memory per test step

inlined) and the child component extensions (both the normal and inlined version) are much similar between each other. Without access to external tools, it is hard to say what causes this discrepancy.

6

Conclusion

Contents

6.1 Conclusion

In conclusion, even though our modified extensions were able to reduce the occurrences of the full MP in memory after logging in and unlocking the vault, they were unable to completely remove every trace of the MP, even after performing tasks on the system. As such, an attacker that gains a snapshot of memory after the vault was unlocked, is likely to able to successfully retrieve the MP of the user.

This shows that changes in the application layer are not enough to completely eliminate leftover MPs references in the memory of the process, and efforts must be made in every step of the stack to ensure that sensitive data is dealt with and properly disposed of to ensure the desired data security properties.

6.2 Future work

Modifying the Angular framework to create a new, more secure way to bridge communication from the native DOM to application would be something to work on. On the browser side, while Chrome is closed-source, there are open-source alternatives (like Chromium [37] and Firefox [38]) that could be changed to introduce a secure API to deal with sensitive data on the DOM so communication with web applications could be more secure. Finally, one could change the OS in which the application is running, to ensure that communication between the OS and the other layers would follow the same secure API protocols that we desire.

Bibliography

[1] J. Bonneau, C. Herley, P. C. Van Oorschot, and F. Stajano, “The quest to replace passwords: A framework for comparative evaluation of web authentication schemes,” in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 553–567.

[2] C. Wang, S. T. Jan, H. Hu, D. Bossart, and G. Wang, “The next domino to fall: Empirical analysis of user passwords across online services,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 196–203.

[3] M. Dell’Amico, P. Michiardi, and Y. Roudier, “Password strength: An empirical analysis,” in *2010 Proceedings IEEE INFOCOM*. IEEE, 2010, pp. 1–9.

[4] S. Riley, “Password security: What users know and what they actually do,” *Usability News*, vol. 8, no. 1, pp. 2833–2836, 2006.

[5] P. Fasulo, Aug 2018. [Online]. Available: https://securityscorecard.com/blog/cybersecurity-data-breaches-statistics-on-the-rise

[6] S. Chiasson, P. C. van Oorschot, and R. Biddle, “A usability study and critique of two password managers.” in *USENIX Security Symposium*, vol. 15, 2006, pp. 1–16.

[7] “65% of people don’t trust password managers despite 60% experiencing a data breach,” Jul 2020. [Online]. Available: https://www.passwordmanager.com/password-manager-trust-survey/

[8] N. Alkaldi and K. Renaud, “Why do people adopt, or reject, smartphone password managers?” 2016.

[9] E. Stobert and R. Biddle, “A password manager that doesn’t remember passwords,” in *Proceedings of the 2014 New Security Paradigms Workshop*, 2014, pp. 39–52.

[10] S. Aurigemma, T. Mattson, and L. Leonard, “So much promise, so little use: What is stopping home end-users from using password manager applications?” 2017.

[11] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[12] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.

[13] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[14] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[15] "Bitwarden open source password manager." [Online]. Available: https://bitwarden.com/

[16] "Google chrome." [Online]. Available: https://www.google.com/chrome/

[17] S. Oesch and S. Ruoti, "That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers," in *Proc. of USENIX Security Symp*, 2020.

[18] M. Grilo, J. F. Ferreira, and J. B. Almeida, "Towards formal verification of password generation algorithms used in password managers," *arXiv preprint arXiv:2106.03626*, 2021.

[19] "Password managers' secrets management: Ise," Oct 2020. [Online]. Available: https://www.ise.io/casestudies/password-manager-hacking/

[20] B. Kaliski, 2000. [Online]. Available: https://tools.ietf.org/html/rfc2898

[21] "Bitwarden security whitepaper." [Online]. Available: https://bitwarden.com/help/bitwarden-security-white-paper/

[22] S. Karayianni, V. Katos, and C. K. Georgiadis, "A framework for password harvesting from volatile memory," *International Journal of Electronic Security and Digital Forensics 7*, vol. 4, no. 2-3, pp. 154–163, 2012.

[23] "V8 javascript engine." [Online]. Available: https://v8.dev/

[24] "Blink (rendering engine)." [Online]. Available: https://www.chromium.org/blink/

[25] "Generate a kernel or complete crash dump - windows." [Online]. Available: https://docs.microsoft.com/en-us/windows/client-management/generate-kernel-or-complete-crash-dump

[26] “Kernel crash dump.” [Online]. Available: https://ubuntu.com/server/docs/kernel-crash-dump

[27] “Java ™ cryptography architecture (jca) reference guide.” [Online]. Available: https://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html#PBEEx

[28] “Functions - javascript.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions

[29] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[30] “Ecmascript 2023 language specification.” [Online]. Available: https://tc39.es/ecma262/#sec-literals-string-literals

[31] J. Lee, A. Chen, and D. S. Wallach, “Total recall: Persistence of passwords in android.” in *NDSS*, 2019.

[32] “Control value accessor angular.” [Online]. Available: https://angular.io/api/forms/ControlValueAccessor

[33] “Defaultvalueaccessor angular.” [Online]. Available: https://angular.io/api/forms/DefaultValueAccessor

[34] “Arraybuffer - javascript.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer

[35] “Pyautogui’s documentation.” [Online]. Available: https://pyautogui.readthedocs.io/en/latest/

[36] “Vimeo.” [Online]. Available: https://vimeo.com/

[37] “The chromium project.” [Online]. Available: https://www.chromium.org/chromium-projects/

[38] “Firefox browser.” [Online]. Available: https://www.mozilla.org/

A

Code of Project

In the following appendix we present the full source code for the testing procedure script, which performs the testing procedure on the environment we set up, and the memory dump analyser script, which analyses the memory dumps created by the previous script and creates a CSV files with the results.

Listing A.1: The source code of the script that performs the testing procedure

```
import logging
import os
import pyautogui
import time
import sys
import psutil
import re
import configparser

'''
```

```
Dependencies (so far)
python
pyautogui
pillow
opencv
psutil
'''

#region Global Variables
#Stages of mem dumps
MEMDUMP_BEFORE_TYPING = '0-before-typing-MP'
MEMDUMP_MID_TYPING_MP = '1-mid-typing-MP'
MEMDUMP_FINISH_TYPING_MP = '2-finish-typing-MP'
MEMDUMP_ON_UNLOCK = '3-on-unlock'
MEMDUMP_ON_TASK_FINISHED = '4-on-task-finished'
MEMDUMP_SESSION_TERMINATED = '5-session-terminated'

#File locations for the Icons
COMMAND_PROMPT = "/home/vagrant/passcert/memdump-tests/icons/Command_Prompt.png"
EXTENSIONS_BUTTON = "/home/vagrant/passcert/memdump-tests/icons/Extensions_Icon.png"
BITWARDEN_BUTTON = "/home/vagrant/passcert/memdump-tests/icons/BitWarden_Icon.png"
LOGIN_BUTTON = "/home/vagrant/passcert/memdump-tests/icons/Log_In_Button.png"
E_MAIL_TEXT = "/home/vagrant/passcert/memdump-tests/icons/E-mail_Text.png"
GOOGLE = "/home/vagrant/passcert/memdump-tests/icons/Google.png"
PLAY_BUTTON = "/home/vagrant/passcert/memdump-tests/icons/Play_Button.png"
BITWARDEN_BLUE_BUTTON = "/home/vagrant/passcert/memdump-tests/icons/BitWarden_Icon_Logged_In.png"
OPTIONS_BUTTON = "/home/vagrant/passcert/memdump-tests/icons/Options.png"
YES_BUTTON = "/home/vagrant/passcert/memdump-tests/icons/Yes_Button.png"
BITWARDEN_ENV_SETTINGS = "/home/vagrant/passcert/memdump-tests/icons/BitWarden_Env_Settings_Button.png"

#Task time
TASK_TIME = 60
#endregion
```

```python

#region Functions
def pause(secs_to_pause=1):
    """Pauses the script for secs_to_pause seconds.
    """

    time.sleep(secs_to_pause)

def getExtensionName(extensionDir):
    head,_ = os.path.split(extensionDir)
    _, extensionName = os.path.split(head)

    return extensionName

def memdump(pid, nthTest, iteration, dumpSaveLocation, extensionName):
    # maps contains the mapping of memory of a specific project
    map_file = f"/proc/{pid}/maps"
    mem_file = f"/proc/{pid}/mem"

    # output directory
    out_dir = os.path.join(dumpSaveLocation, extensionName)
    # output file
    out_file = os.path.join(out_dir, f'{nthTest}-{iteration}.dump')
    #Make sure the directory exists, and if not create it
    os.makedirs(out_dir, exist_ok=True)

    logging.info('Starting mem dump on PID %d...', pid)
    # iterate over regions
    with open(map_file, 'r') as map_f, open(mem_file, 'rb', 0) as mem_f, open(out_file, 'wb') as out_f:
        for line in map_f.readlines():  # for each mapped region
            m = re.match(r'([0-9A-Fa-f]+)-([0-9A-Fa-f]+) ([-r])', line)
            if m.group(3) == 'r':  # readable region
                start = int(m.group(1), 16)
                end = int(m.group(2), 16)
                mem_f.seek(start)  # seek to region start
                #print(hex(start), '-', hex(end))
                try:
                    chunk = mem_f.read(end - start)  # read region contents
                    out_f.write(chunk)  # dump contents to standard output
                except OSError:
                    print(hex(start), '-', hex(end), '[error,skipped]', file=sys.stderr)
                    continue
    logging.info('Memory dump saved to %s', out_file)

def findImage(imageFile):
    """Scans the screen to find a section equal to the imageFile. Returns a box with the position of the match in screen coordinates if successful,
    None otherwise.

    The function matches a section of the screen if 90% of its' pixels match the given imageFile.
    """

    try:
        match = pyautogui.locateOnScreen(imageFile, confidence=0.9)

        if not match:
            logging.info("Image %s not found.", os.path.basename(imageFile))
            return match
        else:
            logging.info('Image %s found at x=%d, y=%d.', os.path.basename(imageFile), match.left, match.top)
            return match
    except pyautogui.ImageNotFoundException:
        #NOTE: Even though the documentation says locateOnScreen should send this exception, I've never actually seen it raise it. Still, for precaution
        logging.info("Image %s not found.", os.path.basename(imageFile))
        return None

def findAndClick(imageFile, delayBeforeClicking=0):
    """Scans the screen to find a section equal to the imageFile and clicks the center of the section if found, after the specified delay in
```

```
        seconds
    (by default it has no delay).
    Returns True if the image was located and clicked, False otherwise.

    The function matches a section of the screen if 90% of its' pixels match the given imageFile.
    """

    buttonLocation = findImage(imageFile)
    if buttonLocation != None:
        pause(delayBeforeClicking)
        pyautogui.click(buttonLocation.left, buttonLocation.top)
        return True
    else:
        return False

def waitForImage(imageFile, addedDelay=0):
    """Continously scans the screen every second to find a section equal to the imageFile until a match is found
    and then pauses for the specified time in addedDelay.
    Returns a box with the position of the match in screen coordinates.

    This function can loop forever if a match is never found.
    """

    while not (location := findImage(imageFile)):
        logging.info("Waiting for image %s. Pausing for 1 second and rechecking...", os.path.basename(imageFile))
        pause()
    pause(addedDelay)
    return location

def waitForImageAndClick(imageFile, delayBeforeClicking=0):
    """Continously scans the screen every second to find a section equal to the imageFile until a match is found
    and then clicks the center of the section after the specified delayBeforeClicking.

    This function can loop forever if a match is never found.
    """

    while not findAndClick(imageFile, delayBeforeClicking):
        logging.info("Waiting for image %s. Pausing for 1 second and rechecking...", os.path.basename(imageFile))
        pause()

def openBitWardenFailSafe(maxRetries = 5):
    """This functions opens the main BitWarden extension window, even if the extension window gets closed by some interruption (new tab opened, etc...).
    The function retries up to maxRetries to open the BitWarden extension and if it fails, it clicks on the extension button again and retries until
    BitWarden is open.

    Necessary because BitWarden opens a new tab to congratulate us for installing it and depending on the timing can cancel the extension window,
    so this approach is easier and more consistent.
    """

    waitForImageAndClick(EXTENSIONS_BUTTON)

    currRetries = 0
    found_bitwarden = None

    while currRetries < maxRetries:
        found_bitwarden = findImage(BITWARDEN_BUTTON)

        if found_bitwarden:
            pyautogui.click(found_bitwarden.left, found_bitwarden.top)
            return

        currRetries += 1
        logging.info("Retrying for image %s (%d out of %d).", os.path.basename(BITWARDEN_BUTTON), currRetries, maxRetries)
        pause()
    else:
        openBitWardenFailSafe()
    return
#endregion

def setEnvironmentURL(googleChromeCmd):
    """This function will properly set BitWarden's environment URL to point to the local BitWarden server. Opens Chrome with the given googleChromeCmd.

    This function closes Chrome at the end. Why? Because if we do not close Chrome after setting the env URL, there will be 4-5 different processes with
    the BitWarden tag sleeping and closing Chrome fixes that issue. We only want 1 BitWarden process so we know which one to memdump. As to why this happens?
    No clue really :(
    """

    logging.info("Environment URL setup: Start")

    # Open chrome with the defined command
    pyautogui.hotkey('alt', 'f2')
    waitForImage(COMMAND_PROMPT, 1)
    #For reference:
    pyautogui.write(googleChromeCmd)
    pause(1)
    pyautogui.press('enter')

    openBitWardenFailSafe()

    #Click the settings button before the log-in
    waitForImageAndClick(BITWARDEN_ENV_SETTINGS)

    #Write localhost as the server URL
    pyautogui.press('tab', 3, 0.15)
    pyautogui.write("localhost")
```

```python
    pyautogui.press('enter')

    #Close chrome to start the testing
    pause(2)
    pyautogui.hotkey('alt', 'f4')
    logging.info("Environment URL setup: Finished")

def performTest(googleChromeCmd, nthTest, memDumpDirectory, extensionName):
    """This function performs an entire test.
    googleChromeCmd: the command to open up Google Chrome
    nthTest: the number of the current test
    memDumpDirectory: where should the memory dumps be stored
    extensionName: the current extension name being tested
    """

    logging.info("Starting test %d.", nthTest)

    # Open chrome with the defined command
    pyautogui.hotkey('alt', 'f2')
    waitForImage(COMMAND_PROMPT, 1)
    #For reference:
    pyautogui.write(googleChromeCmd)
    pause(1)
    pyautogui.press('enter')

    #Click the extension button and then BitWarden
    openBitWardenFailSafe()

    # Select and click Login
    waitForImageAndClick(LOGIN_BUTTON)

    # Get PID of Bitwarden browser extension
    chrome_extensions = [proc for proc in psutil.process_iter() if proc.name() == 'chrome' and ('--extension-process' in proc.cmdline())]
    if len(chrome_extensions) != 1:
        print(chrome_extensions)
        sys.exit("ERROR: Could not get PID of Bitwarden Chrome extension")

    pid = chrome_extensions[0].pid
    logging.info('PID of Bitwarden Chrome extension: %d', pid)

    #E-mail
    email_text = waitForImage(E_MAIL_TEXT)
    pyautogui.click(email_text.left, email_text.top + 10)
    pyautogui.hotkey('ctrl', 'a')

    #Type the e-mail address and replace the old one if there was
    pause()
    pyautogui.write(configFile['username'])
    pyautogui.press('tab')

    #Perform first memory dump (control mem dump)
    memdump(pid, nthTest, MEMDUMP_BEFORE_TYPING, memDumpDirectory, extensionName)
    pause(1)

    #Password details
    secret_password = configFile['password']
    firstpart, secondpart = secret_password[:len(secret_password)//2], secret_password[len(secret_password)//2:]

    #Write half the password first, mem-dump after
    pyautogui.write(firstpart, interval=0.15)
    memdump(pid, nthTest, MEMDUMP_MID_TYPING_MP, memDumpDirectory, extensionName)

    #Write the second half of the MP, mem-dump
    pyautogui.write(secondpart, interval=0.15)
    memdump(pid, nthTest, MEMDUMP_FINISH_TYPING_MP, memDumpDirectory, extensionName)

    #Perform login
    #NOTE: Just press enter to submit the form. This makes it universal for all scripts since the child component one does not follow the same tab order
    pyautogui.press('enter')

    #Perform memdump after the vault opens (Check when the options button is up)
    waitForImage(OPTIONS_BUTTON)
    memdump(pid, nthTest, MEMDUMP_ON_UNLOCK, memDumpDirectory, extensionName)
    pause()

    #Simulate task
    #https://player.vimeo.com/video/604015327
    pyautogui.hotkey('ctrl', 't')
    waitForImage(GOOGLE)
    pyautogui.write('https://player.vimeo.com/video/604015327')
    pyautogui.press('enter')
    waitForImageAndClick(PLAY_BUTTON, 1)
    logging.info('Playing video for %d seconds.', TASK_TIME)

    pause(TASK_TIME)

    logging.info('Simulation of task ended.')
    memdump(pid, nthTest, MEMDUMP_ON_TASK_FINISHED, memDumpDirectory, extensionName)

    # Locate and click the extensions icon
    waitForImageAndClick(EXTENSIONS_BUTTON)

    #Click the (now blue because we're logged in) BitWarden Button
    waitForImageAndClick(BITWARDEN_BLUE_BUTTON)

    #Click the settings button
    #NOTE: It's better to keep the locate button since there could be multiple entries in the password vault
    waitForImageAndClick(OPTIONS_BUTTON)
    #Terminate session button
    pause(3)
    pyautogui.press('tab', presses=14, interval=0.15)
    pyautogui.press('enter')

    #And terminate session
```

```python
    waitForImageAndClick(YES_BUTTON)

    #Mem-dump after exiting the session
    memdump(pid, nthTest, MEMDUMP_SESSION_TERMINATED, memDumpDirectory, extensionName)

    # Close chrome
    pause(2)
    pyautogui.hotkey('alt', 'f4')

if (len(sys.argv)) == 1:
    sys.exit("ERROR: Please run the script with at least 1 extension.\nExample: python3 /home/vagrant/passcert/memdump-tests/runLinux.py /home/vagrant/passcert/bw-browser-v1.55/build/")

#Set up the logger
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.INFO)

# Obtain monitor size
monitor_size = pyautogui.size()

#Config file
configFile = configparser.ConfigParser()

configFile.read('/home/vagrant/passcert/memdump-tests/config.ini')

configFile = configFile['DEFAULT']

if not configFile['username'] or not configFile['password']:
    sys.exit("ERROR: Please follow the instructions in the sampleconfig.ini before starting the tests")

#Set up the directory for the memory dumps
memDumpDirectory = os.getcwd()
if not configFile['memoryDumpDirectory']:
    logging.info('No directory set up for the memory dumps, using the current working directory instead: %s.', memDumpDirectory)
else:
    memDumpDirectory = configFile['memoryDumpDirectory']
    logging.info('Memory dump directory set to: %s.', memDumpDirectory)

numberOfTests = configFile.getint('numberOfTests', 0)

logging.info("Perfoming %d tests.", numberOfTests)

extension_list = []

for i in range(1, len(sys.argv)):
    extension_list.append(sys.argv[i])

extension_names = []
for dir in extension_list:
    extension_names.append(getExtensionName(os.path.normpath(dir)))

for i in range(len(extension_list)):
    # Define the chrome command
    size_opts = f"--window-position=0,0 --window-size={int(monitor_size.width)},{monitor_size.height}"
    other_opts = "--password-store=basic"
    ext_opts = "--load-extension=" + extension_list[i]
    flag_opts = "--allow-insecure-localhost"
    cmd = f"google-chrome {flag_opts} {size_opts} {other_opts} {ext_opts}"

    #Run tests
    for j in range(numberOfTests):

        #NOTE: Reset Chrome settings to avoid a) loading with more than 1 extension and b) Chrome might randomly disable the extension because it deems it "unsafe"
        #Also give it some time because chrome might still be writing stuff in the folder (https://unix.stackexchange.com/questions/506319/why-am-i-getting-directory-not-empty-with-rm-rf)
        pause(2)
        os.system("sudo rm -rf /home/vagrant/.config/google-chrome; sudo mkdir -p /home/vagrant/.config/google-chrome; sudo cp -rf /vagrant/data/google-chrome/* /home/vagrant/.config/google-chrome;
            sudo chown -R vagrant.vagrant /home/vagrant/.config/google-chrome")
        pause(2)

        setEnvironmentURL(cmd)

        performTest(cmd, j, memDumpDirectory, extension_names[i])
        logging.info("Percentage of tests completed for extension %s: %f%%.", extension_names[i], (j + 1) / numberOfTests * 100)

# Print final message
logging.info("ALL TESTS DONE.")
```

B

Results of our extensions

In this appendix we present the raw results of the memory dump analysis for our extensions.

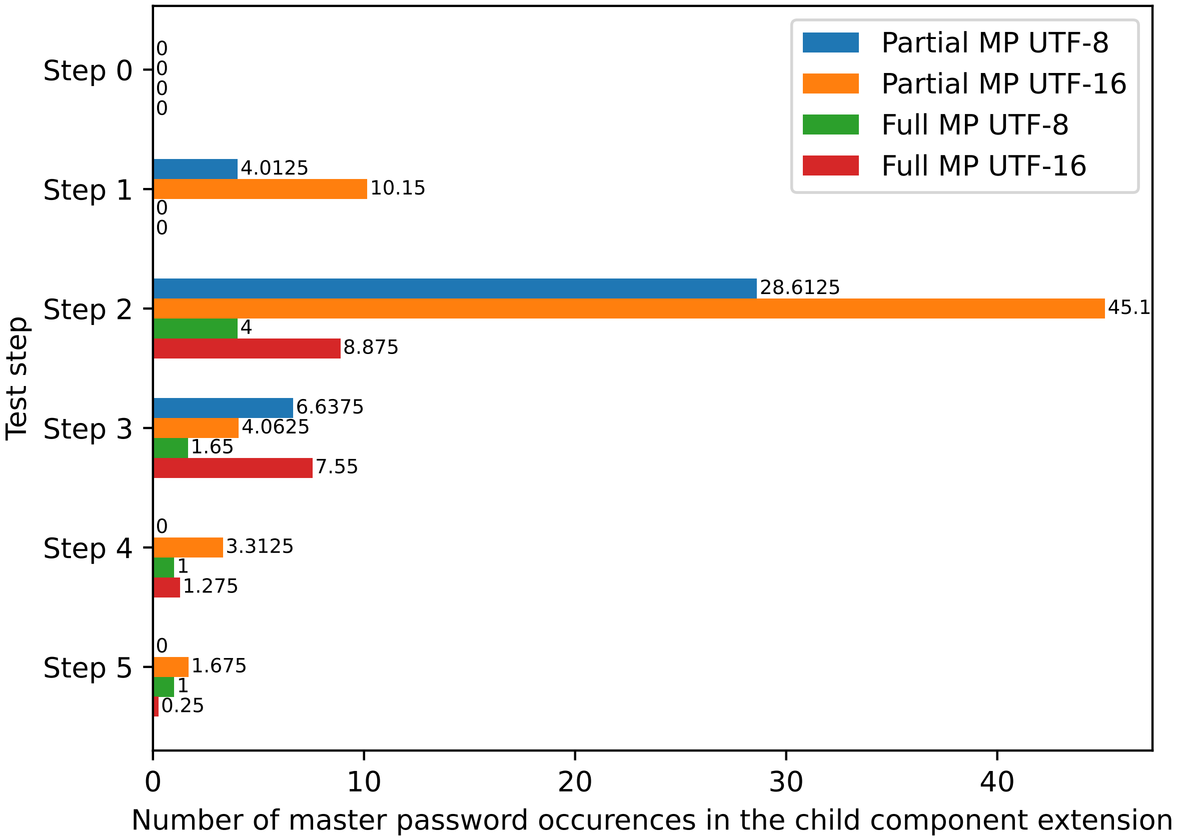

Figure B.1: Number of partial and full master password occurrences in memory in the child component extension by test step

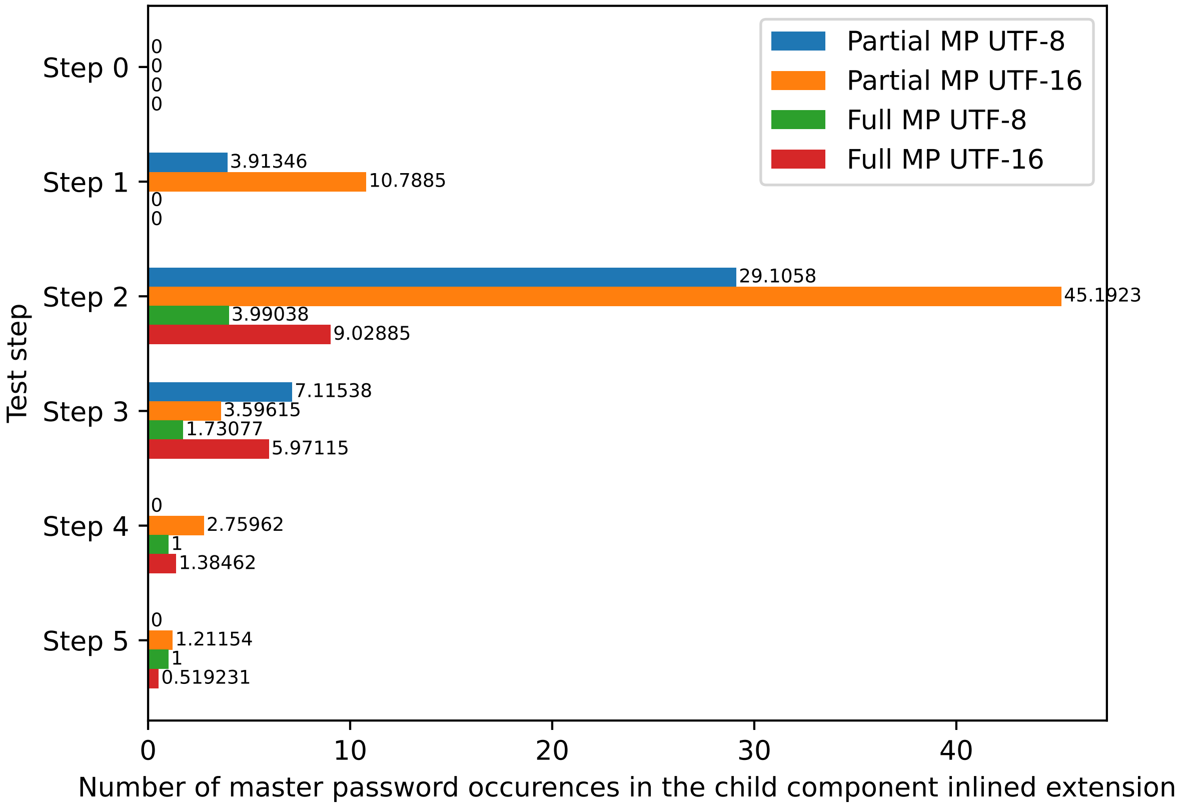

Figure B.2: Number of partial and full master password occurrences in memory in the child component inlined extension by test step

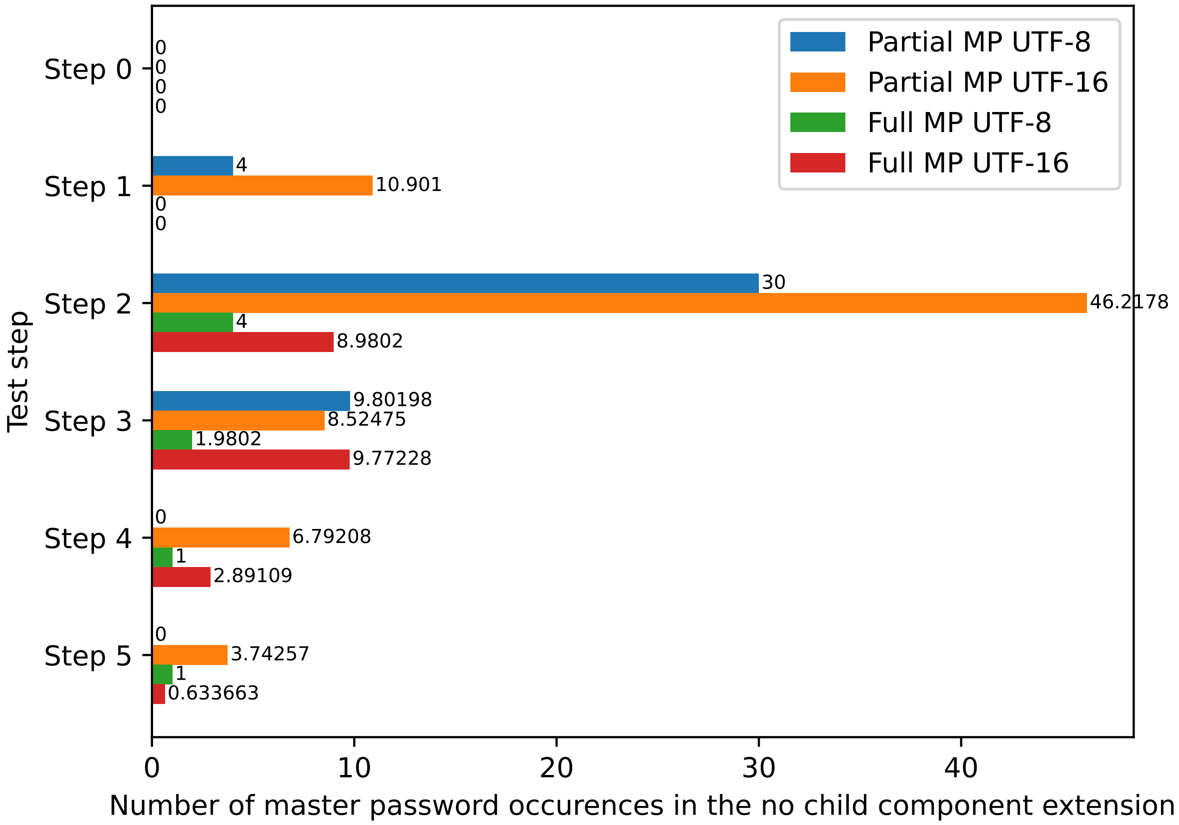

Figure B.3: Number of partial and full master password occurrences in memory in the no child component extension by test step

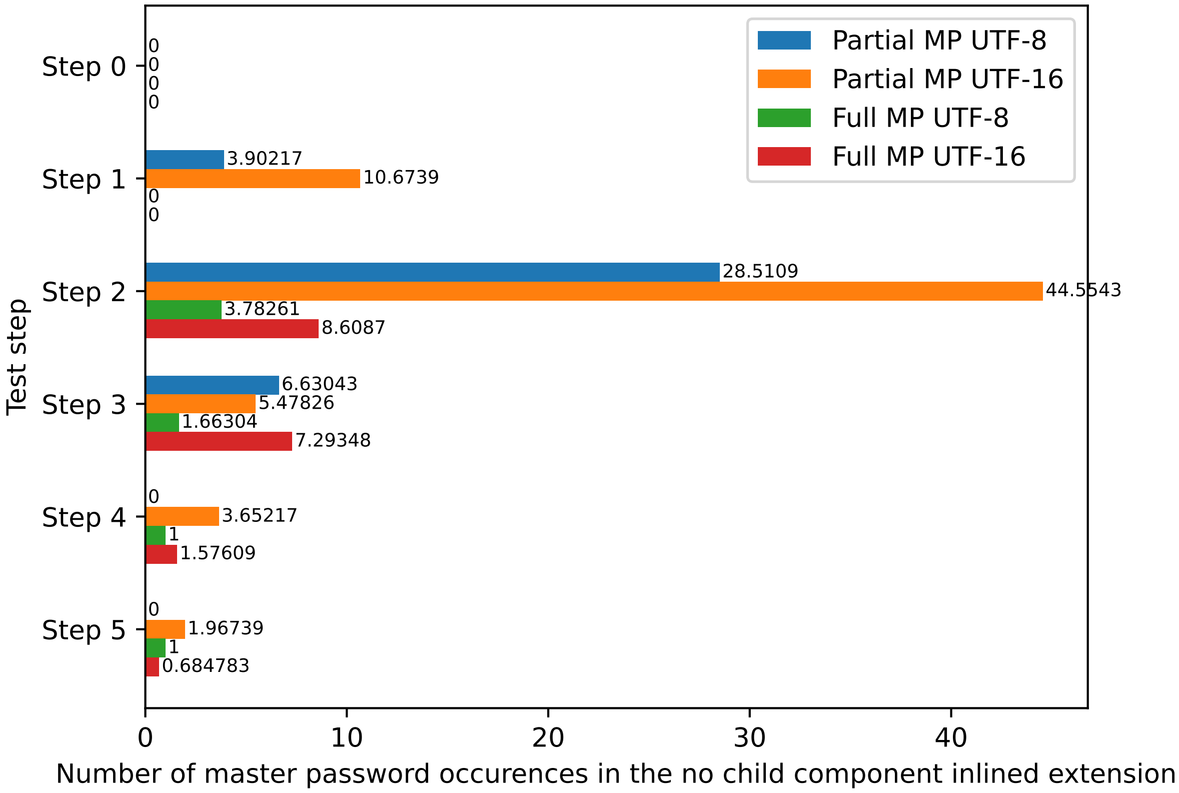

Figure B.4: Number of partial and full master password occurrences in memory in the no child component inlined extension by test step