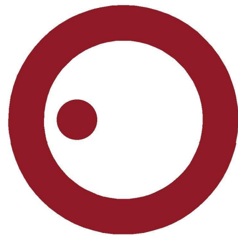


IPL

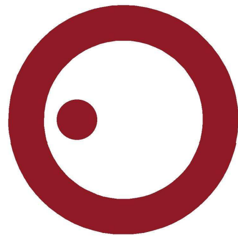
escola superior de tecnologia e gestão
instituto politécnico de leiria

Polytechnic Institute of Leiria
School of Technology and Management
Department of Computer Engineering
Master's Degree in Cybersecurity and Digital Forensics

FORENSIC ANALYSIS OF PASSWORD MANAGERS

MIGUEL FILIPE CUNHA CAMPOS

Leiria, July of 2025



IPL

escola superior de tecnologia e gestão

instituto politécnico de leiria

Polytechnic Institute of Leiria
School of Technology and Management
Department of Computer Engineering
Master's Degree in Cybersecurity and Digital Forensics

FORENSIC ANALYSIS OF PASSWORD MANAGERS

MIGUEL FILIPE CUNHA CAMPOS
Number: 2230465

Carried out as part of Project under the guidance of Professors: Miguel Monteiro de Sousa Frade, Miguel Cerdeira Marreiros Negrão and Patrício Rodrigues Domingues.

Leiria, July of 2025

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to my supervisors, Miguel Monteiro de Sousa Frade, Miguel Cerdeira Marreiros Negrão and Patrício Rodrigues Domingues for their guidance, patience, and support throughout the course of this project. Their insights and encouragement were invaluable and greatly contributed to the successful completion of this work.

I also wish to extend my sincere appreciation to my friend Rúben Sousa for his invaluable help during the writing of this report while I was completing my internship, and to Jorge Ferreira for introducing me to the field of cybersecurity. I am equally grateful to all others who supported me throughout this journey — your encouragement, constructive discussions, and presence during both the challenging and rewarding moments meant a great deal to me.

To my beloved family, especially my mother and my brother, thank you for your unconditional love, support, and belief in me. Your encouragement has been a constant source of strength.

This project would not have been possible without all of you. Thank you from the bottom of my heart.

RESUMO

A utilização crescente de gestores de palavras-passe como o Bitwarden e o KeePassXC levanta preocupações sobre a sua segurança e os vestígios digitais que podem deixar, particularmente em contextos de investigação forense. Este trabalho propõe uma análise forense detalhada destas duas aplicações populares de gestão de palavras-passe, com o objetivo de avaliar o tipo e a quantidade de informação sensível recuperável após a sua utilização.

A metodologia envolveu a criação de ambientes de teste controlados em máquinas virtuais (Windows e Linux), simulando cenários reais de uso. Foram usadas diversas ferramentas forenses (System Informer, HxD, Hashcat e John the Ripper) para análises ao sistema de ficheiros, memória volátil, extensões de navegador e hashes de palavras-passe. Adicionalmente, foi realizada uma análise do código-fonte e da arquitetura das aplicações, focando nos algoritmos de derivação de chaves (KDF), como o PBKDF2 e o Argon2.

Os resultados revelam que, apesar da encriptação robusta, vestígios como palavras-passe mestre, parâmetros de configuração, dados de autenticação e até fragmentos de palavras-passe podem permanecer acessíveis em ficheiros locais ou memória volátil, dependendo da aplicação e do sistema operativo. O Bitwarden, por ser baseado na nuvem, mostrou maior suscetibilidade a deixar artifactos em extensões de navegador, enquanto o KeePassXC, com armazenamento local, apresentou riscos associados à memória volátil e à extração de dados do seu ficheiro de base de dados. Verificou-se ainda que, em certos cenários, é possível extrair e tentar quebrar hashes das palavras-passe, embora a eficácia varie conforme o algoritmo de derivação utilizado.

Esta investigação contribui para a compreensão dos riscos forenses associados aos gestores de palavras-passe, oferecendo recomendações para investigadores forenses e programadores, visando a redução da pegada digital destas aplicações.

ABSTRACT

The increasing use of password managers like Bitwarden and KeePassXC raises concerns about their security and the digital forensic traces they might leave, particularly in forensic investigation contexts. This work proposes a detailed forensic analysis of these two popular password management applications, aiming to assess the type and quantity of sensitive information recoverable after their use.

The methodology involved creating controlled test environments in virtual machines (Windows and Linux), simulating real-world usage scenarios. Various forensic tools (System Informer, HxD, Hashcat, and John the Ripper) were employed for analyses of the file system, volatile memory, browser extensions, and password hashes. Additionally, an analysis of the applications' source code and architecture was performed, focusing on key derivation algorithms (KDFs) such as PBKDF2 and Argon2.

The results reveal that, despite robust encryption, traces like master passwords, configuration parameters, authentication data, and even fragments of passwords can remain accessible in local files or volatile memory, depending on the application and operating system. Bitwarden, being cloud-based, showed greater susceptibility to leaving artifacts in browser extensions, while KeePassXC, with local storage, presented risks associated with volatile memory and data extraction from its database file. It was also found that, in certain scenarios, it's possible to extract and attempt to crack password hashes, although efficacy varies according to the key derivation algorithm used.

This investigation contributes to understanding the forensic risks associated with password managers, offering recommendations for both forensic investigators and developers, aiming to reduce the digital footprint of these applications.

INDEX

Acknowledgments i
Resumo iii
Abstract v
Index vii
List of Figures xi
List of Tables xiii
List of Acronyms xv

1 Introduction 1
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Contributions . . . 2
1.4 Organization of the document . . . 3

2 Background 5
2.1 Password Managers . . . 5
2.2 KDF Algorithms . . . 7
2.2.1 PBKDF2 . . . 7
2.2.2 Memory-Hard KDF Alternatives . . . 9
2.3 Passkeys . . . 14
2.4 Summary . . . 15

3 Related Work 17
3.1 Security and Design of Password Managers . . . 17
3.2 Forensic Analysis of Password Managers . . . 18
3.3 Password Cracking in Forensic Investigations . . . 20
3.4 Summary and Research Gap . . . 21

4 Methodology 23
4.1 Objectives . . . 23
4.2 Testing Environment . . . 24
4.3 Virtualization Tools . . . 25

4.4 Tools and Software Used . . . 25
4.5 Forensic Analysis Workflow . . . 26
4.6 Ethical Considerations and Limitations . . . 27

5 Bitwarden analysis 29
5.1 PassKeys . . . 31
5.2 Clipboard . . . 33
5.3 File Analysis . . . 34
5.3.1 Desktop app . . . 35
5.3.2 Browser extension . . . 37
5.3.3 Web application . . . 44
5.4 Memory . . . 45
5.5 Cracking the hash . . . 50
5.6 Trying the Argon2id option . . . 51
5.7 Summary . . . 54

6 KeePassXC analysis 57
6.1 PassKeys . . . 58
6.2 Clipboard . . . 59
6.3 Autotype . . . 61
6.4 Memory Analysis . . . 62
6.5 File Analysis . . . 64
6.5.1 Application Data . . . 65
6.5.2 KeePassXC Password Database . . . 66
6.6 Proof of Concept . . . 73
6.7 Summary . . . 78

7 Discussion 81
7.1 Comparison of Artifact Recovery . . . 81
7.1.1 File System Artifacts . . . 81
7.1.2 Browser Extension Artifacts . . . 83
7.1.3 Volatile Memory Artifacts . . . 84
7.2 Comparison of Hash-Cracking Feasibility . . . 85
7.3 Security Trade-offs and Implications . . . 86
7.4 Summary . . . 88

8 Conclusion 89
8.1 Future Work . . . 90

Bibliography 91

Appendices

A Appendix A 101

Declaration 107

LIST OF FIGURES

Figure 1 Diagram of a simple Password Manager 6
Figure 2 Argon2 simplified scheme 13
Figure 3 Diagram of Bitwarden self-host deployment(*Bitwarden Blog* 2024) 29
Figure 4 Retrieving Clipboard contents using xclip in Linux 34
Figure 5 Retrieving Clipboard contents using Clip View in Windows 34
Figure 6 Bitwarden default clear clipboard setting 35
Figure 7 Source code from Bitwarden's Github 36
Figure 8 Bitwarden Extension 38
Figure 9 Contents in Google Chrome's directory for Bitwarden extension 40
Figure 10 Local storage of Bitwarden's locked web vault 44
Figure 11 Session storage of Bitwarden's unlocked web vault 44
Figure 12 Master password search command for Firefox memory dump 47
Figure 13 String example for Hashcat using Bitwarden's mode 50
Figure 14 Ouput of Hashcat after running Bitwarden's mode 51
Figure 15 Changing default settings of Bitwarden to Argon2id 52
Figure 16 Simple diagram to show how the argon2 implementation works 53
Figure 17 KeePassXC default encryption settings 58
Figure 18 Using Clip View to show the clipboard 61
Figure 19 Using xclip to show the contents of the clipboard 61
Figure 20 Using a keylogger to on KeePassXC Autotype 62
Figure 21 KeePassXC Memory Dump where we can see the database file 63
Figure 22 KeePassXC memory while the vault is locked 63
Figure 23 Database's XML structure extracted from memory 64
Figure 24 Unencrypted segment structure found in the .kdbx database 67
Figure 25 Breakdown of output of the script `keepass2john.py` 72
Figure 26 KeePassXC Composite Key and Derived Key demonstration 74
Figure 27 KeePassXC Base HMAC Key demonstrattion 75
Figure 28 KeePassXC HMAC Key demonstration 76

LIST OF TABLES

Table 1 Estimated cost to crack passwords 10
Table 2 List of the Operating systems used for the test environment 24
Table 3 List of tools used for the work presented in this report . . . 26
Table 4 List of the Bitwarden application versions. 30
Table 5 List of the Web Browsers used and their versions. 38
Table 6 Firefox Storage Paths on Windows and Linux 38
Table 7 Chrome Extension Storage Path on Windows and Linux . . 40
Table 8 List of the KeePassXC application versions. 58
Table 9 First four fields found in the outer header. 67
Table 10 Fields present in the outer header of the KeePassXC database. 69
Table 11 Comparison of KDF parameters 70
Table 12 Comparison of File System Artifacts found 82
Table 13 Comparison of Browser Extension Artifacts found 83
Table 14 Comparison of Volatile Memory Artifacts found 84
Table 15 Comparison of Password Hash Cracking Characteristics . . . 85

LIST OF ACRONYMS

ASCII American Standard Code for Information Interchange.

ASIC Application-Specific Integrated Circuit.

CLI Command-line interface.

CPU Central Processing Unit.

FPGA Field-Programmable Gate Array.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HMAC Hash-Based Message Authentication Code.

HSM Hardware Security Modules.

IP Internet Protocol.

IV Initialization Vector.

KDF Key Derivation Function.

LSM Log-Structured Merge-Tree.

OWASP Open Web Application Security Project.

PKCS Public-Key Cryptography Standards.

PRF Pseudorandom Function.

SST Sorted String Table.

VM Virtual machines.

1 INTRODUCTION

Password managers have become increasingly popular as a means to address the ever-growing challenge of securely managing digital credentials. As cyber threats continue to evolve, these applications are often relied upon to safeguard sensitive data such as usernames, passwords, cryptographic keys, and other confidential information. However, with this reliance comes the need to assess the security and forensic implications of using such tools, particularly in contexts where user devices may be compromised or subjected to forensic investigation.

Our report presents a forensic analysis of two widely used open-source password manager applications: Bitwarden and KeePassXC, with a specific focus on their behavior in desktop environments. The aim is to evaluate how these applications handle sensitive information, what artifacts remain after usage, and to what extent these artifacts can be recovered using forensic techniques.

1.1 MOTIVATION

The motivation for this work stems from the increasing adoption of password managers —— for example, the European Union Agency for Cybersecurity (ENISA) recommends their use *Tips for secure user authentication | ENISA* (2025), and a study by Padalia et al. (2023) shows that, in an online questionnaire, 51% of individuals who did not use a password manager were likely to adopt one in the future —— as well as the lack of comprehensive forensic research covering both cloud-based and local password managers. While some studies such as Chatzoglou et al. (2024) and Hähni (2023) have highlighted the forensic potential of memory dumps and local file artifacts, there remains a significant research gap in comparative evaluations between differing application architectures.

Moreover, recent advancements in key derivation functions (KDFs), such as Argon2 (Biryukov, Dinu, Khovratovich, and Josefsson, 2021), warrant updated forensic methodologies. Understanding how these mechanisms operate, and whether they

effectively protect data even after device compromise, is crucial for cybersecurity professionals, developers, and digital forensic investigators.

1.2 OBJECTIVES

The primary objective of this work is to simulate realistic forensic scenarios involving password managers to assess the extent to which data can be recovered from them. This includes analyzing file systems, application memory, and browser extensions to uncover residual artifacts such as master passwords, cryptographic hashes, session tokens, and stored credentials.

This report also aims to answer several critical questions:

- What types of sensitive information can be retrieved from these applications after use?
- How do architectural choices —– such as cloud-based (Bitwarden) vs. local storage (KeePassXC) —– affect data recoverability?
- How effective are contemporary forensic tools and hash-cracking techniques, such as Hashcat and John The Ripper, against encrypted artifacts from these applications?
- What are the security trade-offs between usability and forensic traceability?

The findings are intended to inform both forensic practitioners and application developers, contributing to a better understanding of how password manager design decisions impact digital investigations.

1.3 CONTRIBUTIONS

This study is situated at the intersection of digital forensics, application security, and cryptographic analysis. The analysis focuses on two contrasting password manager models:

- **Bitwarden** - a cloud-centric password manager that offers cross-platform synchronization and zero-knowledge encryption.
- **KeePassXC** - a local-only password manager that stores data in encrypted database files without relying on external servers.

The major contributions of this dissertation include:

- A hands-on forensic analysis framework applied to password manager applications.
- Development of custom scripts for extracting and analyzing artifacts when default tools fail, particularly for browser extensions and memory dumps.
- Comparative analysis of forensic artifacts in different environments (Windows, Linux, browser extensions, volatile memory).
- Evaluation of password hash cracking feasibility using dictionary-based attacks and memory-derived wordlists.

1.4 ORGANIZATION OF THE DOCUMENT

This document is organized as follows:

- **Chapter 2: Background** – Introduces foundational concepts relevant to the analysis, including password manager operations, KDFs (PBKDF2 and Argon2), and passkeys.
- **Chapter 3: Related Work** – Reviews literature on the forensic analysis of password managers and outlines existing tools and methodologies.
- **Chapter 4: Methodology** – Describes the test environments, software tools, and step-by-step forensic workflow adopted in this research.
- **Chapter 5: Bitwarden Analysis** – Details the forensic analysis of Bitwarden's desktop application, browser extension, and memory artifacts.
- **Chapter 6: KeePassXC Analysis** – Presents the forensic findings for KeePassXC, focusing on its database structure, memory use, and clipboard behavior.
- **Chapter 7: Discussion** – Compares and contrasts the forensic artifacts discovered in both applications, highlighting differences and implications.
- **Chapter 8: Conclusion and Future Work** – Summarizes the findings, discusses the limitations of the study, and suggests future directions for research.

2 BACKGROUND

This chapter aims to explain the base concepts needed to understand the analysis performed on the selected password managers.

2.1 PASSWORD MANAGERS

Password managers are applications designed to securely store passwords and other sensitive data while simplifying credential management. Their primary purpose is to enhance password security, addressing the common challenge of remembering multiple passwords and usernames. Without a Password Manager, users often resort to unsafe storage methods, such as writing passwords in a notepad or reusing passwords (Ray et al., 2020; Wash et al., 2016). To mitigate this risk, password managers provide a secure and convenient solution.

A Password Manager works by encrypting passwords and storing them in a digital vault, which can be accessed with a single master password. This means the user only needs to remember one password to access all its accounts, making it easier to use strong and unique passwords for each one.

Password Managers generally work by:

- **Installation and Setup**: Users choose a Password Manager and install it on their device or use a web-based service. During setup, they are required to create a strong master password, which serves as the key to unlock the encrypted vault. For example, 1Password (2025) and many other Password Managers emphasize the importance of a robust master password for security.
- **Storing Passwords**: When logging into a website or creating a new account, the Password Manager can generate a strong, random password using its built-in generator. This password, along with the username and website URL, is then saved in the encrypted vault. These features remove the necessity for users to manually generate or memorize intricate passwords.

- **Accessing Passwords**: For login, the Password Manager can automatically fill in the username and password, typically through a browser extension (like in the case of bitwarden browser extension) or a desktop app (such as KeePassXC application).
- **Synchronization Across Devices**: Many password managers, especially cloud-based ones, synchronize the encrypted vault across multiple devices. This ensures users can access their passwords from their computer, smartphone, or tablet seamlessly.
- An interesting and noteworthy point is that certain password managers, like Bitwarden (2025), implement a zero-knowledge architecture. This means data gets encrypted directly on the device prior to being transmitted to their servers, guaranteeing that even the service provider cannot view the stored passwords.

What separates a good Password Manager from a bad one is how it protects the vault. Normally, Password Managers use a Key Derivation Function (KDF) for creating a key from a secret value, often called a master password, and an encryption algorithm to encrypt the vault with the key generated from the KDF. In Figure 1 we can see a simple diagram showing this process.

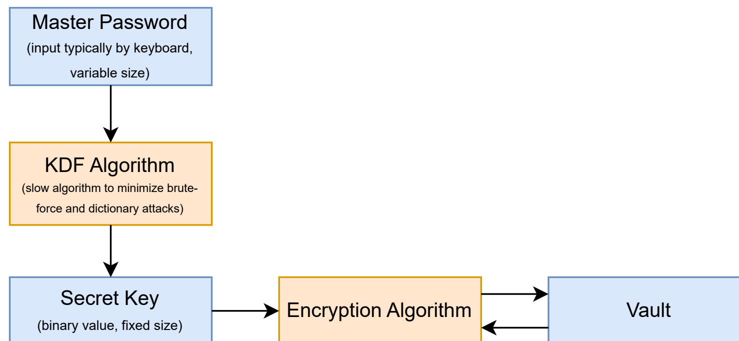


Figure 1: Simple diagram showing how a simple Password Manager works

Password managers can be broadly categorized into two main types: those with an online component and those that operate solely on a local machine. Online password managers, such as Bitwarden and 1Password, store the vault on a remote server, with the master password typically serving as both part of the vault's encryption key and the account authentication credential. In contrast, offline password managers like *KeePass2* (2025) and *KeePassXC Password Manager* (2025) store all data locally, ensuring that no sensitive information is uploaded to external servers.

2.2 KDF ALGORITHMS

By diving deeper into how a Password Manager works, it is important we look at the KDF algorithms used by these applications. According to Chen (2024), a KDF is a function that, given a secret password and additional input data, generates keying material for use in cryptographic algorithms. In most cases, the KDF invokes the Pseudorandom Function (PRF) n times, concatenating the outputs until at least L bits are produced, then truncates the result to exactly L bits of keying material.The approach used to perform these repeated invocations is known as the mode of iteration. A PRF serves as the fundamental component for constructing a key-derivation function. In general, a PRF family $\{\texttt{PRF(s, x)} \mid s \in S\}$ consists of polynomial-time computable functions indexed by a seed s and an input x. When s is randomly chosen from S and remains unknown to observers, `PRF(s, x)` is computationally indistinguishable from a truly random function defined over the same domain and producing outputs within the same range.

One of the primary reasons for utilizing KDFs is their ability to be deliberately configured to operate slowly —– potentially taking several seconds per computation on a standard personal computer (Biryukov, Dinu, Khovratovich, and Josefsson, 2021). This intentional delay significantly mitigates the feasibility of brute-force and dictionary attacks.

As stated by Stallings (2013) a brute-force attack involves an adversary systematically attempting all possible password combinations in order to derive the correct output of the KDF and ultimately recover the original input. In contrast, a dictionary attack leverages a predefined list of commonly used passwords (often derived from password leaks or statistical analyses) to attempt to match a generated hash. In both scenarios, the computational delay imposed by the KDF increases the cost of each attempt, thereby enhancing security.

In this report, the focus will be on two prevalent KDF algorithms, PBKDF2 and Argon2, due to their revelance to the analysis discussed in subsequent chapters.

2.2.1 *PBKDF2*

Starting with PBKDF2, this algorithm is approved and recommended by NIST as of writing this report as can be seen in *NIST Special Publication 800-63B* (2025).

PBKDF2, or Password-Based Key Derivation Function 2, is a cryptographic Key Derivation Function designed to enhance security by making it computationally expensive to derive keys from passwords, thereby reducing vulnerability to brute-force and dictionary attacks. It is part of the RSA Laboratorie's Public-Key Cryptography Standards (PKCS) series, specifically PKCS #5 v2.0, and is also detailed in RFC 2898 (Kaliski, 2000).

The primary purpose of PBKDF2 is to perform key stretching, a process that increases the computational work required to derive a key, making password cracking significantly more difficult. It achieves this by applying a PRF, such as Hash-Based Message Authentication Code (HMAC) to the input password along with a salt value, and repeating this process many times. The salt, a securely generated random value, ensures that even identical passwords result in different derived keys, making precomputed attacks like rainbow tables more difficult. The number of iterations can be configured to balance security and performance, with higher iterations increasing resistance to cracking but also lengthening the time for legitimate operations like login or decryption. PBKDF2 takes four inputs: the password (P), the salt (S), the iteration count (c), and the desired derived key length (dkLen). This function can be represented as `PBKDF2(P, S, c, dkLen)`. It is also possible to select the PRF used by PBKDF2, such as HMAC. All this information can be seen in Moriarty et al. (2017).

The iteration count has seen recommendations to increase over time due to advancements in computing power. For instance, in 2000, a minimum of 1,000 iterations was recommended, while by 2023, the Open Web Application Security Project (OWASP) Password Storage Cheat Sheet recommends 600 000 iterations for PBKDF2-HMAC-SHA256 and 210,000 for PBKDF2-HMAC-SHA512 (*Password Storage - OWASP Cheat Sheet Series* 2025).

The key derivation process can be divided into multiple steps such as:

- **Input Parameters**: The function takes the password, salt, iteration count, desired key length, and a PRF, typically HMAC with a hash function like SHA-1, SHA-256, or SHA-512.
- **Block Generation**: The derived key (DK) is computed as DK = `PBKDF2(P, S, c, dkLen)`. If the desired key length exceeds the PRF output length, multiple blocks are generated. Each block T_i is computed as $T_i = F(P, S, c, i)$, where:
  - $F(P, S, c, i) = U_1 \oplus U_2 \oplus \cdots \oplus U_c$

  - $U_1 = \text{PRF}(P, S + \text{INT_32_BE}(i))$, where INT_32_BE(i) is the 32-bit big-endian representation of the block index i.
  - $U_2 = \text{PRF}(P, U_1), U_3 = \text{PRF}(P, U_2), \ldots, U_c = \text{PRF}(P, U_{c-1})$

    What this means is each block is calculated by applying the PRF to the password and the salt, with a special counter (i) added to each block to make them unique. If more key material is needed than the function can provide in one go, the process repeats multiple times. In each step, the output from the previous block is used as the input for the next, and the results of all these steps are combined to form the final key.

- **Final Output:** The blocks are concatenated to form the derived key of the requested length.

While PBKDF2 is widely used, as evidenced by its implementation in applications like 1Password and Bitwarden and various online services, it has limitations. It is considered old-fashioned and less secure against modern attacks, particularly those leveraging specialized hardware like GPUs, due to its low memory usage and increasing processing speeds of modern hardware. This makes it vulnerable to parallel processing attacks, where attackers can perform many computations simultaneously.

Some possible alternatives to PBKDF2 that take a more modern approach are:

- **Yescrypt**: Yescrypt is a flexible and scalable KDF designed for high-security password hashing based on scrypt. It offers a robust defense against various attacks by being highly configurable (Peslyak, 2015).
- **Scrypt**: Allows arbitrary memory usage, more resistant to hardware attacks (Percival and Josefsson, 2016).

These alternatives are designed to be memory-hard, making them more resistant to attacks using parallel processing.

2.2.2 *Memory-Hard KDF Alternatives*

Password crackers have evolved to efficiently crack password hashes that use multiple iterations, especially those that do not require much memory. To counteract this, memory-hard functions were developed. These are cryptographic functions that demand significant memory to compute, making them much more resistant to

attacks using specialized hardware like GPUs or Application-Specific Integrated Circuit (ASIC)[1].

One notable example of a memory-hard password hashing scheme is scrypt. According to Percival (2009), scrypt works by first using PBKDF2 with a PRF to derive p blocks of length MFlen (in octets) from the given password and salt. Each of these blocks is then independently processed using a mixing function called MF. After mixing, the blocks are fed back into PBKDF2, this time using the mixed blocks as the salt, to generate the final output.

In addition to this, scrypt introduces another layer of complexity by applying the MFcrypt function, which uses the SMix mixing function and the SHA-256 hash function, as can be seen below:

$$scrypt(P,S,N,r,p,dkLen) = MFcrypt_{\mathrm{HMAC_SHA256,SMix_r}}(P,S,N,r,p,dkLen)$$

When used for interactive logins, scrypt is significantly more computationally expensive than other schemes: it is approximately 35 times more expensive than bcrypt, and 260 times more expensive than PBKDF2 as it is stated in by Percival (2009). This high cost makes brute-force attacks far less practical. The Table 1 below, extracted from "STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS", shows estimated cracking costs for different KDFs using a brute-force attack.

KDF	6 letters	8 letters	8 chars	10 chars	40-char text	80-char text
DES CRYPT	$< \$1$	$< \$1$	$< \$1$	$< \$1$	$< \$1$	$< \$1$
MD5	$< \$1$	$< \$1$	$< \$1$	$\$1.1 \times 10^3$	\$1	\$1.5 billion
MD5 CRYPT	$< \$1$	$< \$1$	\$130	\$1.1 million	$\$1.4 \times 10^3$	$\$1.5 \times 10^{15}$
PBKDF2 (100 ms)	$< \$1$	$< \$1$	$\$1.8 \times 10^4$	\$160 million	$\$2.0 \times 10^5$	\$220 trillion
bcrypt (95 ms)	$< \$1$	\$4	$\$1.3 \times 10^5$	$\$1.2 \times 10^{12}$	\$1.5 million	$\$4.8 \times 10^{10}$
scrypt (64 ms)	$< \$1$	\$150	\$4.8 million	$\$4.3 \times 10^{10}$	\$52 million	$\$6 \times 10^{19}$
PBKDF2 (5.0 s)	$< \$1$	\$29	$\$9.2 \times 10^5$	\$8.3 billion	\$10 million	\$11 trillion
bcrypt (3.0 s)	$< \$1$	\$130	\$4.3 million	$\$3.9 \times 10^{10}$	\$47 million	$\$1.5 \times 10^{15}$
scrypt (3.8 s)	\$900	$\$6.1 \times 10^5$	\$19 billion	\$175 billion	$\$2.1 \times 10^{11}$	$\$2.3 \times 10^{23}$

Table 1: Estimated cost to crack passwords (the values were translated from short scale to long scale).

Memory-hard scheme can be used for more than password hashing, like for key derivation from low-entropy source and cryptocurrency designs.

1 ASIC is a microchip designed and manufactured for one specific purpose, rather than for general use. Because it is hardwired to perform a single function, it is significantly faster and more power-efficient at that task than flexible hardware like a Central Processing Unit (CPU) or Graphics Processing Unit (GPU).

The problems associated with this type of scheme is that it usually requires high complexity and its not flexible in separating time and memory costs. A solution for these problems is Argon2, as it aims to for the highest memory filling rate and effective use of multiple units while still offering great protection against tradeoff attacks. Argon2 is optimized for x86 systems and uses the cache and memory organizations of Intel and AMD CPUs. Argon2 has two main variants: Argon2d and Argon2i. Argon2d is faster and uses data-dependent memory, this means that multiple statements in a program access the same memory location and at least one of them writes to it. Argon2i uses data-independent memory access and this variant is the recommended one for password hashing as it more likely to resist side-channel attacks. There is also Argon2id that combines the strengths of both Argon2d and Argon2i making this variant a hybrid alternative for applications looking to use Argon2 (Biryukov, Dinu, Khovratovich, and Josefsson, 2021).

ARGON2 WORKING. First, Argon2 takes the input data (like the user's password, salt, memory value and other parameters) and puts them into a hash function. This produces a starting value, called H_0.

All variable-length inputs (like the password and salt) are added together, but before each one, Argon2 includes the length of the data. This makes sure everything is read correctly. The formula is:

$$H_0 = H(p, \tau, m, t, v, y, P_{length}, P, S_{length}, S, K_{length}, K, X_{length}, X)$$

Where:

- `p` = number of threads
- τ = desired output length
- `m` = memory size
- `t` = number of iterations
- `v` = version
- `y` = type (Argon2d, Argon2i)
- `P`, `S`, `K`, `X` = password, salt, secret key (optional), and extra data
- The actual number of blocks is m', which is rounded down to the multiple of $4p$.

- $q = m'/p$ columns

Argon2 then uses a block of memory. It splits this memory into `p` rows and `q` columns, forming a matrix of blocks.

Each block is 1 024 bytes. These are filled in using a special hash function H' built on top of Blake2b, based on the value H_0.

For the first two blocks in each row, it works like this:

$B_1[i][0] = H'(H_0 \parallel 0 \parallel i)$

$B_1[i][1] = H'(H_0 \parallel 1 \parallel i)$

For the rest of the blocks:

$B_1[i][j] = G(B_1[i][j-1], B_1[i'][j'])$

Here, `G` is a special function that mixes two blocks together, and which blocks get picked (i', j') depends on the Argon2 variant. If the number of iterations is more than 1 (the minimum recomended value is 3), Argon2 repeats the memory-filling process. After all `t` iterations are done, Argon2 takes the last block from each row and XORs(mixes) them together to get a single block:

$B_{\text{final}} = B_T[0][q-1] \oplus B_T[1][q-1] \oplus \cdots \oplus B_T[p-1][q-1]$

Then it hashes this block to produce the final output tag:

$Tag = H'(B_final)$

This process can be seen in the Figure 2 in a very simplified way.

Argon2 offers several key benefits, as can be seen in Biryukov, Dinu, and Khovratovich (2017):

- **High Performance:** Argon2 fills memory very efficiently, about two CPU cycles per byte for Argon2i, and up to three times faster with Argon2d. This makes it a great choice for applications that require memory-hardness but have limited CPU resources.
- **Tradeoff Resilience:** Argon2 is designed to resist time-memory tradeoff attacks, maintaining strong security even against adversaries who try to save memory at the expense of computation.
- **Scalability:** Argon2 can be tuned to use variable amounts of memory and CPU time, making it adaptable to a wide range of use cases.

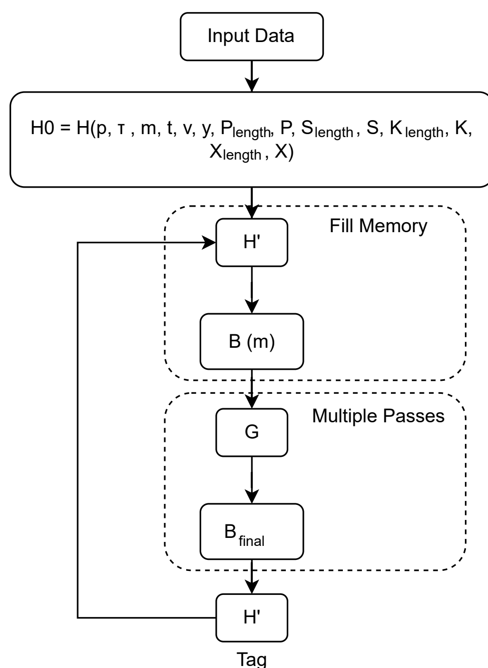


Figure 2: Argon2 simplified scheme

- **Parallelism:** The algorithm supports high levels of parallelism, capable of utilizing up to 2^{24} threads concurrently for increased performance on modern hardware.
- **Resistance to Specialized Hardware:** Argon2 is optimized for general-purpose x86 processors, making it inefficient to run on GPUs, Field-Programmable Gate Array (FPGA)[2], or ASICs, commonly used in password cracking tools.

However, the same features that make Argon2 strong can also present drawbacks:

- **Performance Overhead**: The memory-hard nature of Argon2, while a key security feature, can introduce significant performance overhead. This can be a limitation in resource-constrained environments.

In the context of password managers, using a memory-hard algorithm like Argon2 is a better choice than relying on PBKDF2 with a very high iteration count. It's important to note that this does not make the Password Manager unbreakable, but

2 An FPGA is an integrated circuit that can be configured by a customer or a designer after manufacturing, allowing it to be programmed and reprogrammed to perform a wide variety of digital functions.

when combined with strong security practices, namely large and complex master passwords, it can offer robust protection for the passwords stored within it.

2.3 PASSKEYS

Passkeys are digital authentication credentials that aim to replace traditional passwords, this is supported by an article written by *CNBC* (2023). Technically, each passkey is a cryptographic key pair: a private key kept on the user's device and a matching public key stored by the online service. When a user registers a site that supports passkeys, the user's device generates this key pair and sends only the public key to the server. On login, the device proves possession of the private key by signing a server-issued challenge, which the server verifies using the public key. Because the private key never leaves the device, websites never see or store any secret that could be stolen. In practice, passkeys leverage the FIDO2 standards to manage this public-key process, this information can be seen in *FIDO Authentication Specifications | FIDO Alliance* (2025) . In summary, passkeys are "passwordless" credentials: instead of a memorized string, the user signs in with a biometric or PIN to unlock a device-based secret, a private key.

Beyond reuse, passwords are vulnerable to phishing and theft. A malicious site or email can trick a user into revealing their password, giving attackers full access. Passkeys on the other hand use public-key cryptography and the FIDO2 framework.

The process can be summarized in two phases:

- **Registration** (key creation): When the user sets up an account or enable passkey login, the device's authenticator (e.g. smartphone or security key) generates a key pair. The public key is sent to the service and stored in users account profile. The private key stays on your device (usually within a secure enclave or hardware module) and never leaves it. The device may require the user to unlock it (e.g. via a fingerprint, Face ID, or PIN) for the creation and storage of the key.
- **Authentication** (login): To sign in, the server sends a cryptographic challenge in the form of a random nonce to the device. The device first verifies their identity locally: for example, it asks for the users biometric information or PIN to unlock the private key. It then uses the private key to sign the challenge. This signed challenge is returned to the server, which verifies it with the stored public key. If the verification succeeds, the server knows the users has a

legitimate private key and grants them access. The private key itself is never transmitted, so even if an attacker intercepts the exchange, they cannot learn any secret.

Passkeys rely on open standards. The browser/server interaction is handled by the WebAuthn protocol (*WebAuthn* 2025), which uses public-key cryptography. Additionally, the Client-to-Authenticator Protocol (CTAP)—described in *Client to Authenticator Protocol (CTAP)* (2025)—links browsers or applications to authenticators, such as built-in secure hardware or USB security keys. These standards ensure interoperability across platforms. Also every passkey is tied to the specific site or app where it was created. If a user is tricked into a phishing site, the passkey will simply not work because the origin url does not match. This binding is a key security feature: it prevents a passkey from being used on a malicious lookalike site for example.

Some of the advantages of passkeys outlined in Alliance (2025) are:

- **Stronger Security**: Passkeys eliminate most password-related attack vectors. They provide phishing-resistant and brute-force-resistant logins. Every key is cryptographically strong and unique per site, so there are no weak or reused credentials for attackers to guess.
- **Logging in is streamlined**: Instead of typing a password, the user simply unlocks their device as they normally do (fingerprint, face, PIN).This is both faster and less frustrating.

While some limitations of passkeys, as discussed in Lassak et al. (2024) and Matzen et al. (2025), include:

- **Device Dependence**: Since passkeys rely on having an authenticator device handy. If a user's phone or security key is lost or breaks, they could lose their ability to sign in. Cloud synchronization can help, but it must be set up.
- **Compatibility and Adoption**: Passkeys require both the user's device and the website/application to support the technology. Older sites or browsers that do not support WebAuthn will still require passwords.

2.4 SUMMARY

This chapter presented the foundational concepts necessary for evaluating the security of password managers. It began by introducing password managers, explaining

their role in storing and managing user credentials securely. Key features such as encryption, password generation, and synchronization across devices were discussed, alongside the distinction between offline and online solutions and the importance of zero-knowledge architecture in enhancing user privacy.

The chapter examined the cryptographic foundations of password-based security, focusing on KDFs. It explored PBKDF2 as a widely adopted standard, its reliance on salts and iteration counts, and its vulnerability to hardware-accelerated attacks. More resilient alternatives like bcrypt and scrypt were introduced, emphasizing their memory-hard properties. The section concluded with a discussion of Argon2, the state-of-the-art KDF, detailing its resistance to side-channel and tradeoff attacks, its configurable parameters, and the differences between its variants —– Argon2d, Argon2i, and Argon2id.

Following this, the chapter also addressed a significant emerging trend: passkeys. These are cryptographic credentials designed to replace traditional passwords entirely, offering phishing resistance and stronger authentication by leveraging public-key cryptography and secure device-bound storage. Passkeys represent a major shift toward passwordless authentication, with growing support across platforms and increasing integration into modern password managers like for example using Passkeys instead of a master password.

3 RELATED WORK

In this chapter, relevant literature was reviewed to provide context for the forensic analysis of password managers. Password managers have become essential tools in modern cybersecurity practices, offering users a convenient and secure way to store and manage login credentials. As their adoption increases, so does the interest in evaluating their security and understanding the forensic artifacts they may leave behind.

By analyzing existing studies, this chapter aims to identify gaps in the literature and demonstrate how this report contributes to the field, particularly by offering a practical and comparative forensic analysis of two popular open-source password managers: Bitwarden and KeePassXC.

3.1 SECURITY AND DESIGN OF PASSWORD MANAGERS

A study on the forensic implications of password managers was conducted by Gray et al. (2016). Their research focused on evaluating the residual artifacts left behind by local password managers, including applications such as KeePass[1] and Password Safe. The study emphasized the importance of conducting forensically-sound investigations and highlighted that, despite the use of encryption, sensitive data could still be recovered from system memory, temporary files, and other application traces.

Using a structured forensic methodology, the authors were able to extract various forms of artifacts, such as user credentials, configuration data, and logs, by analyzing memory dumps and file system remnants. In particular, they demonstrated that even after a vault was locked or an application was closed, fragments of passwords and metadata could persist in RAM or in improperly deleted files. This finding underscored the fact that encryption alone is not sufficient to guarantee forensic

1 KeePass is the original, Windows-focused password manager that offers extensive customization through plugins. KeePassXC, on the other hand, is a cross-platform fork of KeePass that provides a more modern and consistent user experience across Windows, macOS, and Linux, with built-in browser integration.

security if applications do not handle memory management and cleanup procedures carefully.

The study also introduced a framework for evaluating the forensic soundness of password manager implementations and suggested improvements for developers to reduce digital footprints. Although the scope was limited to a few local password managers available at the time, this work laid a foundational basis for future research, including this report, which expands on their approach by incorporating both local and cloud-based password managers, more recent application versions, and additional forensic techniques such as browser extension analysis and hash cracking.

Building on the previous security-focused evaluation, Luevanos et al. (2017) conducted an extensive comparative analysis of the security and usability of various open-source password managers. Their study examined factors such as encryption mechanisms, KDFs, storage practices, and the overall user experience. The researchers assessed each application based on its ability to securely store credentials, its resistance to common attack vectors, and the transparency of its security design.

One of the key contributions of this work was the identification of insecure design choices in several password managers, including weak or outdated cryptographic configurations, reliance on default settings that reduce protection, and inconsistent implementations of KDFs like PBKDF2 and Argon2. The study also discussed trade-offs between security and usability, it is worth noting that overly complex security features might discourage proper usage by non-technical users, potentially leading to unsafe workarounds.

While the research did not involve in-depth forensic analysis, it provided a valuable overview of common weaknesses that may later be exploited or revealed during forensic investigations. The findings emphasize the importance of analyzing not only artifacts left behind by password managers but also the architectural decisions that may contribute to those artifacts. This aligns with our objective of assessing both the practical recoverability of sensitive data and the technical robustness of password manager implementations.

3.2 FORENSIC ANALYSIS OF PASSWORD MANAGERS

The report made by Hähni (2023) introduced a comprehensive forensic framework known as Password Manager Forensics (PMF), designed to automate and standardize

the analysis of password manager applications. This study focused on two widely used open-source password managers, KeePass and Bitwarden, and proposed a structured methodology to extract forensic artifacts from both file systems and volatile memory. The PMF framework leverages custom Python scripts and open-source tools to analyze artifacts such as configuration files, encrypted vaults, local databases, browser extensions, and memory dumps.

A key strength of the PMF approach is its modularity and automation. The framework includes predefined modules for parsing specific artifact formats (like SQLite and LevelDB databases), detecting sensitive data patterns, and generating memory wordlists for hash-cracking. The authors demonstrated its effectiveness by recovering passwords, usernames, KDF parameters, and even fragments of master passwords from RAM in test environments.

This work represents one of the most directly comparable studies to the present dissertation. Both adopt a practical forensic approach targeting similar applications and artifacts. However, this report differs by manually applying forensic techniques without relying on this PMF automation layer, and by placing a stronger emphasis on evaluating how architectural differences (local vs. cloud-based) affect data exposure. Moreover, while PMF focuses on structured data extraction, this work complements it by exploring attack feasibility through cracking techniques and browser-based artifact analysis.

The article Chatzoglou et al. (2024) critically examines 24 popular password managers, comprising 12 desktop applications and 12 browser plugins, to assess their handling of sensitive credentials in system memory. The findings reveal that only three desktop applications and two browser plugins avoid storing plaintext passwords in RAM across various usage scenarios. Alarmingly, despite the potential security implications, only two vendors have acknowledged this issue as a vulnerability. The research underscores a significant oversight in the industry regarding in-memory data protection.

In a technical report published by Leipold (2024), the author conducted an in-depth forensic analysis of a self-hosted Bitwarden server deployment. The study focused on the server-side artifacts generated by Bitwarden's Docker-based infrastructure, particularly those stored in the Microsoft SQL Server database used by the platform. The analysis covered aspects such as account creation metadata, configuration files, event logs, and API interactions between the server and client applications.

The author demonstrated that although Bitwarden encrypts sensitive user data client-side, meaning the server does not store plaintext credentials or unencrypted vault contents, valuable metadata can still be recovered from the server environment. For example, login attempts, vault access timestamps, Internet Protocol (IP) addresses, and failed authentication events can be extracted from database tables and log files. These artifacts may provide investigators with useful contextual information during an incident response or forensic examination, even in the absence of decrypted data.

While Leipold analysis is focused solely on the server infrastructure and does not explore local or browser-side artifacts, it offers a complementary perspective to this report. Together, they illustrate a full-spectrum view of Bitwarden's footprint: from server logs and configuration artifacts to client-side memory and application data. This reinforces the need for forensic practitioners to consider all layers of a password manager ecosystem — client, browser, and server — during investigations.

3.3 PASSWORD CRACKING IN FORENSIC INVESTIGATIONS

Kanta, Coray, et al. (2021) investigated the practicality of password cracking in digital forensic investigations by analyzing the guessability of over 3.9×10^9 real-world credentials sourced from publicly available data breaches. Their study aimed to determine how likely it is that forensic analysts can recover plaintext passwords from hashed data using commonly available wordlists and cracking techniques. The researchers categorized passwords based on their entropy, patterns, and structure, and then assessed the effectiveness of various cracking strategies.

The study found that a significant portion of real-world passwords, particularly those reused across platforms or based on predictable structures, such as dictionary words with appended numbers, could be cracked in a relatively short amount of time using tools such as *Hashcat* (2024) and *openwall/john* (2025). These results reinforce the idea that password strength and unpredictability remain critical, especially in the context of forensic recovery, where cracking recovered hashes may expose sensitive user data.

Although the study does not specifically address password managers, its findings are directly relevant to forensic investigations involving such applications. Many password managers store master passwords or encryption keys as hashed values using KDFs like PBKDF2 or Argon2. The success of an investigation may therefore depend on the feasibility of cracking these hashes using realistic attack scenarios.

In the present report, the insights from Kanta, Coray, et al. support the use of dictionary-based cracking attempts and justify the inclusion of memory-derived wordlists as part of the forensic methodology.

In a follow-up to their 2021 study, Kanta, Coisel, et al. (2024) explored the effectiveness of context-based password cracking in digital forensics. Rather than relying solely on generic wordlists, their approach incorporates user-specific or scenario-based information, such as usernames, email addresses, dates, and system artifacts, to generate more accurate and efficient password guesses. The authors evaluated this method using real-world data sets and showed that contextually enriched wordlists significantly improved cracking success rates, particularly against moderately strong passwords.

This technique is especially relevant in forensic contexts where investigators have access to auxiliary data from the same system or environment. For example, strings recovered from memory dumps, filenames, or browser history can inform the construction of targeted attack dictionaries. The study further demonstrated that combining this contextual data with conventional brute-force and rule-based strategies yielded superior results compared to standard cracking approaches alone.

In the context of this report, the findings by Kanta, Coisel, et al. provide a strong justification for the use of custom wordlists derived from forensic artifacts, particularly those extracted from memory dumps of password manager processes. This aligns with the methodology employed in the present report, where captured memory contents were used to construct targeted wordlists for cracking master password hashes, using tools like Hashcat. Their results validate this approach as both realistic and forensically accurate.

3.4 SUMMARY AND RESEARCH GAP

The reviewed literature demonstrates a growing interest in both the security and forensic analysis of password managers. Early foundational studies, such as Gray et al. (2016), highlighted the forensic risks associated with residual data in local password managers, while more recent works by Luevanos et al. (2017) and Hähni (2023) provided structured evaluations of application security and proposed frameworks for systematic artifact extraction. Additionally, specialized analysis, such as Leipold (2024) study on Bitwarden's self-hosted server, expanded the scope of forensic research to include server-side components, illustrating the complexity of password manager ecosystems.

This reviewed literature also emphasizes the increasing relevance of password cracking in forensic contexts. Studies by Kanta, Coisel, et al. (2024) and Kanta, Coray, et al. (2021) underscore the practicality and effectiveness of both traditional and context-based cracking techniques, supporting their use in real-world investigations involving encrypted or hashed credentials.

Other works deserve being meantioned like the article from Gallus et al. (2025). They conduct a comprehensive security assessment of various password managers, employing comparative analysis and penetration testing to identify vulnerabilities. The findings highlight significant security gaps, emphasizing the need for improved protective measures in password management solutions. The research places a big importance of rigorous security evaluations to enhance user data protection.

Despite these contributions, a number of gaps remain. Most studies focus either on security assessments or on isolated forensic artifact extraction, often with a limited range of applications and without a comparative approach. There is a lack of practical research that examines how local and cloud-based password managers differ in terms of forensic traceability, particularly when evaluated across multiple platforms and usage contexts (for example, desktop apps, browser extensions, and server setups).

Our work aims to address these gaps by conducting a comparative forensic analysis of two open-source password managers, KeePassXC and Bitwarden, using a hands-on methodology that encompasses file system analysis, memory dumps, browser artifacts, and password hash cracking. The following chapter outlines the methodological framework and tools used to perform this analysis.

4 METHODOLOGY

This chapter outlines the research design and methodological framework employed to address the study's objectives. It details the data collection techniques, analysis procedures, and the rationale behind the chosen approach.

4.1 OBJECTIVES

This study conducts a forensic analysis of password managers to identify residual artifacts, potential vulnerabilities, and sensitive data left behind by these applications. The goal is to simulate realistic forensic investigations that may arise in cybersecurity incidents, helping determine what types of user information can be retrieved post-usage, and how securely these applications handle critical data such as master passwords, encryption keys, and user credentials.

Additionally, the methodology aims to assess how differences in storage architecture (local vs. cloud-based) impact the exposure or recoverability of forensic artifacts. The study also seeks to evaluate the effectiveness of various forensic tools and techniques in extracting relevant evidence from different operating environments.

Two widely used open-source password managers were selected for evaluation, each representing a distinct approach to data storage and synchronization: KeepassXC and Bitwarden. The former is primarily local, while the latter has a bigger focus on cloud. This comparative framework allows for an examination of not only the specific applications but also the broader security implications of local versus cloud-based password management solutions.

The criteria for selection included:

- Open-source availability, enabling access to source code and file structure for deeper analysis.
- Popularity and active development, ensuring the relevance of findings to real-world usage.

- Support of KDF such as PBKDF2 and Argon2, allowing for evaluation of their implementation in practice.
- Cross-platform support, facilitating analysis across multiple operating systems.

By selecting two contrasting models, this report aims to provide a balanced perspective on the forensic impact of different architectural decisions in password manager design.

4.2 TESTING ENVIRONMENT

All tests conducted on the Password Manager applications were performed within virtual machines in a controlled environment to minimize potential bias or interference with the results. A total of three testing environments were created: one Windows 11 instance and two Linux environments—one running Parrot OS and the other Ubuntu Server.

The Windows 11 and Parrot OS environments were utilized to test the client-side applications of the password managers. The Ubuntu Server environment was specifically set up to host the server-side application for Bitwarden, which will be discussed in greater detail in subsequent chapters.

Details regarding the operating systems and their respective versions for each testing environment are presented in Table 2.

Environment	**Version**
Windows 11	24H4 (OS Build 26 100.2314)
Parrot OS	6.1.0-1parrot1-amd64
Ubuntu Server	Ubuntu 24.04.1 LTS

Table 2: List of the Operating systems and their versions created for the test environment.

The hardware specifications for the system used in these test environments are as follows:

- **GPU**: NVIDIA GeForce RTX 3070, 8GB VRAM
- **CPU**: AMD Ryzen 7 5800X
- **RAM**: 32 GiB

4.3 VIRTUALIZATION TOOLS

VirtualBox is a virtualization software solutions that enable multiple operating systems to run on a single physical machine. VirtualBox is notable for being open-source. This tool was utilized to create and manage the virtual machines that comprised the testing environments.

Virtual machines (VM) were chosen for their flexibility and their ability to provide isolated and controlled testing environments, which are crucial in forensic investigations. By using VMs, it was possible to:

- Take consistent snapshots before and after each stage of testing, allowing for easy rollback and analysis of system states.
- Contain the scope of each experiment, ensuring no contamination between tests or unintended system modifications.
- Simulate different operating system environments (e.g., Windows 11, Parrot OS, Ubuntu Server) without the need for multiple physical devices.
- Facilitate repeatability, so that the same tests can be executed multiple times under identical conditions.

Moreover, VMs usage supports better documentation and traceability, which are essential components in formal forensic procedures. The use of virtualized environments ensures that results are not affected by external variables, such as unrelated background processes or inconsistent hardware behavior, thereby improving the validity of the forensic findings.

4.4 TOOLS AND SOFTWARE USED

This work resorted to several software tools to extract and analyze data from files associated with the password managers for analysis. Table 3 presents the tools used and their respective versions.

Tool	Version
Clip View(*Clip View* 2025)	N/A
Hashcat(*Hashcat* 2024)	6.2.6
HxD(*HxD* 2024)	2.5.0.0
John the Ripper(*openwall/john* 2025)	1.9.0
Python(*Python* 2024)	3.12.8
Sqlitebrowser(*Sqlitebrowser* 2024)	3.13.1
strings (GNU)(*strings* 2024)	2.40
System Informer(*System Informer* 2025)	3.1.24298
VirtualBox(*VirtualBox* 2024)	7.0.22 r165102 (Qt5.15.2)
xclip(*xclip* 2025)	0.13
ChatGPT(*ChatGPT* 2025)	GPT-4.5-turbo

Table 3: List of tools used for the work presented in this report

Additionally *ChatGPT* (2025) was used to assist with proofreading and to generate writing suggestions during the preparation of this thesis.

4.5 FORENSIC ANALYSIS WORKFLOW

The forensic analysis process followed a structured and repeatable methodology for each password manager under investigation. The objective was to identify and extract any residual data or artifacts that could compromise the confidentiality or integrity of the information managed by these applications. The workflow consisted of the following steps:

- **Documentation and Source Code Review**

  Before beginning any practical analysis, the available official documentation and the source code of the application were reviewed. This provided insights into the application's architecture, encryption mechanisms, data storage practices, and configuration files. In open-source applications, such as *Bitwarden - GitHub page* (2025) and *KeePassXC - GitHub page* (2025), this step was especially valuable in identifying potential locations of sensitive data or cryptographic artifacts.

- **File System Analysis**

  After installing and using the password manager under normal conditions, the system's file structure was examined. This involved locating application-specific

directories and configuration files (`e.g., data.json`, SQLite databases, or LevelDB files). Tools such as HxD, Sqlitebrowser, and Python scripts were used to inspect the contents of these files and extract readable information such as email addresses, KDF parameters, and master password hashes.

- **Browser Extension Inspection**

  For password managers with browser extensions, the same analysis process was applied to both the Firefox and Chrome versions. The respective directories used by these browsers to store extension data were identified, and database or local storage files were analyzed. In many cases, decoding or decompressing stored data was necessary to access the relevant information.

- **Memory Dump Analysis**

  The next step involved analyzing the contents of volatile memory. Memory dumps of the password manager processes were created while the application was both in an unlocked and locked state. Tools such as System Informer, `strings`, and custom Python scripts were used to search for sensitive information in memory, including unencrypted credentials, master passwords, or cryptographic keys.

- **Hash Extraction and Cracking Attempts**

  If a password hash was found (typically in a file such as `data.json`), it was extracted and prepared for use with cracking tools. Hashcat was primarily used for this purpose, employing dictionary attacks with custom-generated wordlists based on data found during memory analysis. When applicable, John the Ripper was also tested. In cases where Argon2 was used (such as Bitwarden with Argon2id), limitations were noted due to lack of support in Hashcat for that KDF.

4.6 ETHICAL CONSIDERATIONS AND LIMITATIONS

All experiments conducted during this study were performed using test accounts and simulated environments. No real user data was accessed, stored, or compromised at any point in this research. The tools and techniques applied, like memory dumps, hash extraction, and password cracking, were exclusively used on self-created and controlled data.

Any identified vulnerabilities or insecure practices were analyzed purely for academic purposes and not exploited in any malicious context.

Despite the methodological rigor, several limitations were encountered during the forensic analysis:

- Hashcat does not support Argon2id, the key derivation function used by some password managers (Bitwarden), which limited the ability to test cracking effectiveness for those configurations.
- Some browser session data was stored in compressed formats (*jsonlz4* (2025) in Firefox), making extraction difficult without additional decompression tools.
- Memory analysis was time-sensitive, as some sensitive information (master passwords) would only remain in RAM for short periods after locking the vault.
- The results may vary depending on the operating system, application version, or hardware configuration, which limits the generalizability of the findings to other environments.

These constraints are acknowledged in the interpretation of results and are discussed further in the final chapters of this work.

In the next Chapter we analyse Bitwarden while KeePassXC is studied in Chapter 6.

5 BITWARDEN ANALYSIS

Bitwarden is a widely used cloud-based password manager, with over five million downloads on Google Play alone (*Bitwarden - Password Manager – Apps on Google Play* 2025).

This analysis focuses primarily on its client-side components, including the standalone desktop application, the Firefox extension, and the web vault accessed via browser. Due to its cloud-centric and open-source nature, Bitwarden can be self-hosted for testing. Following the official Bitwarden documentation (*Help Center* 2024), a server was deployed in an Ubuntu Server environment.

Figure 3 shows a diagram that illustrates how various Docker containers work together in Bitwarden's self-hosted deployment to provide secure password management. Clients communicate through the reverse proxy, core services handle authentication and data storage, while additional services manage icons, notifications, and attachments. In our testbed, certain settings such as SMTP configuration, were not set up since they are not critical for Bitwarden's core functionality.

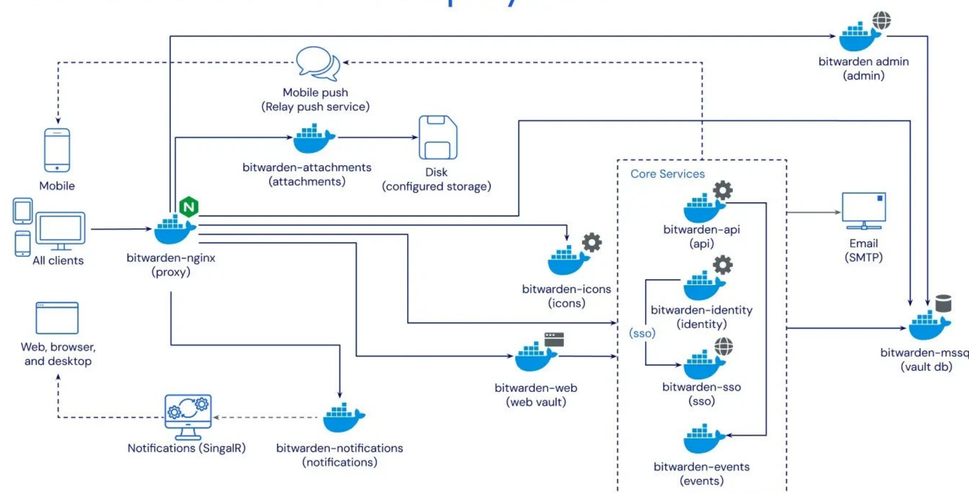


Figure 3: Diagram of Bitwarden self-host deployment(*Bitwarden Blog* 2024)

This setup enabled the use of the web vault in the browser and the browser extension. However, due to the use of a self-signed certificate for HTTPS communication, the standalone application was initially non-functional. To resolve this issue, a valid certificate was generated using *Let's Encrypt* (2024).

As mentioned by Aas et al. (2019), Let's Encrypt is a free, automated, and open certificate authority (CA) that provides SSL/TLS certificates to help secure websites with HTTPS. Unlike traditional CAs that charge for certificates, Let's Encrypt offers them at no cost and automates the entire process, including issuance and renewal, through tools like Certbot. This automation allows website owners to easily secure their sites without the need for manual intervention. Let's Encrypt verifies domain ownership through simple challenges, such as proving control over a specific domain via DNS or HTTP. Its certificates are widely trusted by major browsers, contributing to the growing prevalence of encrypted connections across the web. To create a Let's Encrypt certificate in this case, the process used a DNS challenge, which involves proving domain ownership by adding a specific verification string to the domain's DNS records. Let's Encrypt provides the user with a unique string, and the user must create a DNS TXT record containing that string for the domain they are trying to secure. Once the TXT record is set up, Let's Encrypt checks the DNS to verify that the string matches, confirming that the user has control over the domain.

With the self-hosted Bitwarden server functioning as intended, two test accounts were created using the email addresses **teste@mail.pt** and **teste2@mail.pt**. Each account was secured with a master password that met Bitwarden's requirements.

In Table 4 are listed Bitwarden's version for each application.

Application	**Version**
Desktop App	2024.11.2
Bitwarden Browser Extension	2024.11.2
Bitwarden self-host	2024.10.2

Table 4: List of the Bitwarden application versions.

With all preparations complete, the necessary applications for the analysis were ready for use and testing.

5.1 PASSKEYS

Passkeys are an alternative to traditional passwords, and Bitwarden supports this form of authentication. With passkeys, users can access their vaults without needing to enter their email address or master password. However, this feature is currently in beta for Bitwarden (*Bitwarden Passkey Log In* 2024).

Passkey functionality is limited to PRF-capable browsers. As can be seen in *PRF WebAuthn - Bitwarden Blog* 2024, a PRF-capable browser supports the PRF extension for WebAuthn, which allows security authenticators, such as hardware keys, to generate consistent encryption keys tied to specific websites or services identified as relying parties in passkey parlance. This mechanism uses a salt provided by the relying party to deterministically derive the same key without storing it on the device. For instance, Bitwarden can use a passkey registered with a security key to create an encryption key for securing user vaults. Unlike traditional FIDO2 signatures, which vary with each operation, the PRF extension generates predictable outputs, making it suitable for encryption tasks without requiring direct key storage, such as in Hardware Security Modules (HSM).

This feature was tested, but in the absence of a compatible authenticator, the vault required the master password after using the passkey to unlock it, defeating the purpose of using a passkey for seamless access.

Bitwarden also supports storing passkeys for authentication on other services within its vault. To store a passkey, users simply need to visit a login page that supports passkeys, and the Bitwarden extension will automatically detect and save the passkey in the vault.

By using the passkey demonstrating site, *WebAuthn.io* 2024, to create and store a passkey, and subsequently exporting the vault in unencrypted format, it is possible to observe how the passkey is stored. This is demonstrated in Listing 1.

```
        {
            "passwordHistory": null,
            "revisionDate": "2024-11-22T11:29:19.933Z",
            "creationDate": "2024-11-22T11:29:19.876Z",
            "deletedDate": null,
            "id": "fd840890-91dc-4e03-a54b-b23000bd549d",
            "organizationId": null,
            "folderId": null,
            "type": 1,
            "reprompt": 0,
            "name": "webauthn.io",
            "notes": null,
            "favorite": false,
            "login": {
              "fido2Credentials": [
                {
                  "credentialId": "71cc1f14-775d-4598-b374-26dd6f301dae",
                  "keyType": "public-key",
                  "keyAlgorithm": "ECDSA",
                  "keyCurve": "P-256",
                  "keyValue": "MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgw
                  ↪  GwtHAZbz1k8NufsDazbOmB7eSj0eSg_tMKms0QnFcahRANCAATvPmgZcYs
                  ↪  Gq5_IsJ_Vn9Y780sT4q8i0jdMTl8Z3fE5HTFc2XAvd6dD6qPBYm0xHZduF
                  ↪  jAX3ZCwt3NlwhUsEKfc",
                  "rpId": "webauthn.io",
                  "userHandle": "MlFuV0ZsR0hMMHZfNU5XU2wzTUlJMFdNSVBwQTc3ck12Rk8
                  ↪  4VnZWU3d3cw",
                  "userName": "passkey_bitwarden",
                  "counter": "0",
                  "rpName": "webauthn.io",
                  "userDisplayName": "passkey_bitwarden",
                  "discoverable": "true",
                  "creationDate": "2024-11-22T11:29:19.880Z"
                }
              ],
              "uris": [
                {
                  "match": null,
                  "uri": "https://webauthn.io/"
                }
              ],
              "username": "passkey_bitwarden",
              "password": null,
              "totp": null
            },
            "collectionIds": null
          }
```


Listing 1: Bitwarden‘s vault export containing a passkey

In this export we can see a passkey entry for the webauthn.io service, containing a WebAuthn credential used for secure login. The credential includes a public key and is associated with the user "passkey_bitwarden". The passkey is tied to the domain webauthn.io, and the credential is marked as discoverable, meaning it can be used without entering a username. It has a unique credential ID and is configured for FIDO2-compliant authentication. The entry was created and last revised on November 22, 2024, and includes metadata such as creation date, revision date, and a null password field, indicating it relies on the passkey for authentication rather than a traditional password.

5.2 CLIPBOARD

A common functionality among password managers is the use of the system Clipboard to allow users to copy and paste login credentials into the desired location.

Bitwarden implements this feature as well. To analyze its behavior, we examined the source code and identified the snippet shown in *clipboard.rs* (2025). The relevant portion of the code is displayed in Listing 2.

```rust
// Exclude from windows clipboard history
#[cfg(target_os = "windows")]
fn clipboard_set(set: Set, password: bool) -> Set {
    use arboard::SetExtWindows;
    if password {
        set.exclude_from_cloud().exclude_from_history()
    } else {
        set
    }
}
// Wait for clipboard to be available on linux
#[cfg(target_os = "linux")]
fn clipboard_set(set: Set, _password: bool) -> Set {
    use arboard::SetExtLinux;
    set.wait()
}
#[cfg(target_os = "macos")]
fn clipboard_set(set: Set, password: bool) -> Set {
    use arboard::SetExtApple;
    if password {
        set.exclude_from_history()
    } else {
        set
    }
}
```

Listing 2: Source code from Bitwarden demonstrating Clipboard usage

This code is responsible for managing Bitwarden's interaction with the operating system's Clipboard. It utilizes the `arboard` library to handle clipboard operations across different platforms, its document can be found in *Clipboard in arboard - Rust* (2025). Furthermore, Bitwarden leverages this library to prevent copied credentials from being stored in Clipboard history, thereby enhancing user privacy and security by circumventing clipboard managers that log copied content.

To evaluate this functionality, we conducted tests in a Linux environment using the `xclip` utility to inspect the contents of the system Clipboard. As shown in Figure 4, the credentials copied by Bitwarden are retrievable in plain text.

A similar experiment was performed on Windows using the Clip View tool. As illustrated in Figure 5, the results mirrored those observed in the Linux test: the copied value remained accessible in plain text.

```
└──╼ $xclip -selection clipboard -o
teste-password┌─[parrot@parrot]─[~]
└──╼ $
```

Figure 4: Retrieving Clipboard contents using xclip in Linux

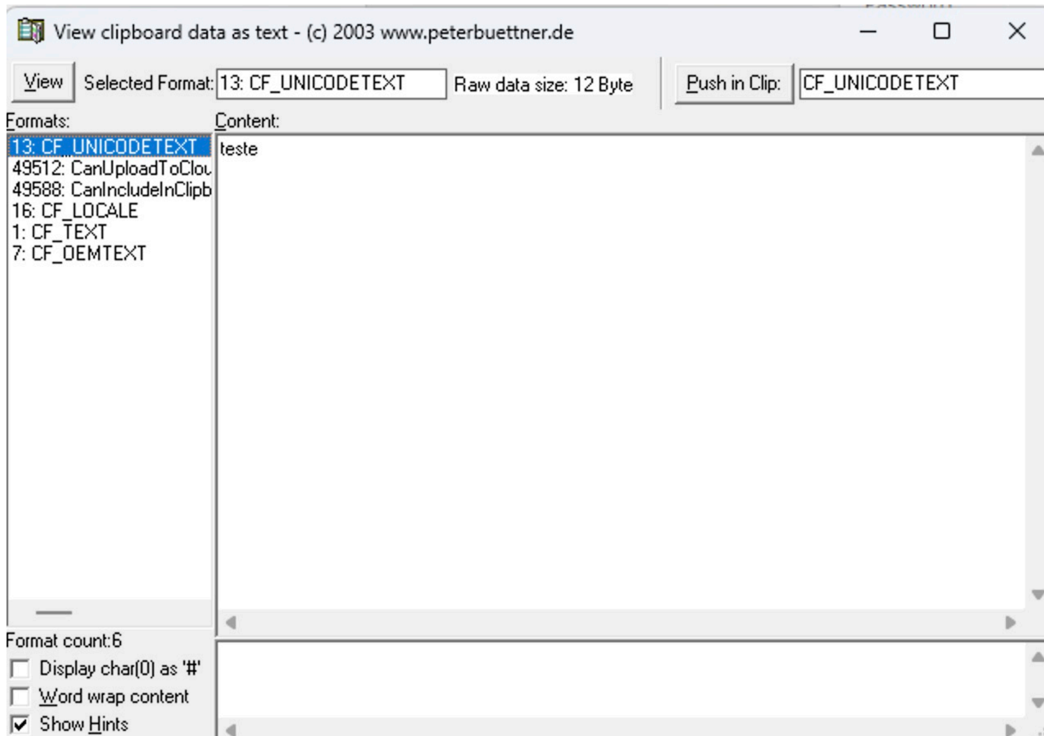


Figure 5: Retrieving Clipboard contents using Clip View in Windows

It is important to note that this feature is not designed to encrypt credentials in the Clipboard, as such an operation must be fast and seamless for the user. However, Bitwarden attempts to reduce the associated security risks by bypassing clipboard history, thus providing a basic level of protection against clipboard managers. Additionally, many password managers, including Bitwarden, implement automatic Clipboard clearing after a configurable timeout period. Nevertheless, this option was not enabled by default in the version used during our tests. This can be seen in the Figure 6.

5.3 FILE ANALYSIS

Bitwarden ensures that all data is encrypted and hashed locally on the user's device before being transmitted to cloud servers for storage. The Bitwarden servers function

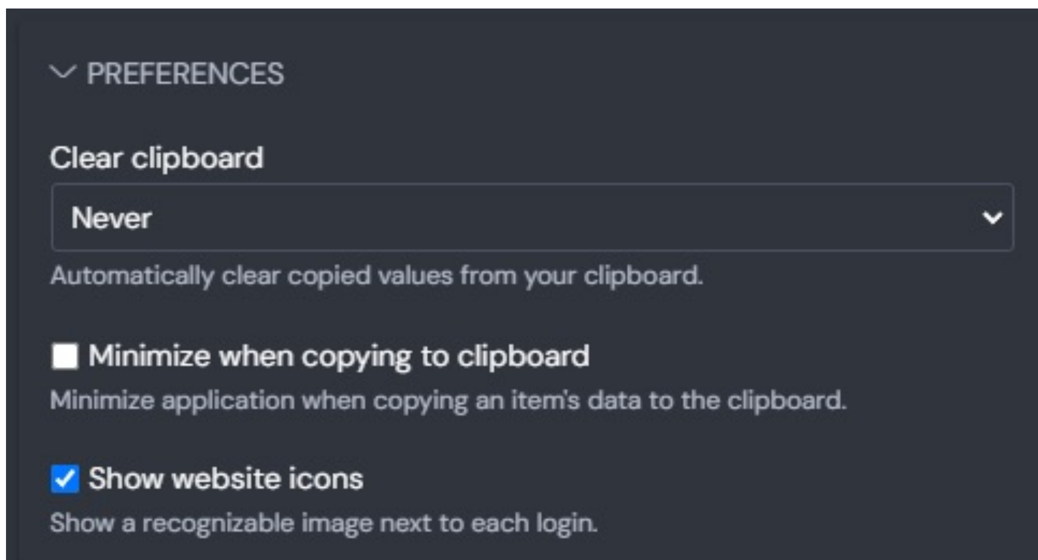


Figure 6: Bitwarden default clear clipboard setting

solely as a repository for encrypted data. Any data stored on the user's device is encrypted and remains so until the vault is unlocked. Decrypted data is retained in memory only and is never written to persistent storage, ensuring enhanced security and privacy.

With this in mind, it is prudent to investigate any files created or left behind by Bitwarden's applications to determine whether they contain information that could potentially compromise the security of the passwords stored in Bitwarden's vault. By following *Storage | Bitwarden Help Center* (2024), we learn where these applications store any files.

This section is divided into three parts, each focusing on one of the Bitwarden applications analyzed.

5.3.1 *Desktop app*

Starting with the desktop app, in the Windows environment, relevant files can be found in "`%AppData%\Bitwarden`", while in Linux, they are stored in "`~/.config/Bitwarden`".

Upon examining these directories in both environments, a json file named data.json was identified. This file contains significant information that can be extracted, including metadata such as:

- Bitwarden server information: This includes the server domain the application is connected to and the version of Bitwarden running on that server, which can be highly valuable in a real forensic analysis.
- User details: The name, email address currently logged into the application, and the user ID.
- Vault configurations: Information such as the type of hash function being used and the number of iterations applied for key derivation.
- Master password hash: A critical field named USERID_masterPassword _masterKeyHash contains the hash of the master password, as the name suggests.

Storing such sensitive information can pose significant risks. A malicious actor with access to this file could potentially exploit it to crack the master password hash. By default, Bitwarden uses the PBKDF2_SHA256 hashing algorithm for vault key derivation instead of the more robust Argon2. This can be verified by examining the kdfType field in the data.json file, which shows a value of 0. This can be seen in Listing 3.

```
{
"user_929583ab-1fc1-40e5-b944-b227013733ee_kdfConfig_kdfConfig": {
    "kdfType": 0,
    "iterations": 600000
}
```

Listing 3: Bitwarden‘s kdfType in data.json

The source code available at *Bitwarden Client's source code* (2024) supports the interpretation of these values. As shown in Figure 7

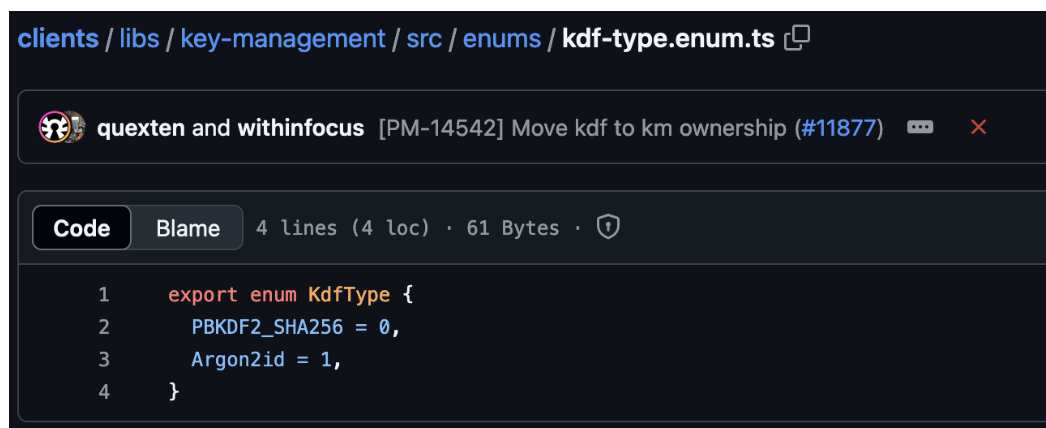


Figure 7: Source code from Bitwarden's Github

Bitwarden's default implementation of PBKDF2 ensures a higher level of security by setting the number of iterations to 600 000. If the user attempts to lower this number, Bitwarden provides a warning.

Hashcat includes a Python script, the `bitwarden2hascat.py`, in its tools directory designed for this exact purpose. The script can be found in the Hashcat repository at *bitwarden2hashcat.py · GitHub* (2024).

This script is specifically tailored to extract information from files of this type to enable the use of a dedicated Hashcat module for cracking Bitwarden master passwords secured with PBKDF2.

An analysis of the script reveals its functionality: it extracts the key hash, the associated email address, and the iteration count from the input file. Furthermore, the script is adaptable to different file types. For instance, it processes JSON files from the desktop application and SQLite files from the Bitwarden browser extension.

Attempting to use this script on the data.json file resulted in errors, likely because the script has not been updated in over two years, and the internal formatting of the `data.json` file has since changed.

To address this issue, based on the original Hashcat script we can reproduce the steps by extracting manually the data from the file data.json. Giving us the result seen in the Listing 4 below.

```
1             "email": "teste@mail.pt",
2             "kdfType": 0,
3             "iterations": 600000
4            "USERID_masterPassword_masterKeyHash":
           ↪  t0UxjFjC0nYQoL0jdBO+WGYEhS3aZofobOyxjkYSAeg=
```

Listing 4: Results of extracting the data manually from data.json

5.3.2 *Browser extension*

The analysis included both the Google Chrome and Firefox Bitwarden extensions to represent the broadest possible scope. Table 5 lists the browsers used along with their respective versions.

In Figure 8 we can see the extension user interface.

Browser	Version
Firefox	133.0.3
Google Chrome	131.0.6778.204

Table 5: List of the Web Browsers used and their versions.

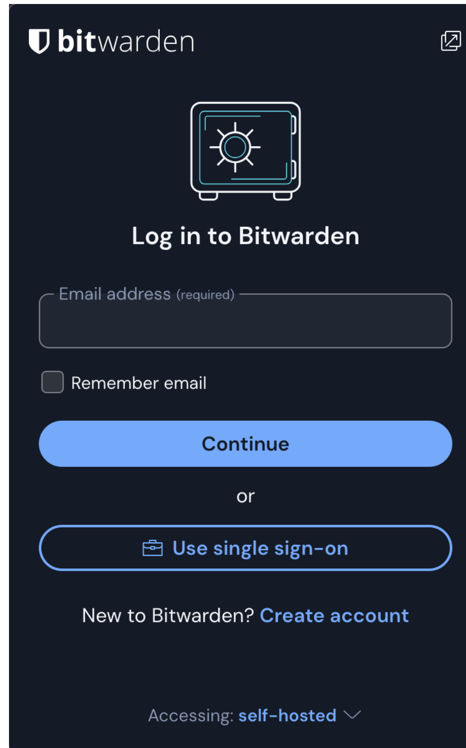


Figure 8: Bitwarden Extension

FIREFOX. Getting to the directory where Bitwarden's Firefox extension is a bit more complicated as we need to know what is the UUID of the extension. The directory path for both Windows and linux can be seen in Table 6.

Operating System	Storage Path
Windows	`%AppData%\Mozilla\Firefox\Profiles\your_profile\storage\default\moz-extension+++UUID^userContextId=integer`
Linux	`~/.mozilla/firefox/your_profile/storage/default/moz-extension+++UUID^userContextId=integer`

Table 6: Firefox Storage Paths on Windows and Linux

In this directory, a SQLite database file was also discovered, which contains valuable information. Using the tool SQLiteBrowser, it is possible to navigate

through the structure of this database file. However, much of the data is encoded in hexadecimal, and decoding it reveals that almost all data is unreadable.

Once again, attempting to use the Hashcat script to process this file resulted in errors. To address this issue, a new script named, `sql_extract.py`, was developed, inspired by the original Hashcat script, specifically designed to extract relevant data from this `.sqlite` file. The script made can be seen in Listing 5 and the source code can be seen in *Bitwarden-Forensic-Tools/sql_extract.py* (2025).

```
import sqlite3
import snappy  # Library to handle data compression/decompression
import json
import re
# Connect to the SQLite database
# Replace "3647222921wleabcEoxlt-eengsairo.sqlite" with your actual database file
database_path = "3647222921wleabcEoxlt-eengsairo.sqlite"
connection = sqlite3.connect(database_path)
cursor = connection.cursor()
decompressed_data = []
# Query all rows from the `object_data` table
data = cursor.execute("SELECT * FROM object_data;").fetchall()
aux= len(data)
print("length of data:" , aux)
# Loop through the first 45 rows of the result set
for row_index in range(aux):  # Adjust range as needed for your dataset
    # Decompress the data from the 5th column (index 4) of each row
    decompressed_data.append(snappy.decompress(data[row_index][4]))
# Convert the decompressed data list to a single string
decompressed_data_str = str(decompressed_data)
print(decompressed_data_str)
iterations_segment = decompressed_data_str[
    decompressed_data_str.find('iterations') - 1:
    ↪  decompressed_data_str.find('iterations') + 18
]
print(iterations_segment)
# Extract a section around the keyword "email"
email_segment = decompressed_data_str[
    decompressed_data_str.find('email') - 1: decompressed_data_str.find('email')
    ↪  + 22
]
print(email_segment)
# Extract a section around the keyword "kdfType"
kdf_type_segment = decompressed_data_str[
    decompressed_data_str.find('kdfType') - 1:
    ↪  decompressed_data_str.find('kdfType') + 10
]
print(kdf_type_segment)
pattern = r'"(.{44})"' #the key is composed of 44 characters and its encclosed in
↪  double quotes so to find it we can use this pattern to out advantage
matches = re.findall(pattern, decompressed_data_str)
print("44-character words enclosed in double quotes (possible master key hash):")
for match in matches:
    print(match)
connection.close()
```

Listing 5: Python script for extracting data from the sqlite file

By utilizing the Snappy module, a significant portion of the database's contents was successfully extracted in a readable format. *Snappy* (2025) is a compression

and decompression library designed for high-speed performance. It appears that certain data in the database was compressed, therefore snappy was used to make unreadable portions of the file accessible and readable.

Following this, the script searched for specific instances of fields such as "iterations", "email", and "kdf type" to retrieve their associated values.

Since no field named "`USERID_masterPassword_masterKeyHash`" was found, the script instead performed a comprehensive search across the entire output for potential candidates. A list of these candidates was then displayed to the user for manual selection. The key hash, being distinct, was expected to stand out among the listed candidates, simplifying identification. This method is not foolproof nor the more recommended but for testing purposes will suffice.

GOOGLE CHROME. The process differs slightly for Chrome compared to Firefox. The storage path of the files associated with the Bitwarden extension is detailed in Table 7.

Operating System	**Storage Path**
Windows	`%LocalAppData%\Google\Chrome\User Data\Default\Local Extension Settings\nngceckbapebfimnlniiiahkandclblb`
Linux	`~/.config/google-chrome/Default/Local Extension Settings/nngceckbapebfimnlniiiahkandclblb`

Table 7: Chrome Extension Storage Path on Windows and Linux

Unlike Firefox, this directory does not contain a straightforward database file for analysis. An example of what can be found in this directory can be seen in Figure 9

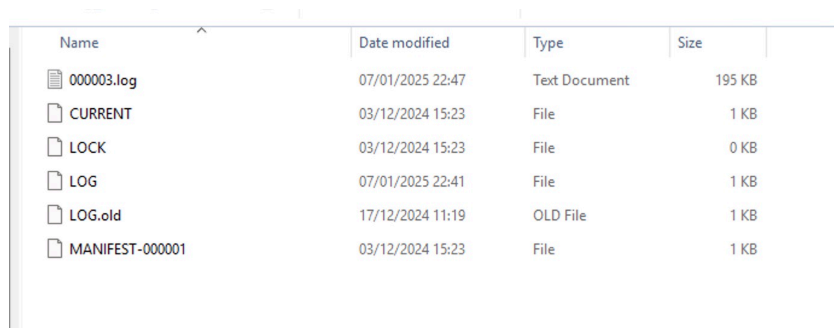


Figure 9: Contents in Google Chrome’s directory for Bitwarden extension

Instead, Chrome relies on LevelDB for data storage, as can be seen in *More efficient IndexedDB storage in Chrome* (2025). LevelDB, an open-source key-value storage library developed by Google, offers fast and efficient data access by providing an ordered mapping of string keys to string values. Its GitHub repository can be found here *LevelDB* (2025).

According to *Demystifying LevelDB* (2019), LevelDB structures its data storage using a Log-Structured Merge-Tree (LSM) architecture. This architecture primarily involves three main components: Sorted String Table (SST) files, log files, and memtables. When data is written, it is first appended to a `.log` file and simultaneously stored in a memtable (a memory-based structure). Once the `.log` file reaches a size limit (typically 4 MB), its data is converted into an SST file and new `.log` files and memtables are created to continue the process. LevelDB organizes and tracks data using a `MANIFEST` file, which holds metadata about the current SST files. This file is referenced by the `CURRENT` file. During read operations, LevelDB identifies the relevant SST files using the `MANIFEST` file and retrieves data by reconciling the information in the SST files with the memtable, while also accounting for overwrites and deletions. Additionally, all operations are logged in a separate `LOG` file to ensure reliability.

With this understanding, a script named `chrome_extension_data_parser.py` was developed to automatically extract information from the Chrome extension's directory or a user-specified directory. This script utilizes the `plyvel` library, which provides an interface for interacting with LevelDB databases. By leveraging this library, the script opens the database and queries it for relevant information. For further details about `plyvel`, refer to its official documentation at *Plyvel - documentation* (2025).

The code for the script `chrome_extension_data_parser.py` can be seen in Listing 6 and its source code can be found in *Bitwarden-Forensic-Tools/chrome_extension_data_parser.py* (2025).

```
1  import os
2  import plyvel
3  import re
4  import json
5  import platform
6  try:
7      # Detect the operating system
8      is_windows = platform.system().lower() == "windows"
9
10     # Default paths
```

```
11      if is_windows:
12          default_path = os.path.join(
13              os.getenv("userprofile"),
14              "AppData", "Local", "Google", "Chrome", "User Data",
15              "Default", "Local Extension Settings",
                ↪  "nngceckbapebfimnlniiiahkandclblb"
16          )
17      else:
18          default_path = os.path.join(
19              os.getenv("HOME"),
20              ".config", "google-chrome", "Default",
21              "Local Extension Settings", "nngceckbapebfimnlniiiahkandclblb"
22          )
23      # Let the user choose between default or custom path
24      print(f"Default database path: {default_path}")
25      use_default = input("Do you want to use the default path? (yes/no):
        ↪  ").strip().lower()
26      if use_default == "no":
27          db_path = input("Enter the full path to the LevelDB database: ").strip()
28          if not os.path.exists(db_path):
29              raise FileNotFoundError(f"The specified path does not exist:
                ↪  {db_path}")
30      else:
31          db_path = default_path
32      # Open the LevelDB database
33      try:
34          db = plyvel.DB(db_path, create_if_missing=False)
35      except Exception as e:
36          raise FileNotFoundError(f"Unable to open LevelDB at {db_path}. Ensure the
            ↪  path is correct. Error: {e}")
37      # Step 1: Extract the User ID
38      try:
39          # Retrieve the active account ID from the database
40          user_id_raw = db.get(b'global_account_activeAccountId').decode()
41          # Split the raw data to isolate the user ID
42          user_id_parts = user_id_raw.split("\"")
43          flattened_string = ' '.join(user_id_parts)
44          # Use a regular expression to extract the UUID format UserID
45          user_id_match = re.search(r'\b[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-
            ↪  f]{4}-[0-9a-f]{12}\b',
            ↪  flattened_string)
46          if not user_id_match:
47              raise ValueError("UserID not found in the database.")
48          user_id = str(user_id_match.group(0))
49          print("UserID:", user_id)
50      except Exception as e:
51          raise RuntimeError(f"Failed to extract UserID. Error: {e}")
52      # Step 2: Extract kdfType and Iterations
53      try:
54          # Construct the query to retrieve the KDF configuration
55          kdf_query = f'user_{user_id}_kdfConfig_kdfConfig'.encode()
56          kdf_config_raw = db.get(kdf_query).decode()
57          # Parse the KDF configuration JSON
58          kdf_config_json = json.loads(kdf_config_raw)
```

```
59         nested_kdf_config = json.loads(kdf_config_json['value'])
60         # Extract KDF type and iterations
61         kdf_type = nested_kdf_config['kdfType']
62         iterations = nested_kdf_config['iterations']
63         print("KDF Type:", kdf_type)
64         print("Iterations:", iterations)
65     except Exception as e:
66         raise RuntimeError(f"Failed to extract KDF Type and Iterations. Error:
           ↪  {e}")
67     # Step 3: Extract the user's email
68     try:
69         # Retrieve the account information from the database
70         accounts_raw = db.get(b'global_account_accounts').decode()
71         accounts_json = json.loads(accounts_raw)
72         nested_accounts = json.loads(accounts_json['value'])
73         # Extract the email address associated with the user ID
74         email = nested_accounts[user_id]['email']
75         print("Email:", email)
76     except Exception as e:
77         raise RuntimeError(f"Failed to extract email. Error: {e}")
78     # Step 4: Extract the Master Key Hash
79     try:
80         # Construct the query to retrieve the Master Key Hash
81         master_key_query =
           ↪  f'user_{user_id}_masterPassword_masterKeyHash'.encode()
82         master_key_raw = db.get(master_key_query).decode()
83         # Parse the Master Key Hash JSON
84         master_key_json = json.loads(master_key_raw)
85         # Extract and clean up the Master Key Hash value
86         master_key_hash = master_key_json['value'].replace('"', '')
87         print("Master Key Hash:", master_key_hash)
88     except Exception as e:
89         raise RuntimeError(f"Failed to extract Master Key Hash. Error: {e}")
90 except Exception as e:
91     print(f"An error occurred: {e}")
92 finally:
93     # Ensure the database is closed properly
94     try:
95         db.close()
96     except Exception:
97         pass
```

Listing 6: Python script ,`chrome_extension_data_parser.py`, for extracting data from the LevelDB

The script retrieves user-related data from the LevelDB database, such as the user ID, email, kdfType, iterations, and the master key hash. It detects the operating system to either provide a default database path or allow the user to input a custom path. The script accesses specific database keys, parses nested JSON data, and extracts the necessary values. These extracted values can later be used in Hashcat for further analysis or cracking as it can be seen in later chapters of this work.

5.3.3 *Web application*

Extracting information from the web vault differs slightly from other methods. To achieve this, access to both the local storage and session storage of the browser is required.

When the vault is locked, the local storage can be examined, as shown in Figure 10. From this storage, information such as the account's name, email address associated with the locked vault, and the server domain can be extracted.

It is important to note that no instance of the master key hash field is present in the local storage.

Filter Items

Key	Value
929583ab-1fc1-40e5-b944-b227013733ee	{"keys":{"cryptoSymmetricKey":{}},"profile":{"userId":"929583ab-1fc1-40e5-b944-b227013733ee","name":"teste","email":"teste@mail.pt"}}

Figure 10: Local storage of Bitwarden's locked web vault

By unlocking the vault, it is also possible to analyze the session storage available. As expected, this storage contains much more critical information. Upon examining the session storage, familiar fields such as "`kdfType`", "`iterations`", and "`USERID_masterPassword_masterKeyHash`" can be observed. The information extracted from session storage is similar to that obtained from the `data.json` file; however, this method is somewhat more intrusive.

A notable limitation of session storage is that it functions as a temporary file. This means that the information stored in it is cleared when the vault is locked or when the browser is closed and reopened, unless the user unlocks the vault again.

This behavior was tested on both Firefox and Chrome, and the results were consistent across both browsers. An example of the session storage structure is presented in Figure 11.

<table>
<tr><th>global_router_deepLinkRedirectUrl</th><th>"/vault"</th></tr>
<tr><td>stateVersion</td><td>68</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_billing_accountProfile</td><td>{"hasPremiumPersonally":false,"hasPremiumFromAnyOrg</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_crypto_everHadUserKey</td><td>true</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_crypto_organizationKeys</td><td>{}</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_crypto_privateKey</td><td>"2.klmrPaNdfLE85RhSX8r6xw==|BMTdc5xJxNCJVLFys+p2</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_crypto_providerKeys</td><td>{}</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_domainSettings_equivalentDomains</td><td>[]</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_environment_environment</td><td>{}</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_kdfConfig_kdfConfig</td><td>{"kdfType":0,"iterations":600000}</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_keyConnector_usesKeyConnector</td><td>false</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_masterPassword_masterKeyEncryptedUserKey</td><td>"2.57/IlpLPbLYs2AN4Gpwv6g==|/gmC02kinuDee/JpTSjU</td></tr>
<tr><td>user_929583ab-1fc1-40e5-b944-b227013733ee_masterPassword_masterKeyHash</td><td>"t0UxjFjC0nYQoL0jdBO+WGYEhS3aZofobOyxjkYSAeg="</td></tr>
</table>

Figure 11: Session storage of Bitwarden's unlocked web vault

```
python -c "import lz4.block; f=open('sessionstore.jsonlz4','rb'); f.read(8);
↪  d=lz4.block.decompress(f.read()); open('output.json','wb').write(d)"
```

Listing 7: Python command to open .jsonlz4 file

The persistence of Firefox session storage data subsequent to browser closure is facilitated by a backup mechanism that saves some session information within the user's profile directory which could have some leftover information that could be useful for a forensic analysis. These backup files are compressed using the non-standard jsonlz4 format, necessitating a specific approach for data extraction. To overcome this, the Python command presented in Listing 7 was utilized on the files located within:

- %APPDATA%\Mozilla\Firefox\Profiles\<your_profile>\sessionstore-backups for Windows systems.
- ~/.mozilla/firefox/<your_profile>/sessionstore-backups/ for Linux systems.

Unfortunately the contents of these files were not helpful for our analysis, as no information regarding the session storage of Bitwarden was found. An example of the content found in these files can be seen in Listing 8.

```
{
    "host": ".bitwarden.com",
    "value": "DGagP_...",
    "path": "/",
    "name": "_cfuvid",
    "secure": true,
    "httponly": true,
    "originAttributes": {
    "firstPartyDomain": "",
    "geckoViewSessionContextId": "",
    "partitionKey": "",
    "privateBrowsingId": 0,
    "userContextId": 0
    },
    "schemeMap": 2
},
```

Listing 8: Content found inside the .jsonlz4 files

5.4 MEMORY

This section explores a potential method for extracting information by directly reading the memory used by Bitwarden.

Bitwarden never writes decrypted data to persistent storage. Instead, the application only decrypts data in volatile memory. The decrypted data is stored exclusively in memory or, in the case of a self-hosted Bitwarden instance, on the server (e.g., in the Ubuntu server environment).

For example, when using the extension or desktop app on Windows, performing a memory dump of the process via the Task Manager allows access to the contents of the vault. By analyzing the memory dump, it is possible to view all the data stored in the vault. This behavior is expected, as the vault is unlocked, and the only location where this data could reside would be in memory. An example of this can be seen in Listing 9.

```
1         login": {"username" : "memory-test" , "password" : "mymasterpassword"
```

Listing 9: Small example of the memory dump of Bitwarden's Desktop App while the vault is unlocked

After identifying related issues on GitHub *Issue #6231 · bitwarden/clients* 2024 and *Issue #1516 · bitwarden/clients* 2024, several tests were conducted. Specifically, for the Firefox extension, after unlocking the vault and subsequently locking it, performing a memory dump of the Firefox process revealed instances of the master password stored in clear text.

This behavior is likely unintentional, as Bitwarden explicitly guarantees memory security only when the vault is locked or the user is logged out. However, in this scenario, despite the vault being locked, traces of the master password in clear text could still be found where they should ideally be cleared from memory. Dump files of Firefox, where the Bitwarden extension was running, were created in both the Windows and Linux environments. To analyze these dump files, the `strings` command was used to extract any readable text from the files.

To conduct further testing after locking the vault, experiments were performed to determine how long the master passwords remained in memory.

In the Linux environment, the command-line represented in Listing 10 was used to analyze memory.

```
1 while true; do date; dumpproc -q -o - <PID> | grep -c <password>; sleep 3s; done
```

Listing 10: Command-line used for memory analysis on linux

It was observed that one minute after locking the vault, all instances of the master password were cleared from memory. However, as shown in Figure 12, Firefox

retained one instance of the master password in memory even after the vault had been locked for at least one minute.

```
┌─[parrot@parrot]─[~/Desktop/tese]
└──╼ $while true; do date; dumpproc -q -o - 1679 | grep  -c 'gxNPY486\$3p9'; sleep 3s; done
Sun 15 Dec 15:46:13 GMT 2024
dumpproc: pid 1679: Dumping memory
1
Sun 15 Dec 15:46:19 GMT 2024
dumpproc: pid 1679: Dumping memory
1
Sun 15 Dec 15:46:25 GMT 2024
dumpproc: pid 1679: Dumping memory
1
Sun 15 Dec 15:46:31 GMT 2024
dumpproc: pid 1679: Dumping memory
1
Sun 15 Dec 15:46:38 GMT 2024
dumpproc: pid 1679: Dumping memory
1
Sun 15 Dec 15:46:44 GMT 2024
dumpproc: pid 1679: Dumping memory
1
Sun 15 Dec 15:46:51 GMT 2024
dumpproc: pid 1679: Dumping memory
1
```

Figure 12: Ouput of the the shell command to dump and search for the master password every 3 seconds

For the Windows environment, similar tests were conducted, but with the use of PowerShell, requiring a different approach to analyze the created dump files. A PowerShell command was crafted to replicate the functionality of the Linux shell command. The complete PowerShell command is provided in Listing 11.

This method, while functional, proved to be highly inefficient in terms of speed. Nonetheless, it was sufficient for the task. The output of the PowerShell command was very similar to the results obtained in the Linux environment. However, during testing over a 10-minute period, Firefox continued to retain two instances of the master password in memory, even after the vault was locked.

This issue is likely due to Firefox itself, as the extension code is shared across operating systems.

```
while ($true) {
    Get-Date
    $dumpFile = "process_dump.dmp"
    $outputFile = "strings_output.txt"

    # Create the memory dump
    .\procdump -ma PID $dumpFile

    # Ensure the dump file exists before proceeding
    if (Test-Path $dumpFile) {
        # Extract readable strings from the dump file
        .\strings.exe -accepteula -nobanner -o -u $dumpFile > $outputFile

        # Search the extracted strings for the pattern
        $matches = Select-String -Path $outputFile -Pattern 'gxNPY486\$3p9' |
        ↪  Measure-Object | ForEach-Object { $_.Count }
        Write-Output "Matches found: $matches"

        # Clean up the generated files
        Remove-Item $dumpFile, $outputFile
    } else {
        Write-Output "Dump file creation failed!"
    }

    Start-Sleep -Seconds 3
}
```

Listing 11: Powershell command created for search the master password in memory dumps of Firefox

After analyzing the memory dump files, no consistent patterns were identified that would enable the straightforward extraction of the master password. However, a Python script, named `extract_masterPassword_memory.py`, was developed to process these files and generate a wordlist. The script extracts potential strings from the dump files based on criteria that could match a master password, such as a minimum length of 12 characters.

The primary method used by the script to identify potential candidates is to select strings that are repeated within the dump file. During testing, the master password consistently appeared multiple times in the memory dumps. Additionally, the script includes an optional mode to select candidates based solely on Bitwarden's password requirements, completely disregarding the frequency of repetition of each potential string in the file. It is important to note that this mode generates significantly larger wordlists.

The Python script can be found in Listing 12 and the source code is provided in *Bitwarden-Forensic-Tools/extract_masterPassword_memory.py* (2025).

```
        import re
        import sys
        def is_possible_password(line):
```

```
4            # Define criteria for a password
5            min_length = 12
6            max_length = 25  # Adjust as needed
7            # Check length
8            if not (min_length <= len(line) <= max_length):
9                return False
10           # Check if it contains a mix of letters, numbers, or symbols
11           if not re.search(r'[A-Za-z]', line) or not re.search(r'\d|\W', line):
12               return False
13           # Optional: Exclude common non-password lines (adjust as needed)
14           common_exclusions = ["null", "error", "unknown", "failure"]
15           if any(term in line.lower() for term in common_exclusions):
16               return False
17           return True
18       def find_repeated_passwords(file_path,repeatedWords):
19           line_counts = {}
20           with open(file_path, 'r', encoding='utf-8', errors='ignore') as file:
21               for line in file:
22                   line = line.strip()
23                   if is_possible_password(line):
24                       if line in line_counts:
25                           line_counts[line] += 1
26                       else:
27                           line_counts[line] = 1
28           repeated_passwords = {line: count for line, count in
             ↪  line_counts.items() if count > int(repeatedWords) }
29           return repeated_passwords
30       def save_to_wordlist(wordlist_path, repeated_passwords):
31           with open(wordlist_path, 'w', encoding='utf-8') as wordlist_file:
32               for password in repeated_passwords:
33                   wordlist_file.write(password + '\n')
34       if __name__ == "__main__":
35           if len(sys.argv) > 1:
36               file_path = sys.argv[1]
37           else:
38               # Ask the user for a file path if none is provided
39               file_path = input("Enter the path to the text file: ").strip()
40           wordlist_path = "wordlist.txt"
41           try:
42               repeatedWords = input("How much should a word be repeated in the
                 ↪  dump file to appear in the wordlist?(0 is recomended but will
                 ↪  create a bigger wordlist): ").strip()
43               repeated_passwords =
                 ↪  find_repeated_passwords(file_path,repeatedWords)
44                   # Save results to a wordlist file
45               save_to_wordlist(wordlist_path, repeated_passwords)
46               with open(wordlist_path, 'r') as fp:
47                   lines = len(fp.readlines())
48               print(f"\nA wordlist was created and saved to {wordlist_path} and
                 ↪  has {lines} total words")
49           except:
50               print("Error - Did you write the name of the file correctly?")
```

Listing 12: Python script to create a wordlist based of the dump files created

After running this script, a wordlist containing all potential strings is generated. This wordlist can then be used with Hashcat, as demonstrated in the next section, provided the remaining necessary information for utilizing Hashcat's Bitwarden mode is available.

5.5 CRACKING THE HASH

With the necessary information extracted from the applications, it becomes possible to utilize a specialized mode in Hashcat for cracking the hash.

Hashcat is a highly versatile and powerful tool for password recovery, with one of its major advantages being the ability to leverage GPU acceleration. Hashcat supports numerous "hash-modes", each designed for specific hashing algorithms. By referencing the mode with the number 23400, we find that it is specifically built to crack Bitwarden's PBKDF2 hashes.

For this test, a dictionary attack will be performed on the extracted hash. While a brute force attack is also an option, it would require several hours, if not days, to complete, depending on the complexity of the password and processing power.

First we have to construct the specific string to be able to use the mode. To do that we need to first encode the email of the Bitwarden account to Base64, after that we arrange our string like the way it is on Figure 13.

Figure 13: Example of the string needed to use Bitwarden's mode on Hashcat

With a prepared wordlist, it is now possible to attempt cracking the hash. To demonstrate this proof of concept, a wordlist containing over 100 000 entries was used, and the correct master password was included in the list. The purpose of this test is to verify whether Hashcat can successfully crack the hash when the correct password is present in the wordlist.

The Hashcat command used is presented in Listing 13.

Using Hashcat to perform the attack, we achieved a successful result. The attack lasted approximately one and a half minutes, likely due to the relatively small size of the wordlist. It is important to note that the cracking speed heavily depends on

```
1 hashcat -m 23400 $bitwarden$2*600000*2*dGVzd...wdA==*t0UxjFj...YSAeg=
  ↪ wordlist.txt
```

Listing 13: Hashcat command used for attacking the extracted hash

the GPU used and the correct password was at the bottom of the list. In this case, an RTX 3070 achieved a processing speed of 1 959 hashes per second.

Given this speed, it is unsurprising that a wordlist of this size was processed quickly. The output of the successful attack is shown in Figure 14.

```
$bitwarden$2*600000*2*dGVzdGVAbWFpbC5wdA==*t0UxjFjC0nYQoL0jdBO+WGYEhS3aZofobOyxjkYSAeg=:gxNPY486$3p9

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 23400 (Bitwarden)
Hash.Target......: $bitwarden$2*600000*2*dGVzdGVAbWFpbC5wdA==*t0UxjFjC...YSAeg=
Time.Started.....: Tue Dec 10 11:43:06 2024 (1 min, 16 secs)
Time.Estimated...: Tue Dec 10 11:44:22 2024 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (C:\Users\filip\Downloads\tese\bitwarden\wordlist.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:     1959 H/s (0.20ms) @ Accel:256 Loops:64 Thr:32 Vec:1
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
Progress.........: 150001/150001 (100.00%)
Rejected.........: 0/150001 (0.00%)
Restore.Point....: 145328/150001 (96.88%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: tamara3 -> ducker
Hardware.Mon.#1..: Temp: 57c Fan: 62% Util: 80% Core:1920MHz Mem:6800MHz Bus:16

Started: Tue Dec 10 11:43:04 2024
Stopped: Tue Dec 10 11:44:24 2024
```

Figure 14: Output of Hashcat after running a dictionary attack on Bitwarden's hash

5.6 TRYING THE ARGON2ID OPTION

This section discusses the default settings utilized by Bitwarden, specifically the implementation of the PBKDF2 key derivation function with 600 000 iterations. Additionally, to evaluate an alternative security option, we tested the Argon2id key derivation function using its default parameters, as illustrated in Figure 15.

When creating an account on Bitwarden to establish a vault, the KDF algorithm employed is PBKDF2. To utilize Argon2id, users must manually adjust the security settings. The default parameters for Argon2id are those recommended by the OWASP as of writing this, as detailed in *Password Storage - OWASP Cheat Sheet Series* 2025.

It is worth noting the default parameters provided by Bitwarden represent the minimum recommended settings. Increasing these parameters enhances the security of the vault by slowing down the decryption process, thereby making it more resistant

Higher KDF iterations can help protect your master password from being brute forced by an attacker.

For older devices, setting your KDF too high may lead to performance issues. Increase the value in small increments and test your devices.

KDF algorithm (required) Argon2id

KDF iterations (required) 3

KDF memory (MB) (required) 64

KDF parallelism (required) 4

Change KDF

Figure 15: Changing default settings of Bitwarden to Argon2id

to brute-force or dictionary attacks. In particular, adjusting the memory setting to a higher value is advisable if the decryption machine can allocate additional memory resources. Similarly, increasing the number of iterations will require more computational power, but it further strengthens security. The choice of parameter values is ultimately a matter of personal preference. However, Bitwarden issues a warning if any settings are configured below the default values.

Upon switching the KDF algorithm to Argon2id, it is expected that the values identified during the file analysis of the application, browser extension, and web application will exhibit differences. This expectation can be confirmed by replicating the prior analysis procedure. Specifically, by examining the `data.json` file, which is notably easier to read, we observe the presence of new values, as illustrated in Listing 14.

```
1    "user_e9d32a17-1336-43f0-b82d-b29a01151d69_kdfConfig_kdfConfig": {
2        "kdfType": 1,
3        "iterations": 3,
4        "memory": 64,
5        "parallelism": 4
6        }
```

Listing 14: New values found in data.json for Argon2 KDF

These new parameters are essential for computing an Argon2 hash. The "`kdfType`" indicates the type of algorithm used; as previously observed, when using PBKDF2, this value was 0. The "`iterations`" value specifies the number of iterations, which determines the computational effort required. The "`memory`" value indicates the memory cost in mebibytes (MiB), and the "`parallelism`" value defines the number of parallel threads. These details are documented in *Argon2-cffi 23.1.0 documentation* (2025).

Given this information, it is theoretically feasible to attempt the computation of the original Argon2 hash for the purpose of conducting a dictionary attack, akin to the methodology applied to PBKDF2 using Hashcat. However, a key distinction arises: unlike PBKDF2, the salt utilized in the Argon2 implementation remains unknown just by analysing both the `data.json` file where all parameters are stored and by reading the documentation. It is safe to assume that the Argon2 process entails salting the master password with a value unique to the user such as the username (typically the email), which is then processed through a one-way cryptographic function, specifically blake2b, the documentation for which can be referred to in *BLAKE2* (2025). The resulting output is employed as the "`secret`" value in generating the Argon2id hash. Subsequently, this hash is computed using the identified parameters, namely iterations, memory, and parallelism alongside the salt value. This can be seen in Figure 16 and this information can be found in *KDF Algorithms* 2025.

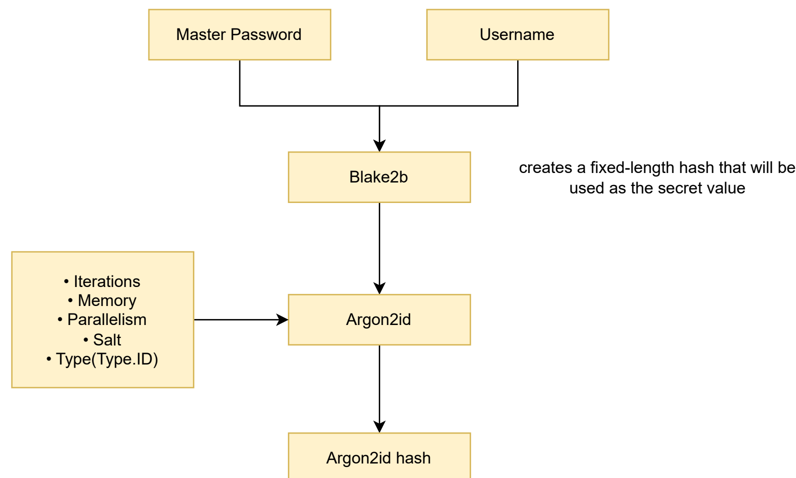


Figure 16: Simple diagram to show how the argon2 implementation works

Bitwarden then passes this hash to an additional hashing operation and encodes the result in Base64, as observed in the "`masterKeyHash`" value within the `data.json` file, shown in Listing 15.

```
1  "user_e9d32a17-1336-43f0-b82d-629a01151d69_masterPassword _masterKeyHash":
↪  "HnBRspdxquRvvdt/hcbtK/LDDuMB×VHRO4gtUeKwayg=",
```

Listing 15: Master Key Hash value found in data.json when using Argon2

It is worth noting that even if the salt were known and it was feasible to brute-force the secret by attempting to match the masterKeyHash value, the process would be

highly time-consuming with sufficiently high parameters. At the time of writing, `Hashcat` has recently accepted a plugin that provides support for Argon2-based passwords (*Support for Argon2id on NVIDIA CUDA GPUs · hashcat/hashcat* 2025). However, it sill lacks clear support for the specific Argon2 implementation used by Bitwarden and KeePassXC, it is not yet a practical option. This is because the plugin requires the derived key to function, and obtaining that key necessitates knowledge of the user's password. A similar situation exists with `John the Ripper`, which also supports Argon2 hashes but operates even more slowly, as it cannot leverage GPU acceleration in the way that `Hashcat` can. While this development marks progress for Hashcat, it is still not a viable solution in this context until a dedicated module is created for these specific implementations.

In conclusion, from the security perspective, it is strongly recommended to adopt a different KDF algorithm such as Argon2, as it provides significantly enhanced security, at least for the present time. Notably, other password managers, namely KeePassXC, have already implemented this as a standard practice.

5.7 SUMMARY

This section summarizes the key findings from the forensic analysis of Bitwarden's desktop application, browser extension, and web application, as detailed in Chapter 5. The analysis aimed to identify residual artifacts and assess the security implications of Bitwarden's data handling practices.

FILE SYSTEM ANALYSIS. The investigation revealed that Bitwarden's desktop application stores critical metadata and the master password hash in a `data.json` file (located in `~/.config/Bitwarden` on Linux and `%AppData%\Bitwarden` on Windows). This file contains the user's email, the KDF type (defaulting to PBKDF2-SHA256), the number of iterations (600 000 by default), and the Base64-encoded master password hash. This persistent storage of the hash and KDF parameters enables offline cracking attempts, particularly for PBKDF2.

Similarly, the browser extensions (Firefox and Google Chrome) store comparable information within SQLite (Firefox) and LevelDB (Chrome) databases. These databases also contain encrypted vault fragments and session states, which, when parsed with custom Python scripts, reveal the KDF metadata and master password hash. This means that even if an attacker gains access to these local files, they possess the necessary information to attempt a password cracking attack.

While Bitwarden supports Argon2id, a more robust and memory-hard KDF, it is not the default. The analysis confirmed that when Argon2id is used, its parameters (iterations, memory, parallelism) are also stored in `data.json`. However, cracking Argon2id hashes is significantly more challenging due to its design making it a more secure choice.

MEMORY ANALYSIS. A significant finding was the presence of sensitive decrypted data in RAM when the Bitwarden vault was unlocked. This included the full plaintext master password and fragments of vault entries (usernames, item labels). Alarmingly, for the Firefox browser extension, instances of the master password were found to persist in clear text in memory even after the vault was locked or the application was closed, depending on system caching behavior. This vulnerability implies that if a forensic investigator or attacker can acquire a memory dump during or shortly after a session, the master password could be recovered without needing to crack any hashes.

CLIPBOARD. Bitwarden utilizes the system clipboard for copying credentials. While it attempts to enhance privacy by bypassing clipboard history managers, tests confirmed that copied credentials are still retrievable in plaintext using tools like xclip (Linux) and Clip View (Windows). The application offers a configurable timeout for clearing the clipboard, but this feature is not enabled by default, increasing the window of exposure for sensitive data.

PASSKEYS. Bitwarden supports storing passkeys within its vault, and the analysis of an exported vault showed the structured storage of passkey credentials (e.g., `credentialId`, `keyType`, `rpId`). While Bitwarden is developing functionality to unlock the vault using passkeys (currently in beta), the tests conducted without a compatible authenticator still required the master password for vault access.

CONCLUSION. In summary, Bitwarden provides a convenient and cloud-centric password management solution, but its design presents certain forensic implications. The storage of master password hashes and KDF parameters in accessible local files (desktop app and browser extensions) makes them susceptible to offline cracking, especially when the default PBKDF2 algorithm is used. More critically, the persistence of plaintext master passwords and decrypted vault data in volatile memory, even after the vault is locked, poses a significant risk if memory dumps are acquired.

While Bitwarden attempts to mitigate some risks (e.g., clipboard history bypass), these efforts are not foolproof. For enhanced security against sophisticated attacks, users are strongly recommended to switch from the default PBKDF2 to Argon2id and be aware of the transient data exposure in memory.

6 KEEPASSXC ANALYSIS

We now analyze KeePassXC. KeePassXC is a derivative of the well-known password manager KeePass. It is supported across multiple platforms, including Windows, Linux, and macOS, while remaining open-source.

KeePassXC operates differently from Bitwarden in several key respects. Primarily, KeePassXC is designed with offline usage as its foremost priority, whereas Bitwarden relies on a server-client architecture, even in self-hosted configurations. This fundamental difference allows KeePassXC to bypass the complexities and security concerns associated with client-server communication, enabling it to focus on providing a robust offline solution. It is worth noting that KeePassXC includes an optional feature, KeePassXC Team (2025), which facilitates database synchronization, however, this functionality is disabled by default and is not the application's primary focus. To compensate for its cloudless architecture, KeePassXC significantly enhances its security measures, maximizing the protective mechanisms employed.

The setup required for KeePassXC is minimal and varies slightly depending on the operating system in use, however, the installation process remains straightforward. Consistent with the methodology previously applied to Bitwarden, this analysis will evaluate KeePassXC within both Windows and Linux environments.

Upon launching KeePassXC for the first time, users are prompted to create a database, which serves as the repository of sorts for all sensitive information stored within the application, effectively functioning as the vault. Before finalizing the database creation, users have the option to modify the encryption settings. A key distinction is that KeePassXC employs Argon2d as its default KDF algorithm, as shown in Figure 17. To mitigate brute-force or dictionary attacks, KeePassXC allows users to adjust the encryption time. By default, it tweaks the user settings to achieve a one-second encryption delay, based on the user's hardware. This parameter can be increased to a maximum of five seconds or decreased to a minimum of 100 milliseconds, with the purpose of making such automated attacks more time-consuming.

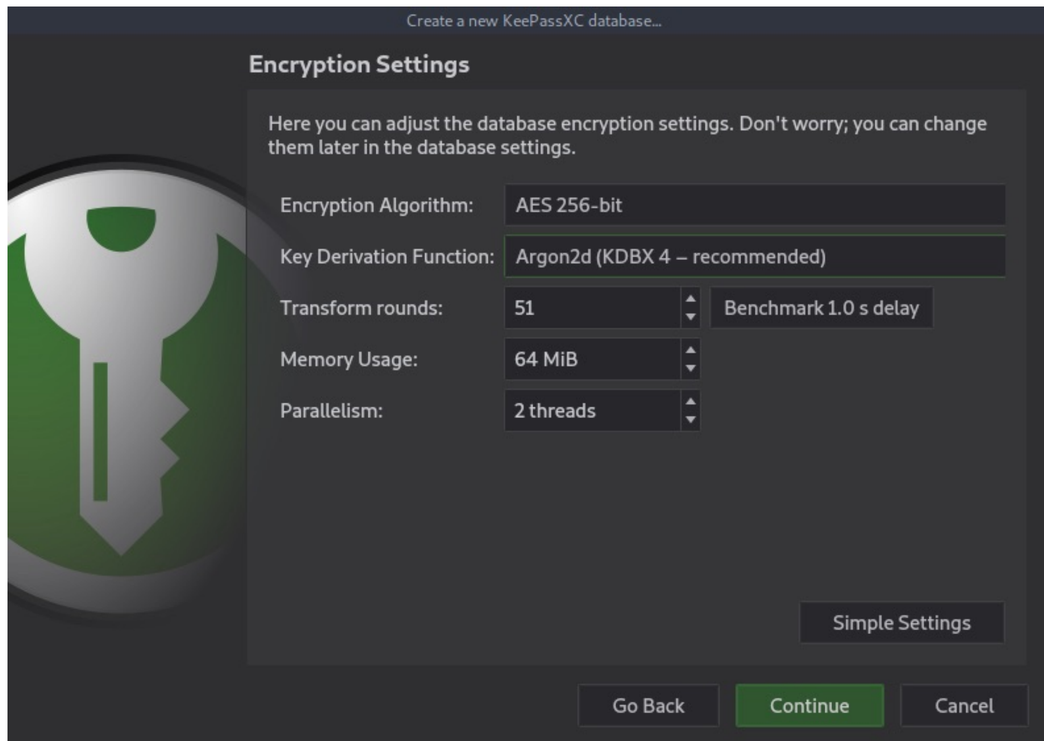


Figure 17: KeePassXC default encryption settings

With the setup finished and the database created with the default settings it's finally possible to start the analysis on the desktop application as well as the browser extension, for both Chrome and Firefox.

In Table 8 are listed KeePassXC's version for each application analyzed.

Application	Version
Desktop App	2.7.9
KeePassXC Browser extension	1.9.7

Table 8: List of the KeePassXC application versions.

6.1 PASSKEYS

Similar to Bitwarden, KeePassXC provides the functionality to store passkeys within its database. However, a notable distinction exists between the two: Bitwarden permits the use of passkeys to unlock the vault, whereas KeePassXC does not support this feature. Instead, KeePassXC solely stores the passkeys and, with the use of its browser extension, enables their use for login purposes.

To evaluate this functionality, the same procedure applied to Bitwarden was utilized. This involved accessing the website referenced in *WebAuthn.io* (2024) to create and store a passkey, followed by exporting it in an unencrypted format to examining it. This process is illustrated in Listing 16.

```
1          {
2              {
3                  "credentialId": "Oh-W_haP2pKF4js-vvEoAZ1gHVcxW7OvId360gDfE8k",
4                  "privateKey": "-----BEGIN PRIVATE KEY-----\nMIGHAgEAMBMGByqGSM49
                   ↪  AgEGCCqGSM49AwEHBG0wawIBAQQgdH5vnmeUFnLzuOMO\nfAz6umtdLyIUB4
                   ↪  zpFtv3GqsgwXGhRANCAARHyMeAF70Tkk3BPnxczd3h4ECEeczo\nfzNxUKk/
                   ↪  Ewhrx/neK3ernj2HguA0so2HMGakcGkVkOBd212Ah+5MG23Y\n-----END
                   ↪  PRIVATE KEY-----\n",
5                  "relyingParty": "webauthn.io",
6                  "url": "https://webauthn.io",
7                  "userHandle":
                   ↪  "S1NJVHFiM19NOWwwRktqUE1wN2lMYkNaZklqeDlWTXNuZGhWaExpQnFmbw",
8                  "username": "test"
9              }
10           }
```

Listing 16: KeePassXC‘s vault export containing a passkey

In this export we can see a vault entry for the `webauthn.io` service, containing a WebAuthn credential used for secure login. The credential includes a PEM-formatted private key, being utilized for cryptographic signing, and is associated with the user `"test"`. It is tied to the domain webauthn.io as indicated by the `"relyingParty"` and `"URLfields"`, it features a unique credential ID and a userHandle that securely links the credential to a specific user account. This configuration ensures a secure alternative for password based authentication while also using a FIDO compliant procedure. This information can be seen in *KeePassXC: User Guide* 2025.

6.2 CLIPBOARD

This analysis also covers how the clipboard and the auto type functions present in the application work.

KeePassXC by default clears the clipboard 10 seconds after any credential has been copied to the clipboard. This serves to avoid that credentials can be left in the system clipboard. Now for better security KeePassXC also hides the credentials copied from clipboard managers, this means that tools like *CopyQ* 2025 and Windows default clipboard manager will not show in their history of the clipboard any traces of the credentials in question. This does not mean this sensitive data is impossible to find though. A code snippet extracted from *keepassxc/src/gui/Clipboard.cpp* (2025) illustrates this functionality, as depicted in Listing 17.

```
void Clipboard:: setText(const String& text, bool clear)
{
    auto* clipboard = Application::clipboard:
    if (Iclipboard) {
        qWarning("Unable to access the clipboard."):
        return;
    }
    auto* mime = new MimeData;
    mime-›setText (text);
#if defined (_OS_MACOS)
    mime-›setData ("application/x-nspasteboard-concealed-type",
    ↪  text.toUtf8());
#elif defined (ROS_UNIX)
    mime-›setData("x-kde-passwordManagerHint",
    ↪  @ByteArrayLiteral("secret"));
#elif defined(R_OS_WIN)
    mime-›setData("ExcludeClipboardContentFromMonitorProcessing",
    ↪  ByteArrayLiteral("1"));
#endif
    if (clipboard-›supportsSelection) {
        clipboard-›setMimeData(mime, @Clipboard::Selection):
    }
}
```

Listing 17: How KeePassXC hides credentials from clipboard managers

In this tiny code snippet we can see KeePassXC uses QMimeData which is a class in the Qt framework, more information can be found in *Qt Documentation Feedback* (2025). In this context, QMimeData is used to manage the transfer of data, particularly for clipboard operations, ensuring that sensitive information like passwords is handled securely and compatibly across different operating systems. Depending on the platform — Linux or Windows — two different MIME types are used:

- For linux it is used "`x-kde-passwordManagerHint`" with the value "`secret`": This hints to compatible applications or clipboard managers that the data is sensitive, potentially triggering special handling, in this case not storing it in clipboard history.
- While for Windows it is "`ExcludeClipboardContentFromMonitorProcessing`" with the value "`1`" which instructs certain tools to exclude the data from monitoring or processing, further protecting sensitive information.

By employing tools such as *xclip* (2025) on Linux or *ClipView* (2025) on Windows, it is possible to inspect the raw data stored in the clipboard. To evaluate these functionality, an entry was created in the vault with the password set to "`teste_password`". Using ClipView, as illustrated in Figure 18, the password value is displayed in the clipboard. Once the clipboard timer configured in KeePassXC expires, the clipboard is cleared, removing the sensitive information.

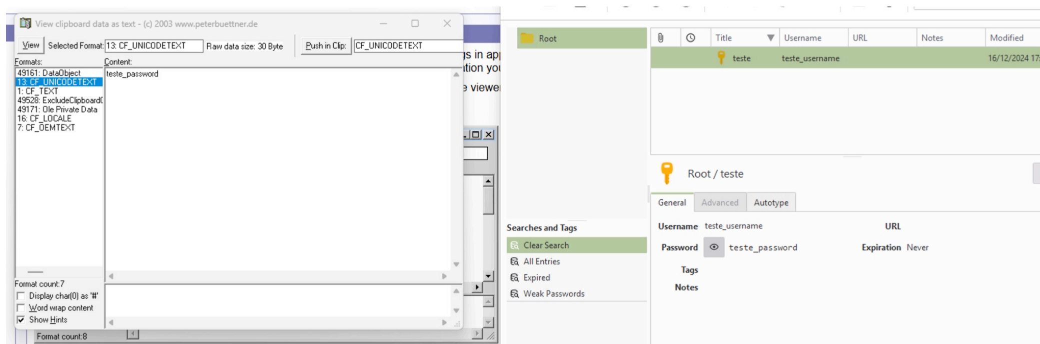


Figure 18: Using Clip View to show the clipboard

Similarly, `xclip`, a command-line utility without a graphical user interface works similarly. By selecting the `"TEXT"` section, the clipboard content is printed to the terminal. If the KeePassXC clipboard timer elapses, the clipboard is cleared, resulting in no value being displayed. This behavior is demonstrated in Figure 19.

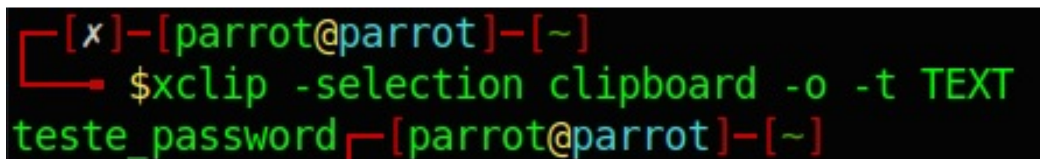


Figure 19: Using xclip to show the contents of the clipboard

Upon examining the results, it is evident that the value extracted from the clipboard is in plaintext. This outcome is expected, as encrypting the value prior to storing it in the clipboard, only to decrypt it when the user wants to access it, would be highly inefficient and could introduce unnecessary delays. KeePassXC does a good job concealing this data from standard clipboard managers, thereby adding a layer of security. However, the value remains accessible as long as the clipboard has not been cleared.

6.3 AUTOTYPE

Moving on to the autotype feature, this functionality is straightforward in its operation. It simulates keystrokes to automatically input credentials, providing a more automated approach to entering login information. However, a significant drawback is that these keystrokes can be easily intercepted, for example, by a simple keylogger. To evaluate this feature, we employed a basic keylogger, whose source code is available in *benign-key-logger* (2025). The test involved opening a notepad

and activating the auto-type functionality with the same password. The results of this experiment are illustrated in Figure 20.

```
C:\Users\home\Downloads\tese\benign-key-logger-master- THIS WORKS\benign-key-logger-master>python key_logger.py
2025-03-12 19:48:20.483 : INFO  : getting set up
2025-03-12 19:48:20.507 : INFO  : SQLite database set up: key_log.sqlite
2025-03-12 19:48:20.523 : INFO  : starting to listen for keyboard events
2025-03-12 19:48:23.001 : INFO  : key: <space>
C:\Users\home\Downloads\tese\benign-key-logger-master- THIS WORKS\benign-key-logger-master\key_logger.py:283: Deprecatio
nWarning: datetime.datetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezone-aware obj
ects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
  row_values = (datetime.utcnow().isoformat(), log_entry)
2025-03-12 19:48:54.535 : INFO  : key: t
2025-03-12 19:48:54.587 : INFO  : key: e
2025-03-12 19:48:54.627 : INFO  : key: s
2025-03-12 19:48:54.661 : INFO  : key: t
2025-03-12 19:48:54.693 : INFO  : key: e
2025-03-12 19:48:54.727 : INFO  : key: _
2025-03-12 19:48:54.760 : INFO  : key: p
2025-03-12 19:48:54.794 : INFO  : key: a
2025-03-12 19:48:54.828 : INFO  : key: s
2025-03-12 19:48:54.860 : INFO  : key: s
2025-03-12 19:48:54.911 : INFO  : key: w
2025-03-12 19:48:54.961 : INFO  : key: o
2025-03-12 19:48:54.994 : INFO  : key: r
2025-03-12 19:48:55.026 : INFO  : key: d
```

Figure 20: Using a keylogger to on KeePassXC Autotype

The results resemble those from the clipboard analysis. As mentioned previously, this outcome is expected since these functionalities are standard among password managers. Such features inherently require a level of access to the user's system, leaving little the application can do. Nevertheless, KeePassXC still tries to keep vault data private from the rest of the system.

6.4 MEMORY ANALYSIS

In this section, we explore whether any information can be extracted from KeePassXC by analyzing its memory.

Like Bitwarden, KeePassXC does not write decrypted data to persistent storage, as it keeps the information solely in volatile memory.

We followed the same approach as with Bitwarden: launching the application, unlocking the vault, dumping its process memory, and extracting readable data using the `strings` tool. However, two key challenges emerged when performing this on KeePassXC.

First, dumping process memory requires administrator privileges, preventing regular users or processes from accessing KeePassXC's memory without elevated permissions. This security measure helps mitigate threats from unauthorized access. However, if an attacker gains administrator privileges, no password manager can fully protect against memory-based attacks.

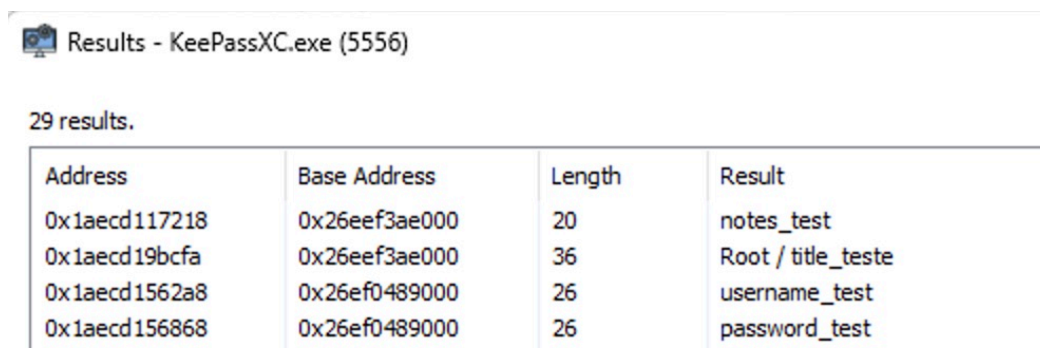
Second, using **strings** to analyze the memory dump yielded limited extractable information. Despite this, we were still able to retrieve useful details, such as the names and file paths of the databases currently in use by KeePassXC, as shown in Figure 21.

```
❯ strings KeePassXCv3.DMP | grep .kdbx
"C:\Program Files\KeePassXC\KeePassXC.exe" "C:\Users\home\Desktop\teste123.kdbx"
C:\Users\home\Desktop\teste123.kdbx
C:\Users\home\Desktop\teste123.kdbx
```

Figure 21: KeePassXC Memory Dump where we can see the database file

Beyond this, we were unable to extract significant additional information from the memory dump. However, community posts on GitHub (*Issue #7335* 2025; *Issue #8939* 2025) indicate that the open-source tool `System Informer` can reveal further strings. `System Informer` is designed to monitor system resources, assist in software debugging, and potentially detect malware. It is capable of identifying a broader range of strings by detecting both American Standard Code for Information Interchange (ASCII) and Unicode characters —— similar to the functionality of `strings` —— but offers the added advantage of a graphical user interface, which facilitates the analysis process. More information about this software can be found in *System Informer* (2025).

Using this tool, you can view the database contents by dumping memory while the database remains unlocked. For example, we created an entry with the username "`username_test`" and the password "`password_test`." As shown in Figure 22, the tool extracts these values from process memory. Note that this functionality requires running `Process Informer` with administrator permissions.


Results - KeePassXC.exe (5556)

29 results.

Address	Base Address	Length	Result
0x1aecd117218	0x26eef3ae000	20	notes_test
0x1aecd19bcfa	0x26eef3ae000	36	Root / title_teste
0x1aecd1562a8	0x26ef0489000	26	username_test
0x1aecd156868	0x26ef0489000	26	password_test

Figure 22: `System Informer` reading KeePassXC memory while the vault is unlocked

These results are expected since the vault is unlocked. However, repeating the test with the database locked reveals no credentials in clear text or any trace of the master password, unlike Bitwarden. Some fragments of the database's XML structure are still visible, as shown in Figure 23. These fragments appear to represent

a database entry, displaying values for the "`Notes`," "`Title`," "`URL`," and "`Username`" fields, while the "`Password`" field remains encrypted and encoded in base64.


```
0x1aecd9def90 (409):                    </Times>
                                <String>
                                        <Key>Notes</Key>
                                        <Value>notes_test</Value>
                                </String>
                                <String>
                                        <Key>Password</Key>
                                        <Value Protected="True">dfPiz0oKsch4iAqNrQ==</Value>
                                </String>
                                <String>
                                        <Key>Title</Key>
                                        <Value>title_teste</Value>
                                </String>
                                <String>
                                        <Key>URL</Key>
                                        <Value/>
                                </String>
                                <String>
                                        <Key>UserName</Key>
                                        <Value>username_test<o
|
```


Figure 23: Fragment of the database's XML structure extracted from `System Informer`

This behavior stems from the Qt XML parser, which consists of classes and tools within the Qt framework for parsing, handling, and generating XML data. More details are available in *Qt Documentation - XML Processing* (2025). As noted in the previously referenced GitHub issues, the XML parser retains memory information in areas that KeePassXC cannot clear. However, memory dump-based data exposure poses minimal threat since it requires administrator privileges. If an adversary has already obtained administrator access, the system is already fully compromised, making memory dumps a secondary concern.

In the case of the KeePassXC extension, no relevant data was found in memory. Unlike Bitwarden, the extension functions solely as a connector for the desktop application and does not handle any encryption or decryption processes.

6.5 FILE ANALYSIS

Like Bitwarden, KeePassXC never stores decrypted data on non-volatile storage. Given this, it is essential to examine whether any files left by KeePassXC are vulnerable to attacks or could potentially expose information.

This section is divided into two parts — Application Data and KeePassXC Password Database — each focusing on the types of data identified from KeePassXC.

6.5.1 *Application Data*

We now examine the information left by the desktop application and the browser extension.

Starting with the desktop application, the only files containing extractable information were located in the folder `%AppData%\local\KeePassXC`. This directory holds a single `.ini` file named `keepassxc.ini`, which serves as a configuration file for KeePassXC, containing settings required to launch the application correctly.

It is worth noting that another `keepassxc.ini` file exists in the `%AppData%\Roaming\KeePassXC` folder. However, this file did not contain any information valuable for a forensic analysis. This distinction arises from the fact that files stored in the `Roaming` directory can be synchronized across devices within the same domain, enabling user settings to persist across different systems. In contrast, the `Local` folder contains application data that is specific to a single machine, such as cache files and logs, and is not synchronized across devices or user profiles. In this context, the presence of data in this file may contain information of potential interest to an attacker or forensic analyst. It includes a history of all password database files that have interacted with KeePassXC, regardless of whether they were successfully unlocked. Additionally, it records the last active database, indicating the most recent database that was successfully accessed, and the last opened databases, providing a history of previously unlocked databases. What makes this information even more valuable is that these values show the entire path of the database files, which could be useful to find where these files are or were located in the machine. In Listing 18, we can see an example of the contents found in this type of file.

```
1   [General]
2   LastDatabases=C:\\Users\\home\\Desktop\\Passwords.kdbx,
    ↪  C:\\Users\\home\\Desktop\\teste123.kdbx, C:\\Users\\home\\teste123.kdbx,
    ↪  C:\\Users\\home\\Desktop\\teste_123.kdbx,
    ↪  C:\\Users\\home\\Desktop\\test_chacha_argon2d.kdbx
3   LastKeyFiles=@Variant(\0\0\0\x1c\0\0\0\0)
4   LastChallengeResponse=@Variant(\0\0\0\x1c\0\0\0\0)
5   LastActiveDatabase=C:\\Users\\home\\Desktop\\teste123.kdbx
6   LastOpenedDatabases=C:\\Users\\home\\Desktop\\teste123.kdbx,
    ↪  C:\\Users\\home\\Desktop\\Passwords.kdbx
7   LastDir=@Variant(\0\0\0\x1c\0\0\0\x1\0\0\0\x4\0\x64\0\x62\0\0\0\n\0\0\0*\0\x
    ↪  43\0:\0/\0U\0s\0\x65\0r\0s\0/\0h\0o\0m\0\x65\0/\0\x44\0\x65\0s\0k\0t\0o\
    ↪  0p)
```

Listing 18: Information on the `keepassxc.ini` file

The file also contains additional information related to the application's Graphical User Interface (GUI), as well as a component referred to as "SSHAgent" — a feature

that is disabled by default in KeePassXC, as documented in *SSH Agent Integration* (2025). However, these fields are either empty or lack significant information.

Moving on to the browser extension, as done with Bitwarden, it was tested with both Chrome and Firefox. However, unlike Bitwarden, this extension functions solely as a connector to the desktop application, meaning it does not store any data itself. We analyzed the residual data left by the extension, specifically, a LevelDB file for Chrome and a `.sqlite` file for Firefox, but found no information of forensic relevance.

6.5.2 *KeePassXC Password Database*

The main difference between KeePassXC and Bitwarden is that KeePassXC stores the vault locally on the user's machine instead of syncing it with a server. This means the complete vault resides on the computer, and KeePassXC employs highly secure methods for encryption and decryption when needed. Consequently, it is possible to analyze the local vault file to determine if any information can be extracted for forensic purposes.

KeePassXC saves protected information in a `database` file with the `".kdbx"` format. This format originates from KeePass2, which aligns with KeePassXC, as it is based on KeePass2.

Before proceeding it is important to know how this type of format works, so it is easier to try and extract its contents. KeePassXC default configuration uses version 4 of the KDBX format, as this is the one that supports the Argon2d KDF algorithm. Complete information about this format can be found in *KDBX File Format Specification - KeePass* (2025).

The file consists of two primary segments: the unencrypted and the encrypted segments.

The unencrypted segment includes:

- An `outer header` containing two signatures, the format version, KDF algorithm parameters, encryption details, and information about the compression algorithm.
- A `SHA-256 hash of the header` for integrity verification, without knowing the master key.

- A `HMAC-SHA-256 of the header` to ensure the header's integrity and authenticity, where the correct master key is required for verifying the HMAC.

The encrypted segment comprises:

- An `inner header` specifying the inner encryption algorithm, the corresponding encryption key, and some binary content.
- An `XML document` that holds most of the user data, structured according to the XML schema outlined in *KDBX - XML schema* (2025).

In this analysis, we will primarily focus on the unencrypted segment, as it is the only part we can examine without prior knowledge of the master password or any other sensitive information.

The diagram shown in Figure 24 depicts the fields of the unencrypted segment, and its location relatively to the encrypted segment.

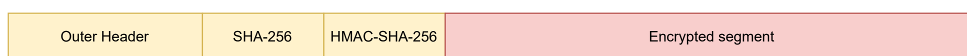


Figure 24: Diagram showing how the unencrypted segment fits in the .kdbx format

Using the tool HxD and the website, *Documenting KeePass KDBX4 file format* (2025), as a guide we analyzed the database file to extract information from the outer header. It is important to note that all integers are stored in little-endian format and strings are stored in UTF-8 enconding.

When analyzing the outer header, the first two fields encountered are of type `Uint32`, representing the format's signatures. These values are consistent across all versions of this format. Following these are two `Uint16` fields, `versionMajor` and `versionMinor`, which serve solely for version identification. For a clearer visualization, refer to Table 9.

Field	**Type**	**Value**
Signature 1	Uint32	0x9AA2D903
Signature 2	Uint32	0xB54BFB67
Version major	Uint16	Depends on the version of the format
Version minor	Uint16	Depends on the version of the format

Table 9: First four fields found in the outer header.

The remaining header structure consists of a sequence of header fields, where each entry includes:

- A `type` field (`Uint8`) indicating the field's ID.

- A `size` field (`Uint32`) specifying the byte length of the subsequent data.
- A `data` field, with its type dependent on the field's content, containing the actual information.

With this in mind it is not very hard to navigate the values present in the outer header, in total we have six fields in this type of structure, starting with the first one found:

- ID 2: This field has a size of 16 bytes and is a UUID, as specified in the KeePass2 documentation. The value depends on the encryption algorithm used:
  - `31C1F2E6BF714350BE5805216AFC5AFF` for AES-256
  - `D6038A2B8B6F4CB5A524339A31DBB59A` for ChaCha20
- ID 3: This unsigned four-byte field value of type `Uint32` is related to the usage of a compression algorithm. Specifically, zero means no compression, while one indicates that GZip compression is applied to the encrypted segment.
- ID 4: This field has a size of 32 bytes and holds the master salt used for deriving encryption keys.
- ID 7: The size of this field varies based on the encryption algorithm and contains the Initialization Vector (IV) required for encryption.
- ID 11: This special field is a `Variant Dictionary` that includes parameters for the KDF.
- ID 12 (Optional): Intended for plugin-related data, this field is common in KeePass2 but rarely used in KeePassXC. Consequently, it was absent in the tested database files.

Before looking deeper into the `Variant Dictionary` (ID 11), refer to Table 10 for an overview of these fields.

ID	Size (Bytes)	Description	Notes
2	16	UUID	Depends on encryption algorithm (AES-256 or ChaCha20)
3	4	Compression Flag	0 = None, 1 = GZip
4	32	Master Salt for Key Derivation	Used to compute encryption keys
7	Variable	Initialization Vector (IV)	Size depends on encryption algorithm
11	Variable	Variant Dictionary	Contains KDF parameters
12	Variable (Optional)	Plugin Data	Rare in KeePassXC

Table 10: Fields present in the outer header of the KeePassXC database.

The `Variant Dictionary` fields function as a key-value storage system. They begin with a `UInt16` value that specifies the format version. Typically, this version is encoded as `0x100`. Each entry in the `Variant Dictionary` follows a structured format:

- **Type**: A one byte (`Uint8`) field indicating the data type of the value. Possible types include:
  - `0x04`: `UInt32`
  - `0x05`: `UInt64`
  - `0x08`: `Boolean`
  - `0x0C`: `Int32`
  - `0x0D`: `Int64`
  - `0x18`: `String`
  - `0x42`: `Byte[]`
- **Key Size**: A four byte (`Uint32`) field specifying the length of the key.
- **Key**: The name of the key, with its size determined by the preceding `keySize` field.
- **Value Size**: A four byte (`Uint32`) field indicating the length of the value.
- **Value**: The actual value corresponding to the key, with a size based on the preceding `valueSize` field.

When examining the Variant dictionary, two distinct sets of fields emerge depending on the chosen KDF algorithm, either AES-KDF or Argon2d.

For the AES-KDF algorithm, the following fields are present:

- **Key `$UUID`**: This field identifies the unique ID of the KDF being used. For AES-KDF, the expected value is `C9D9F39A628A4460BF740D08C18A4FEA`.
- **Key `S`**: This field stores the salt utilized in the KDF algorithm. It is 32 bytes in length.
- **Key `R`**: This field specifies the number of rounds used in the KDF algorithm.

For the Argon2d algorithm, the Variant dictionary includes the following fields:

- **Key `$UUID`**: Identifies the unique ID of the KDF. For Argon2d, the expected value is `EF636DDF8C29444B91F7A9A403E30A0C`.
- **Key `V`**: Specifies the version of the Argon2 algorithm, either `0x10` (version 1.0) or `0x13` (version 1.3).
- **Key `S`**: Stores the salt used in the KDF algorithm. The size is flexible, but 32 bytes is recommended.
- **Key `I`**: Represents the number of iterations, using an 8-byte (`Uint64`) format.
- **Key `M`**: Indicates the memory usage in bytes for the KDF parameters, also 8 bytes long (`Uint64`).
- **Key `P`**: Defines the level of parallelism in the KDF process, stored as a 4-byte (`Uint32`) value.

These details can be better visualized in Table 11.

KDF Algorithm	Key	Description	Data Type / Size
AES-KDF	`$UUID`	Unique ID of the KDF	Fixed
	`S`	Salt for the KDF process	32 bytes
	`R`	Number of rounds	Uint32
Argon2d	`$UUID`	Unique ID of the KDF	Fixed
	`V`	Argon2 version	Uint32 (`0x10` or `0x13`)
	`S`	Salt for the KDF process	Recommended 32 bytes
	`I`	Number of iterations	Uint64 (8 bytes)
	`M`	Memory usage (in KiB)	Uint64 (8 bytes)
	`P`	Parallelism factor	Uint32 (4 bytes)

Table 11: Comparison of KDF parameters for AES-KDF and Argon2d algorithms.

The single byte 0x00 marks the end of the `Variant Dictionary`. From this point, we continue parsing the outer header until we identify the field with ID 0, which is four bytes in size and contains the byte array value `0x0D, 0x0A, 0x0D, 0x0A`. This marks the final field of the header.

Following this, the structure proceeds with the `SHA-256` hash of the header content, ensuring its integrity, and an `HMAC-SHA-256` for both integrity and authenticity validation. Beyond these, we enter the encrypted segment, where the **`inner header`** and the **`XML document`** containing user data are located.

With the outer header fully analyzed, it becomes evident that a substantial amount of useful information can be extracted from it. The next logical step is to automate this extraction process.

While several scripts already exist for this purpose, most are only compatible with version 3 of the `kdbx` file. Given this limitation, we developed the `kdbx4_header_extractor.py` Python script to automate the extraction process for version 4 of the kdbx database file. The script is available at *KeePass2-kdbx-4-header-extract* (2025) and the full Python implementation is included in Appendix A, Listing 21.

This script extracts all the fields mentioned above and displays them in the console for the user. More details can be found in the referenced GitHub repository.

An example output of this script can be seen in Listing 19.

```
1  Value of the first signature (uint32, little-endian): 0x9aa2d903
2  Value of the second signature (uint32, little-endian): 0xb54bfb67
3  Format version: 4.0
4  ID 2 Encryption algorithm - AES-256 (size 16 bytes):
   ↪  31c1f2e6bf714350be5805216afc5aff (hexdecimal)
5  ID 3 Compression algorithm (size 4 bytes): 1 (using GZIP)
6  ID 4 Master salt/seed (size 32 bytes):
   ↪  470541af0812fa595728152e0fae91e785ab8887bc954b5cd64f276587ba8fad
   ↪  (hexadecimal)
7  ID 7 Encryption IV/nonce (size 16 bytes): 9077dbc963b4656f139ed29065c99f5f
   ↪  (hexadecimal)
8  ID 11 KDF parameters (size 139 bytes) - Variant dictionary:
9  Format version: 0x100
10 KDF algorithm:
11  entry name -> $UUID
12  size -> 16
13  value -> ef636ddf8c29444b91f7a9a403e30a0c
14 This means the used KDF ALgorithm used was Argon2
15 Argon2 Iterations:
16  entry name -> I
17  size -> 8
18  value -> 58
19 Argon2 Memory:
20  entry name -> M
21  size -> 8
22  value -> 16.0 MiB
23 Argon2 Parallelism:
24  entry name -> P
```

```
25   size -> 4
26   value -> 2 threads
27  Argon2 Salt:
28   entry name -> S
29   size -> 32
30   value -> 48148e2966c90461318ccb5907ae10602ffabbfd6b5744963ae8ab3bd9c68c2d
    ↪  (hexadecimal)
31  Argon2 Version:
32   entry name -> V
33   size -> 4
34   value -> 0x13 (hexadecimal)
35  Reached the end of the Variant dictionary, continuing to read the headers of the
    ↪  file...
36  ID 0 End of headers reached! (size 4 bytes): 0xa0d0a0d
```

Listing 19: Example output of the script `kdbx4_header_extractor.py`

Since we now have all the values that can be extracted from the outer header, another GitHub repository was created where these values are transformed into an input format compatible with the `John the Ripper` tool. `John the Ripper` is a tool similar to `Hashcat`, with more information available in *openwall/john* (2025).

The new script builds upon `extract_header.py` but, instead of simply printing the results, it formats them into a string compatible with `John the Ripper`. Since modules for KeePassXC Argon2d implementation are not available in `John the Ripper` at the time of writing, our approach adapts the original `keepass2john`, a C-based program from the official repository, by replicating its functionality in Python. The GitHub repository for this script is available in *CamposTaPro/Keepass2John* (2025).

The output generated by this script is straightforward, a single string containing the following values: KDBX version, iterations, KDF UUID, memory, Argon2 version, parallelism, master salt, KDF salt, SHA-256 header, and HMAC-SHA-256 header. Each value is separated by an asterisk (“*”), as illustrated in Figure 25.

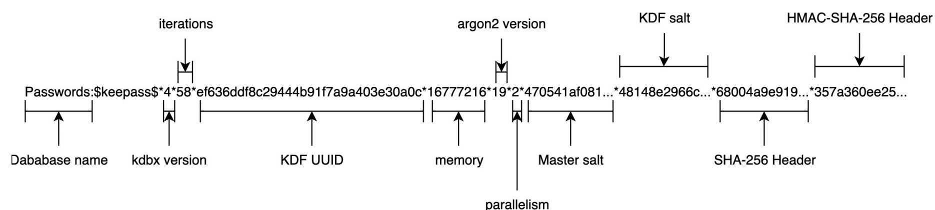


Figure 25: Breakdown of output of the script `keepass2john.py`

6.6 PROOF OF CONCEPT

An experiment with the extracted results is to attempt an attack on the database to determine if it's possible to eventually uncover the master password. However, unlike with Bitwarden, tools like `Hashcat` or `John the Ripper` are not viable options this time, even with some support for Argon2d hashes there is not a dedicated module for KeePassXC Argon2 implementation. Therefore, we focused on understanding how KeePassXC handles encryption and decryption using the information available in the outer header and turns out there is a way to brute force attack the database even if not the most efficient.

Before diving deeper, it is essential to have at least a basic understanding of how KeePassXC encryption works. The process begins with key computation:

1. KeePassXC first creates a key by passing the master password through SHA-256 twice. This is referred to internally as R, but we can call it the composite key.
2. This composite key is then used as the secret input for the KDF, alongside other parameters found in the outer header.
3. The output of this process is what KeePassXC calls it T, which we can refer to as the derived key.

The process is illustrated in Figure 26.

After generating the derived key (T), it can be used in several ways:

- **Creating the encryption key**: This is done by concatenating the master salt with the derived key.
- **Generating an HMAC key**: This key is used to compute the HMAC-SHA-256 hash for individual data blocks.

KeePassXC takes a unique approach to encryption. Instead of storing a single HMAC signature for the entire encrypted database, it **splits the encrypted data into blocks**, typically 1 MB each. Each block is separately signed, ensuring integrity and authentication.

The first block is assigned **BlockIndex 0**, the next **BlockIndex 1**, and so on. This block-wise HMAC signing method enhances security and efficiency. To illustrate this structure, refer to Figure 27.

The **Base HMAC Key** can now be used with any **Block Index** to generate the HMAC-SHA-256 hash. Conveniently, we already have the HMAC-SHA-256 hash for

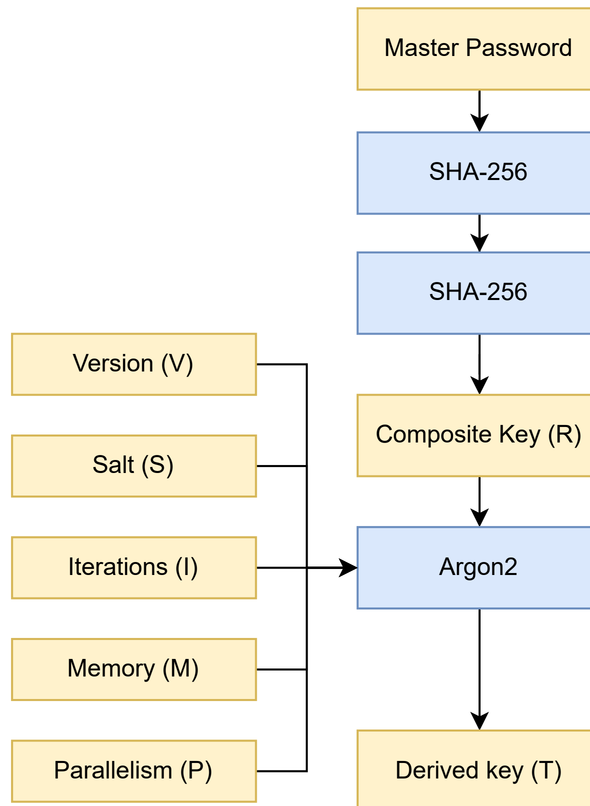


Figure 26: KeePassXC Composite Key and Derived Key demonstration

the header, and the block index for the header is static with the eight byte value `0xFFFFFFFFFFFFFFFF`.

This means we can replicate the process of generating the HMAC-SHA-256 hash using different master passwords. By iterating through various password guesses and computing the corresponding HMAC-SHA-256 hash, we can compare it against the known hash stored in the `.kdbx` file. When we find a match, we have successfully discovered the correct master password.

Before proceeding, we can visualize this final step in Figure 28.

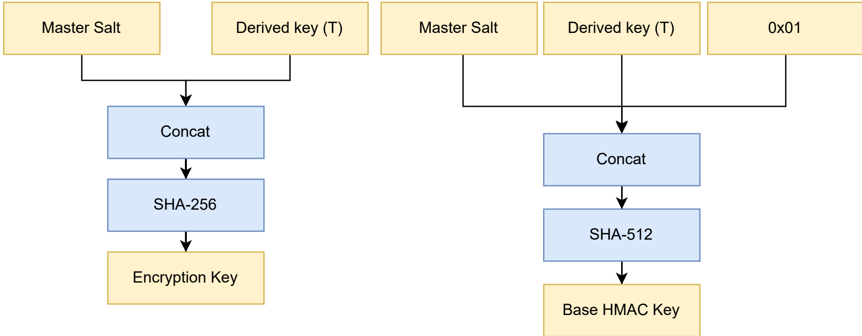


Figure 27: KeePassXC Base HMAC Key demonstrattion

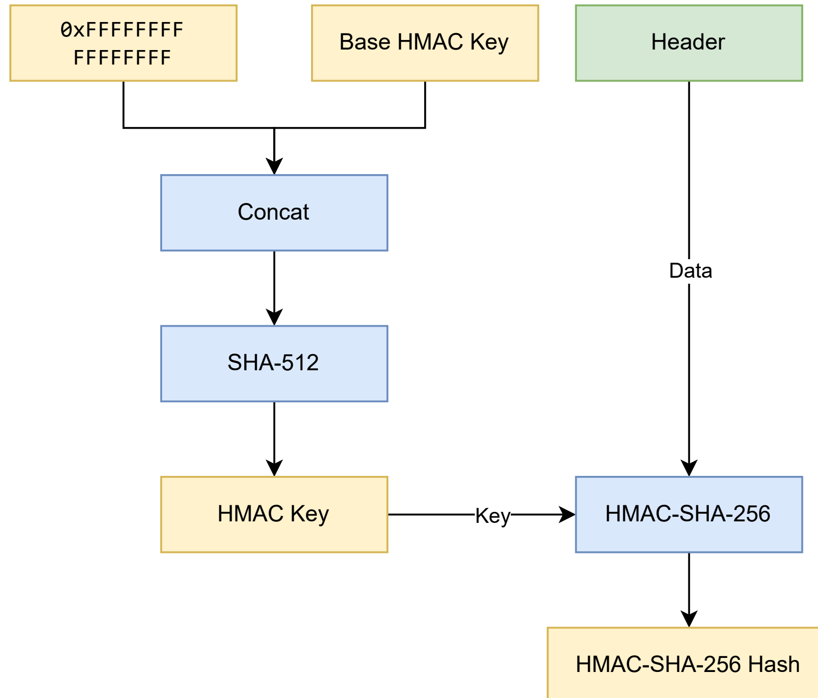


Figure 28: KeePassXC HMAC Key demonstration

To test this, we added a directory in the GitHub repository from the header extraction project, where you can find a `.kdbx` file protected by AES encryption and Argon2 with deliberately low parameter values. These include only two iterations, 16 384 KiB of memory, and a parallelism factor of one. Additionally, the master password is only three characters long. These settings were chosen to allow for faster password attempts, as real-world values would make each attempt significantly slower.

Alongside this, we included a script named `pocArgon2d.py`. Its code can be seen in Listing 20.

```
1  import argon2.low_level
2  from argon2 import Type
3  import hmac
4  import hashlib
5  def double_sha256(password: bytes) -> bytes:
6      """Compute SHA256(SHA256(password)) as per KDBX spec."""
7      return hashlib.sha256(hashlib.sha256(password).digest()).digest()
8  def read_header(filename: str) -> bytes:
9      """Read KDBX4 header until End-of-Header marker."""
10     with open(filename, "rb") as f:
```

```
11          data = b""
12          while True:
13              chunk = f.read(1024)
14              if not chunk:
15                  break
16              data += chunk
17              if b"\x0d\x0a\x0d\x0a" in data:  # End-of-Header marker
18                  eoh = data.index(b"\x0d\x0a\x0d\x0a") + 4
19                  return data[:eoh]
20          raise ValueError("Invalid KDBX header")
21  def brute_force_argon2(
22      wordlist: list[bytes],
23      iterations: int,
24      memory: int,
25      parallelism: int,
26      salt: bytes,
27      master_seed: bytes,
28      stored_hmac: bytes,
29      header: bytes
30  ) -> None:
31      """Brute-force a KDBX4 database using extracted parameters."""
32      block_index = b"\xff" * 8   # 0xFFFFFFFFFFFFFFFF
33      for password in wordlist:
34          try:
35              print(f"Trying password: {password.decode('utf-8',
                ↪  errors='replace')}")
36              # 1. Composite key
37              composite_key = double_sha256(password)
38              # 2. Argon2d derived key
39              derived_key = argon2.low_level.hash_secret_raw(
40                  secret=composite_key,
41                  salt=salt,
42                  time_cost=iterations,
43                  memory_cost=memory,
44                  parallelism=parallelism,
45                  hash_len=32,
46                  type=Type.D
47              )
48              # 3. HMAC key derivation
49              x = hashlib.sha512(master_seed + derived_key + b"\x01").digest()
50              hmac_key = hashlib.sha512(block_index + x).digest()
51              computed_hmac = hmac.new(hmac_key, header, hashlib.sha256).digest()
52              # 4. Validate
53              if hmac.compare_digest(computed_hmac, stored_hmac):
54                  print(f"\nSUCCESS! Password: {password.decode()}")
55                  return
56          except Exception as e:
57              print(f"Error processing '{password.decode('utf-8',
                ↪  errors='replace')}': {e}")
58      print("\nPassword not found.")
59  # ------------------------------------------
60  # Example usage (replace with extracted values)
61  if __name__ == "__main__":
62      # Parameters (extract these from your KDBX4 file)
```

```
63      ITERATIONS = 2          # WARNING: Use 100,000+ in real-world
64      MEMORY = 16384          # WARNING: Use 1,048,576 (1 GiB) in real-world
65      PARALLELISM = 1
66      SALT = bytes.fromhex("5c7121609bd58f0333b33b3fddb29d22bc57343cf868fde9ca0034
        ↪  d16c308156")
67      MASTER_SEED = bytes.fromhex("32da47790ac65e8a321aca4d8268319f946d8dd763a2b80
        ↪  7f2408e2ba6e72049")
68      STORED_HMAC = bytes.fromhex("5503588b694086b194c2ef08f285c59849989e0b85e6f36
        ↪  bfb63fd200aec84b5")
69      HEADER = read_header("testePoc.kdbx")
70      # Wordlist
71      WORDLIST = [
72      b"111", b"112", b"113",b"121", b"122",b"131", b"132", b"133",b"211", b"212",
        ↪  b"213",b"221", b"222", b"223",b"231", b"232", b"233",b"311", b"312",
        ↪  b"313",b"321", b"123", b"322", b"323",b"331", b"332", b"333"]
73      brute_force_argon2(WORDLIST, ITERATIONS, MEMORY, PARALLELISM, SALT,
        ↪  MASTER_SEED, STORED_HMAC, HEADER)
```

Listing 20: Code of the script `pocArgon2d.py`

This proof of concept demonstrates that the database file can be successfully targeted using information extracted from it, even if the approach is not optimized for efficiency.

6.7 SUMMARY

This section summarizes the key findings from the forensic analysis of KeePassXC, as detailed in Chapter 6. The analysis aimed to identify residual artifacts and assess the security implications of KeePassXC's data handling practices, focusing on its desktop application and browser extension.

PASSKEYS KeePassXC supports storing passkeys within its database, similar to Bitwarden. However, a key distinction is that KeePassXC does not allow passkeys to unlock the vault; instead, it solely stores them and facilitates their use for login purposes via its browser extension. Analysis of an exported vault revealed the structured storage of passkey credentials, including a PEM-formatted private key, relying party information, and user details, ensuring a secure alternative for password-based authentication.

CLIPBOARD. KeePassXC includes a clipboard functionality that automatically clears copied credentials after a default timeout of 10 seconds, preventing sensitive

data from lingering in the system clipboard. It also attempts to hide credentials from standard clipboard managers by using specific MIME types for Linux and Windows. While these measures enhance privacy, tests using tools like `xclip` (Linux) and `Clip View` (Windows) confirmed that copied credentials are retrievable in plaintext as long as they remain in the clipboard. This behavior is expected, as encrypting clipboard content would introduce inefficiencies.

AUTOTYPE. The autotype feature in KeePassXC simulates keystrokes to automate credential input. While convenient, this functionality is vulnerable to interception by simple keyloggers, as demonstrated by tests. This is an inherent limitation of such features, as they require direct interaction with the user's system, leaving little room for the application to prevent interception.

MEMORY ANALYSIS. KeePassXC, like Bitwarden, does not write decrypted data to persistent storage, keeping it solely in volatile memory. Memory dumps of the KeePassXC process, requiring administrator privileges, revealed limited extractable information using the `strings` tool, primarily database names and file paths. However, using System Informer, more detailed information, including plaintext usernames and passwords, was observed when the vault was unlocked. Crucially, when the database was locked, no cleartext credentials or master password traces were found, unlike Bitwarden. Fragments of the database's XML structure were visible due to the Qt XML parser retaining memory information that KeePassXC cannot clear. The browser extension showed no sensitive data in memory, as it acts purely as a connector to the desktop application.

FILE ANALYSIS. KeePassXC's desktop application stores configuration data in a `keepassxc.ini` file (in `%AppData%\local \KeePassXC` or `~/.config/keepassxc /keepassxc.ini`). This file contains a history of accessed database file paths, which can be valuable for forensic analysis, but no sensitive credential data or hashes. The browser extension acts only as a connector and does not store any data itself.

The core of KeePassXC's data storage is the local `.kdbx` database file, which uses version 4 of the KDBX format. This file consists of an unencrypted outer header and an encrypted segment. The unencrypted outer header contains critical metadata, including two signatures, the format version, KDF algorithm parameters (defaulting to Argon2d), encryption details, and compression information. This metadata, including the master salt, KDF parameters (iterations, memory, parallelism), and

the HMAC-SHA-256 hash of the header, can be extracted using custom Python scripts. While the encrypted segment contains the user data, the information in the outer header, particularly the KDF parameters and HMAC-SHA-256 hash, can be used to attempt offline brute-force attacks, as demonstrated by a proof-of-concept script.

CONCLUSION. In conclusion, KeePassXC prioritizes offline usage and employs robust security measures, including Argon2d as its default KDF and strong memory hygiene. While it effectively prevents plaintext credential persistence in memory after locking and conceals clipboard data from standard managers, the extractable metadata from its `.kdbx` file's outer header can still be leveraged for offline cracking attempts, albeit with significant computational effort due to Argon2d's design. The application's design significantly reduces its forensic footprint compared to cloud-centric solutions, but forensic analysts can still find valuable information from configuration files and the database header.

7 DISCUSSION

This chapter consolidates the findings from the forensic analysis of two prominent mobile password manager applications: Bitwarden and KeePassXC. Chapters 5 and 6 detailed the specific artifacts recovered and the behaviors observed for each application, respectively. Building upon those individual analyses, the primary objective of this chapter is to conduct a comprehensive comparative analysis of the forensic characteristics of Bitwarden and KeePassXC.

The discussion will focus on comparing and contrasting the types, locations, and persistence of digital artifacts left by both applications across various forensic domains, including file systems, browser extensions, and volatile memory. Furthermore, this chapter assesses the feasibility of password hash cracking for each application based on their underlying cryptographic implementations. By highlighting similarities, differences, strengths, and vulnerabilities, this comparative approach aims to derive deeper insights into the security postures and forensic traceability of these widely used password managers.

7.1 COMPARISON OF ARTIFACT RECOVERY

The analysis of Bitwarden and KeePassXC revealed distinct patterns in the types and locations of digital artifacts left behind, directly impacting their forensic recoverability. This section provides a comparative discussion of these artifacts across file system, browser extension, and volatile memory contexts, leveraging the summary tables presented earlier to highlight key differences and similarities.

7.1.1 *File System Artifacts*

As summarized in Table 12, both Bitwarden and KeePassXC leave a discernible forensic footprint on the file system, yet significant differences exist in their nature and quantity. Bitwarden, as a cloud-centric solution, was observed to generate and store various application-specific files related to cached data, logs, and syn-

chronization states. These often included SQLite databases for local storage and configuration files. In contrast, KeePassXC, being primarily a local-first application, exhibited a different artifact profile, with its primary forensic value residing in the .kdbx database file itself and associated temporary or backup files.

Feature	Bitwarden	KeePassXC	Security Implications
Storage Location	`"~/.config /Bitwarden"` or `"\%AppData\% /Bitwarden"`	`"AppData/local /KeePassXC"` or `"~/.config /keepassxc /keepassxc.ini"`	Knowing storage locations aids in targeted forensic acquisition.
Data Format	JSON files (encrypted vault data, key derivation metadata)	Configuration files, KDBX database file (outer header metadata)	JSON files are generally easier to parse, potentially simplifying analysis.
Master Password Hash Storage	Stores base64 encoded master password hash with KDF parameters	Does not store master password hash separately from the encrypted database	Both application's approach enable offline password cracking attempts.
Key Derivation Metadata	User email, account ID, KDF algorithm, KDF parameters	Outer header of KDBX file contains encryption settings and KDF parameters	Metadata is crucial for understanding encryption and potential weaknesses.
Decrypted Data Persistence	Encrypted vault data is stored, requiring decryption	Does not store decrypted data in non-volatile storage	KeePassXC's design minimizes the risk of recovering decrypted secrets from the file system.

Table 12: Comparison of File System Artifacts found

A key comparative point lies in the potential for credential exposure. While KeePassXC's file system artifacts largely revolve around its encrypted database, Bitwarden's local storage mechanisms presented instances where sensitive, albeit encrypted, data might persist or be temporarily accessible in various application directories, particularly during synchronization or caching processes. The analysis showed that recovering actionable intelligence from Bitwarden often required a deeper dive into its application-specific database structures and cached files, whereas KeePassXC's file system artifacts primarily pointed towards the location and status of the main database file. The presence of robust encryption in both cases means that merely recovering these files does not equate to direct access to plaintext credentials, but their existence and characteristics are crucial for understanding user activity and potential attack vectors.

7.1.2 *Browser Extension Artifacts*

The forensic analysis of browser extensions, detailed in Table 13, revealed a notable disparity between Bitwarden and KeePassXC. Bitwarden, with its fully integrated browser extension, generated a wealth of client-side artifacts. These included cached vault data, user preferences among other information all stored within the browser's local storage and session storage. The direct interaction of the Bitwarden extension with the user's vault, including decryption operations, meant that a forensic examination of browser data yielded significant insights into recently accessed credentials and vault states.

Feature	Bitwarden	KeePassXC	Security Implications
Data Storage	LevelDB database or sqlite database file in browser extension directory.	Acts as a connector to the main application, no local storage.	Local storage increases the attack surface for the browser extension.
Sensitive Data Storage	Stores encrypted credentials, URLs, and metadata).	Does not store sensitive data	Bitwarden's storage of encrypted data within the browser extension poses a higher risk.
Encryption Key Derivation	Encryption key derived from values in the LevelDB database or sqlite database file.	N/A	Compromise of the database could lead to decryption of stored credentials.

Table 13: Comparison of Browser Extension Artifacts found

Conversely, the KeePassXC browser extension, primarily designed as a connector to the desktop application, exhibited a comparatively smaller and less sensitive artifact footprint within the browser itself. Its function is largely to facilitate communication with the KeePassXC application rather than directly store or decrypt vault data. Therefore, while some configuration or communication logs might be present, the critical sensitive data remains primarily within the desktop application's memory or the `.kdbx` file. This highlights a fundamental difference in architecture: Bitwarden uses distributed data handling, storing data locally within the browser, whereas KeePassXC relies on centralized data handling within the application itself, with the extension functioning as a proxy. From a forensic perspective, this implies that a full understanding of Bitwarden usage necessitates thorough browser artifact analysis, whereas for KeePassXC, the focus would remain predominantly on the desktop application.

7.1.3 *Volatile Memory Artifacts*

The examination of volatile memory artifacts, summarized in Table 14, provided critical insights into the real-time exposure of sensitive data by both password managers. Both Bitwarden and KeePassXC, when active and unlocked, were found to contain plaintext credentials and master passwords in RAM, as is expected for any active application managing secrets. However, the duration and specific contexts of this exposure varied.

Feature	Bitwarden	KeePassXC	Security Implications
Master Password in Memory	Master password can persist in memory after vault lock	Master password less likely to persist, requires admin privileges for memory dumping	Persistence of master password in memory increases the risk of extraction.
Decrypted Data in Memory	Decrypted data can be found when the vault is unlocked	Potential for decrypted data to be present	Decrypted data in memory is vulnerable during application use.
Memory Access Requirements	Standard user privileges	Administrator privileges for full process memory access	KeePassXC's requirement for admin privileges provides a degree of protection.
Browser Extension Data in Memory	Data related to extension might be present	No sensitive data stored in extension's memory	Bitwarden's extension might leave more sensitive data in memory.

Table 14: Comparison of Volatile Memory Artifacts found

Bitwarden's memory footprint often included plaintext vault data, recently accessed credentials, and master password derivatives, particularly when the application or extension was in an unlocked state or actively synchronizing data. The challenge lay in identifying and extracting these specific data structures within the larger memory dump. KeePassXC, on the other hand, while also exposing plaintext data when unlocked, demonstrated a different characteristic related to its database-centric approach.

A significant comparative point is the management of sensitive data post-lock or after application closure. While both applications attempt to clear sensitive data from memory, the effectiveness and completeness of these processes are critical. The analysis revealed that despite efforts, remnants of plaintext credentials or master password components could still be recovered from memory dumps, underscoring the inherent challenges in completely sanitizing RAM after sensitive operations. The findings emphasize that volatile memory analysis remains a powerful technique for both applications, offering a window into the state of the password manager at

the time of memory acquisition. Its memory artifacts were more directly tied to the open `.kdbx` database.

7.2 COMPARISON OF HASH-CRACKING FEASIBILITY

The security of stored credentials in password managers is heavily reliant on the strength of their KDFs and the computational cost associated with deriving the master key. As detailed in Table 15, Bitwarden and KeePassXC employ different KDFs, which have significant implications for the feasibility of offline hash-cracking attacks.

Feature	Bitwarden	KeePassXC
Key Derivation Function (KDF)	PBKDF2-SHA256 (default); Argon2id (alternative)	Argon2d (default); AES-KDF (alternative)
Iteration/Cost Parameters	PBKDF2: 600 000 iterations; Argon2id: 1024 MiB memory, 2 time cost, 4 parallelism	High transformation rounds; Argon2d parameters defined in outer header
Hash Extraction Method	Extracted from `data.json` using `bitwarden2 hashcat.py`	Extracted from database outer header using `keepass2john.py`
Use of Salt	Salt applied (details may vary per KDF); assumed unique per user	Salt explicitly used and stored in database metadata
Hash Cracking Tools	Cracked using Hashcat mode 23 400	Possible yet unrealistic in a real world scenario by using the python PoC `pocArgon2d.py`
Decryption Complexity	High due to iteration count and KDF settings	High due to composite key mechanism and configurable KDF
Security Implications	Strong resistance, but offline cracking is feasible with enough resources	Additional indirection from composite key increases difficulty

Table 15: Comparison of Password Hash Cracking Characteristics

Bitwarden uses PBKDF2-SHA256 as its default KDF, a well-established and widely adopted algorithm. It is typically configured with a high iteration count, usually around 600 000 for web and desktop clients, although this value can be adjusted by the user. The primary defense against brute-force attacks on the derived key is this high iteration count, which increases the computational cost of each password guess. While PBKDF2 is a CPU-bound algorithm and therefore more vulnerable to acceleration using specialized hardware, the large number of iterations still makes brute-force attacks impractical for most adversaries when a strong master password is used. Argon2, however, provides a more robust KDF option. It is designed to be memory-hard and resistant to parallelization, offering stronger

protection against modern attack techniques. The main advantage of PBKDF2 is its status as a NIST-approved standard, whereas Argon2, despite being a more secure alternative in practice, has not been formally recognized by any government standards body.

In contrast, KeePassXC employs Argon2 by default, specifically Argon2id, which is designed to be highly resistant to both CPU and GPU-based attacks, as well as memory-hard attacks. Argon2's strength comes from its configurable parameters for iteration count, memory consumption, and parallelism. Its memory-hardness characteristic makes it particularly challenging for attackers to leverage parallel processing capabilities (like those in GPUs) effectively, as each computation requires a significant amount of memory, limiting the number of parallel attempts.

From a comparative perspective, Argon2 (as used by KeePassXC) offers a superior defense against modern hash-cracking techniques, especially those leveraging specialized hardware. While Bitwarden's use of PBKDF2 with high iterations is a strong measure, it is inherently less resistant to memory-hard and parallel processing attacks than Argon2. The analysis of recovered hashes showed that while both presented a formidable challenge with strong master passwords, the inherent design of Argon2 in KeePassXC theoretically imposes a greater resource burden on an attacker attempting offline cracking. Therefore, in terms of hash-cracking feasibility, KeePassXC's choice of Argon2 generally provides a more resilient defense, particularly against advanced, resource-rich adversaries.

7.3 SECURITY TRADE-OFFS AND IMPLICATIONS

The comparative analysis of Bitwarden and KeePassXC reveals distinct security postures and forensic implications, largely influenced by their fundamental design philosophies: a cloud-synchronized model versus a local, file-based approach. Each model presents inherent trade-offs between user convenience and the extent of forensic traceability or data exposure.

Bitwarden, because of its cloud synchronization capabilities, offers significant usability advantages, including seamless access across multiple devices and automatic vault backups. However, this convenience introduces a broader attack surface and more dispersed forensic artifacts. The presence of cached data, logs, and synchronization states on various devices (including browser extensions) means that sensitive, although encrypted, information might reside in more locations, potentially increasing the complexity of forensic investigations. While the data

is encrypted, the sheer volume and distribution of potential remnants require a more extensive and multi-faceted forensic acquisition strategy. The trade-off here is enhanced accessibility at the cost of a potentially larger and more varied digital footprint.

Conversely, KeePassXC, with its emphasis on local storage via the `.kdbx`file, offers a more contained forensic profile. The primary source of sensitive data is the encrypted database file itself, limiting the distribution of forensic artifacts to the systems where the file is stored or accessed. This design inherently reduces the attack surface associated with cloud infrastructure and external synchronization services. However, this model places a greater burden on the user for backup and synchronization management, potentially leading to less frequent backups or less seamless cross-device access. The trade-off for KeePassXC is heightened control over data locality and a simpler forensic scope, but at the expense of some user convenience.

From a security perspective, both applications demonstrate robust cryptographic implementations, though their default choices of KDFs (PBKDF2 for Bitwarden, Argon2 for KeePassXC) offer different levels of resilience against offline cracking attacks. KeePassXC's use of Argon2, with its memory-hardness, generally provides a stronger defense against resource-rich adversaries. This highlights a critical implication: while both are secure when strong master passwords are used, KeePassXC's cryptographic design offers an additional layer of resistance against brute-force attacks.

Some implications and recommendations are:

- **For Users**: Bitwarden users should be aware of the distributed nature of their encrypted data and ensure that strong security practices are applied consistently across all synchronized devices and browsers. Additionally, it is recommended to change the KDF to Argon2id for enhanced protection. Users of KeePassXC benefit from a more confined data footprint but must diligently manage their `.kdbx` file backups and secure storage. Regardless of the application, strong, unique master passwords is perhaps the most important factor.
- **For Digital Forensic Investigators**: Forensic acquisition strategies must be tailored to the application's architecture. For Bitwarden, a comprehensive approach encompassing cloud data, all synchronized devices, and browser artifacts is essential. For KeePassXC, the focus will primarily be on locating and analyzing the `.kdbx` file and associated memory dumps.

- **For Developers**: The findings suggest that while usability features like cloud sync are valuable, developers should continue to explore mechanisms to minimize the persistence and exposure of sensitive encrypted data on local file systems and in volatile memory, even if temporarily. Further research into and adoption of memory-hard KDFs like Argon2, where appropriate, will enhance the resilience of password managers against offline attacks.

7.4 SUMMARY

Both Bitwarden and KeePassXC exhibit security trade-offs. Bitwarden's emphasis on being an online and user friendly option is countered by possible vulnerabilities in file storage and memory management. KeePassXC's robust key derivation and strong choice of default KDF is offset by risks associated with the header extraction. As a primary offline password manager, it lacks certain usability features and conveniences offered by Bitwarden and is also not present the the same number of platforms.

- Bitwarden: Very user friendly, potential vulnerabilities in file extraction and memory management. Online password manager, cross-platform usability.
- KeePassXC: Robust key derivation, risks associated with and `.kdbx` file header extraction. Offline password manager, PC-focused.

Ultimately, the choice of password manager involves balancing these trade-offs based on specific security priorities and risk assessments.

8 CONCLUSION

This report aimed to comprehensively investigate the security and forensic implications of two widely used open-source password managers, Bitwarden and KeePassXC, particularly within desktop environments. The study simulated realistic forensic scenarios to assess the extent of data recoverability and the types of sensitive information that can be retrieved post-usage.

The research successfully identified various residual artifacts from both applications, shedding light on how their architectural choices — cloud-based for Bitwarden and local storage for KeePassXC — impact data recoverability. Key findings from the forensic analysis revealed that Bitwarden's desktop application stores critical metadata and the master password hash in a `data.json` file, defaulting to PBKDF2-SHA256 for key derivation. In contrast, KeePassXC stores all data locally in encrypted database files, not relying on external servers.

We also evaluated the effectiveness of contemporary forensic tools and hash-cracking techniques against encrypted artifacts. It was demonstrated that despite encryption, sensitive data fragments could persist in system memory, temporary files, and other application traces. The feasibility of cracking password hashes was assessed using dictionary-based attacks and memory-derived wordlists, highlighting the continued relevance of password strength and unpredictability in forensic recovery. This aligns with the understanding that contextually enriched wordlists can significantly improve cracking success rates as seen in Kanta, Coisel, et al. (2023).

A significant aspect of this work was the discussion of security trade-offs between usability and forensic traceability. While password managers can enhance user convenience, their design decisions inherently influence the digital footprint left behind, which can be crucial for digital investigations. The analysis strongly suggests adopting more robust KDFs like Argon2, as implemented by KeePassXC, for enhanced security over PBKDF2, which Bitwarden uses by default.

The major contributions of this report include:

- A hands-on forensic analysis framework specifically applied to password manager applications.

- The development of custom scripts to extract and analyze artifacts, particularly from browser extensions and memory dumps, when default tools proved insufficient.
- A comparative analysis of forensic artifacts found in different operating system environments (Windows, Linux) and across various application components (browser extensions, volatile memory).
- An evaluation of the feasibility of cracking password hashes using dictionary-based attacks.

8.1 FUTURE WORK

Future research could expand upon the findings by:

- Investigating the available mobile versions of Bitwarden and various KeePassXC-compatible applications to assess their functionality, security implementations, and compatibility across different platforms.
- Investigating a broader range of password manager applications, including closed-source and commercial solutions, to provide a more comprehensive view of the landscape.
- Exploring advanced forensic techniques beyond those covered in this study, such as hardware-based memory acquisition or more sophisticated data carving methods.
- Analyzing the forensic implications of emerging authentication technologies like FIDO Passkeys in greater detail, as their adoption becomes more widespread.
- Developing automated tools and methodologies for cross-platform forensic artifact correlation, which could further streamline digital investigations involving multiple devices or operating systems.
- Conducting longitudinal studies to observe how design changes and updates in password manager applications affect the persistence and recoverability of forensic artifacts over time.

BIBLIOGRAPHY

1Password (2025). *How password managers work | 1Password*. 1Password Blog. Section: 1Password. URL: https://blog.1password.com/how-password-managers-work/ (visited on 2025-03-31).

Aas, Josh et al. (2019). "Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, pp. 2473–2487. ISBN: 9781450367479. DOI: 10.1145/3319535.3363192. URL: https://doi.org/10.1145/3319535.3363192.

Alliance, Fido (2025). *Passkeys: Passwordless Authentication | FIDO Alliance*. URL: https://fidoalliance.org/passkeys/ (visited on 2025-06-16).

Argon2-cffi 23.1.0 documentation (2025). URL: https://argon2-cffi.readthedocs.io/en/stable/api.html#argon2.PasswordHasher (visited on 2025-03-09).

benign-key-logger (Feb. 10, 2025). original-date: 2021-07-28T19:54:02Z. URL: https://github.com/Ga68/benign-key-logger (visited on 2025-03-12).

Biryukov, Alex, Daniel Dinu, and Dmitry Khovratovich (2017). *phc-winner-argon2/argon2-specs.pdf*. GitHub. URL: https://github.com/P-H-C/phc-winner-argon2/blob/master/argon2-specs.pdf (visited on 2017-03-24).

Biryukov, Alex, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson (Sept. 2021). *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*. RFC 9106. DOI: 10.17487/RFC9106. URL: https://www.rfc-editor.org/info/rfc9106.

Bitwarden (2025). *Bitwarden - How End-to-End Encryption Paves the Way for Zero Knowledge*. Bitwarden. URL: https://bitwarden.com/resources/zero-knowledge-encryption-white-paper/ (visited on 2025-03-31).

Bitwarden - GitHub page (June 1, 2025). GitHub. URL: https://github.com/bitwarden (visited on 2025-06-01).

Bitwarden - Password Manager – Apps on Google Play (2025). URL: https://play.google.com/store/apps/details?id=com.x8bit.bitwarden&hl=pt_PT (visited on 2025-06-17).

Bitwarden Blog (2024). Bitwarden. URL: https://bitwarden.com/blog/new-deployment-option-for-self-hosting-bitwarden/ (visited on 2024-12-09).

Bitwarden Client's source code (Dec. 9, 2024). original-date: 2016-03-09T23:14:01Z. URL: https://github.com/bitwarden/clients (visited on 2024-12-09).

Bitwarden Passkey Log In (2024). Bitwarden. URL: https://bitwarden.com/help/login-with-passkeys/ (visited on 2024-12-09).

Bitwarden-Forensic-Tools/chrome_ extension_data_parser.py (2025). URL: https://github.com/CamposTaPro/Bitwarden-Foresensic-Tools/blob/main/chrome_extension_data_parser.py (visited on 2025-06-09).

Bitwarden-Forensic-Tools/extract_masterPassword_memory.py (2025). URL: https://github.com/CamposTaPro/Bitwarden-Foresensic-Tools/blob/main/extract_masterPassword_memory.py (visited on 2025-06-09).

Bitwarden-Forensic-Tools/sql_extract.py (2025). URL: https://github.com/CamposTaPro/Bitwarden-Foresensic-Tools/blob/main/sql_extract.py (visited on 2025-06-09).

bitwarden2hashcat.py · GitHub (2024). URL: https://github.com/hashcat/hashcat/blob/master/tools/bitwarden2hashcat.py (visited on 2024-12-09).

BLAKE2 (2025). URL: https://www.blake2.net/ (visited on 2025-06-02).

CamposTaPro/Keepass2John (Jan. 30, 2025). original-date: 2025-01-30T15:58:46Z. URL: https://github.com/CamposTaPro/Keepass2John (visited on 2025-03-17).

ChatGPT (2025). URL: https://chatgpt.com (visited on 2025-07-06).

Chatzoglou, Efstratios et al. (Mar. 2024). *Keep your memory dump shut: Unveiling data leaks in password managers*. DOI: 10.48550/arXiv.2404.00423. arXiv: 2404.00423[cs].

Chen, Lily (Feb. 2, 2024). *Recommendation for key derivation using pseudorandom functions*. NIST SP 800-108r1-upd1. Gaithersburg, MD: National Institute of Standards and Technology (U.S.), NIST SP 800–108r1–upd1. DOI: 10.6028/NIST.SP.800-108r1-upd1. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-108r1-upd1.pdf (visited on 2025-04-01).

Client to Authenticator Protocol (CTAP) (2025). URL: https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html (visited on 2025-06-16).

Clipboard in arboard - Rust (2025). URL: https://openrr.github.io/openrr/arboard/struct.Clipboard.html (visited on 2025-06-02).

clipboard.rs (2025). URL: https://github.com/bitwarden/clients/blob/79fa246df29c9ed70e0e59d801c0ac1d93ec5e87/apps/desktop/desktop_native/core/src/clipboard.rs#L21 (visited on 2025-06-02).

ClipView (2025). URL: http://www.peterbuettner.de/develop/tools/clipview/index.html (visited on 2025-03-12).

CNBC (Feb. 17, 2023). *CNBC: Why passkeys from Apple, Google, Microsoft may soon replace your passwords - FIDO Alliance*. Section: FIDO in the News. URL: https://fidoalliance.org/cnbc-why-passkeys-from-apple-google-microsoft-may-soon-replace-your-passwords/ (visited on 2025-06-16).

CopyQ (2025). URL: https://hluk.github.io/CopyQ/ (visited on 2025-03-01).

Demystifying LevelDB (Sept. 17, 2019). SenX. URL: https://blog.senx.io/demystifying-leveldb/ (visited on 2025-01-08).

Documenting KeePass KDBX4 file format (2025). URL: https://palant.info/2023/03/29/documenting-keepass-kdbx4-file-format/ (visited on 2025-03-16).

FIDO Authentication Specifications | FIDO Alliance (2025). URL: https://fidoalliance.org/specifications/download/ (visited on 2025-06-16).

Gallus, Petr, Dominik Staněk, and Ivo Klaban (Mar. 2025). "Security Evaluation of Password Managers: A Comparative Analysis and Penetration Testing of Existing Solutions". In: *International Conference on Cyber Warfare and Security* 20, pp. 105–113. DOI: 10.34190/iccws.20.1.3330.

Gray, Joshua, Virginia N. L. Franqueira, and Yijun Yu (Sept. 2016). "Forensically-Sound Analysis of Security Risks of Using Local Password Managers". In: 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW). Beijing, China: IEEE, pp. 114–121. ISBN: 978-1-5090-3694-3. DOI: 10.1109/REW.2016.034.

Hähni, Sascha David (2023). "Password Managers in Digital Forensics". PhD thesis. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:su:diva-219709.

Hashcat (2024). URL: https://hashcat.net/hashcat/ (visited on 2024-12-09).

Help Center (2024). Bitwarden. URL: https://bitwarden.com/help/ (visited on 2024-12-09).

HxD (2024). URL: https://mh-nexus.de/en/hxd/ (visited on 2024-12-09).

Issue #1516 · bitwarden/clients (2024). GitHub. URL: https://github.com/bitwarden/clients/issues/1516 (visited on 2024-12-15).

Issue #6231 · bitwarden/clients (2024). GitHub. URL: https://github.com/bitwarden/clients/issues/6231 (visited on 2024-12-10).

Issue #7335 (2025). URL: https://github.com/keepassxreboot/keepassxc/issues/7335.

Issue #8939 (2025). GitHub. URL: https://github.com/keepassxreboot/keepassxc/issues/8939 (visited on 2025-03-13).

jsonlz4 (2025). URL: https://fileinfo.com/extension/jsonlz4 (visited on 2025-06-01).

Kaliski, Burt (Sept. 2000). *PKCS #5: Password-Based Cryptography Specification Version 2.0*. Request for Comments RFC 2898. Internet Engineering Task Force. DOI: 10.17487/RFC2898.

Kanta, Aikaterini, Iwen Coisel, and Mark Scanlon (2023). "Harder, better, faster, stronger: Optimising the performance of context-based password cracking dictionaries". In: *Forensic Science International: Digital Investigation* 44. Selected papers of the Tenth Annual DFRWS EU Conference, p. 301507. ISSN: 2666-2817. DOI: https://doi.org/10.1016/j.fsidi.2023.301507.

— (2024). "A comprehensive evaluation on the benefits of context based password cracking for digital forensics". In: *Journal of Information Security and Applications* 84, p. 103809. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2024.103809.

Kanta, Aikaterini, Sein Coray, et al. (July 1, 2021). "How viable is password cracking in digital forensic investigation? Analyzing the guessability of over 3.9 billion real-world accounts". In: *Forensic Science International: Digital Investigation* 37, p. 301186. ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2021.301186.

KDBX - XML schema (2025). URL: https://keepass.info/help/download/KDBX_XML.xsd (visited on 2025-03-16).

KDBX File Format Specification - KeePass (2025). URL: https://keepass.info/help/kb/kdbx.html (visited on 2025-03-16).

KDF Algorithms (2025). Bitwarden. URL: https://bitwarden.com/help/kdf-algorithms/ (visited on 2025-03-09).

KeePass2 (2025). URL: https://keepass.info/index.html (visited on 2025-05-19).

KeePass2-kdbx-4-header-extract (Feb. 19, 2025). original-date: 2025-01-21T16:23:25Z. URL: https://github.com/CamposTaPro/KeePass2-kdbx-4-header-extract (visited on 2025-03-17).

KeePassXC - GitHub page (June 1, 2025). original-date: 2016-02-28T15:52:40Z. URL: https://github.com/keepassxreboot/keepassxc (visited on 2025-06-01).

KeePassXC Password Manager (2025). URL: https://keepassxc.org/ (visited on 2025-05-19).

KeePassXC Team (2025). *KeeShare*. URL: https://keepassxc.org/docs/KeePassXC_UserGuide#_database_sharing_with_keeshare (visited on 2025-06-03).

KeePassXC: User Guide (2025). URL: https://keepassxc.org/docs/KeePassXC_UserGuide (visited on 2025-03-01).

keepassxc/src/gui/Clipboard.cpp (2025). GitHub. URL: https://github.com/keepassxreboot/keepassxc/blob/31c0b2389007f85c03ae73870e3d6f2a648fa67b/src/gui/Clipboard.cpp (visited on 2025-01-02).

Lassak, Leona et al. (Aug. 2024). "Why Aren't We Using Passkeys? Obstacles Companies Face Deploying FIDO2 Passwordless Authentication". In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, pp. 7231–7248. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/lassak.

Leipold, Noam (Oct. 14, 2024). *Forensic analysis of bitwarden self-hosted server*. Synacktiv. URL: https://www.synacktiv.com/en/publications/forensic-analysis-of-bitwarden-self-hosted-server (visited on 2025-04-20).

Let's Encrypt (2024). URL: https://letsencrypt.org/ (visited on 2024-12-09).

LevelDB (Jan. 7, 2025). original-date: 2014-08-27T21:17:52Z. URL: https://github.com/google/leveldb (visited on 2025-01-07).

Luevanos, Carlos et al. (2017). "Analysis on the Security and Use of Password Managers". In: *2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 17–24. DOI: 10.1109/PDCAT.2017.00013.

Matzen, Alexander et al. (2025). "Challenges and Potential Improvements for Passkey Adoption—A Literature Review with a User-Centric Perspective". In: *Applied Sciences* 15.8. ISSN: 2076-3417. DOI: 10.3390/app15084414. URL: https://www.mdpi.com/2076-3417/15/8/4414.

More efficient IndexedDB storage in Chrome (2025). Chrome for Developers. URL: https://developer.chrome.com/docs/chromium/indexeddb-storage-improvements (visited on 2025-06-02).

Moriarty, Kathleen, Burt Kaliski, and Aneas Rusch (Jan. 2017). *PKCS #5: Password-Based Cryptography Specification Version 2.1*. Request for Comments RFC 8018. Num Pages: 40. Internet Engineering Task Force. DOI: 10.17487/RFC8018. URL: https://datatracker.ietf.org/doc/rfc8018 (visited on 2025-04-01).

NIST Special Publication 800-63B (2025). URL: https://pages.nist.gov/800-63-3/sp800-63b.html (visited on 2025-05-19).

openwall/john (Mar. 17, 2025). original-date: 2011-12-16T19:43:47Z. URL: https://github.com/openwall/john (visited on 2025-03-17).

Padalia, Hrithik et al. (2023). "A Study on Password Manager: Users' Perspective". In: *2023 International Conference on Computational Intelligence for Infor-*

mation, Security and Communication Applications (CIISCA), pp. 72–75. DOI: 10.1109/CIISCA59740.2023.00024.

Password Storage - OWASP Cheat Sheet Series (2025). URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#introduction (visited on 2025-03-09).

Percival, Colin (2009). "STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS". In: URL: https://api.semanticscholar.org/CorpusID:15333875.

Percival, Colin and Simon Josefsson (Aug. 2016). *The scrypt Password-Based Key Derivation Function*. RFC 7914. DOI: 10.17487/RFC7914. URL: https://www.rfc-editor.org/info/rfc7914.

Peslyak, Alexander (Jan. 2015). *yescrypt - a Password Hashing Competition submission*. Tech. rep. Password Hashing Competition. URL: https://password-hashing.net/submissions/specs/yescrypt-v2.pdf.

Plyvel - documentation (2025). URL: https://plyvel.readthedocs.io/en/latest/ (visited on 2025-01-08).

PRF WebAuthn - Bitwarden Blog (2024). Bitwarden. URL: https://bitwarden.com/blog/prf-webauthn-and-its-role-in-passkeys/ (visited on 2024-12-09).

Python (2024). Python.org. URL: https://www.python.org/ (visited on 2024-12-09).

Qt Documentation - XML Processing (2025). URL: https://www.surveymonkey.com/r/8XKW76J?pagename=%2Fqt-6%2Fxml-processing.html (visited on 2025-03-13).

Qt Documentation Feedback (2025). URL: https://www.surveymonkey.com/r/8XKW76J?pagename=%2Fqt-6%2Fqmimedata.html (visited on 2025-03-12).

Ray, Hirak et al. (Oct. 2020). *Why Older Adults (Don't) Use Password Managers*. DOI: 10.48550/arXiv.2010.01973.

Snappy (June 2, 2025). original-date: 2014-03-03T21:58:09Z. URL: https://github.com/google/snappy (visited on 2025-06-02).

Sqlitebrowser (2024). URL: https://sqlitebrowser.org/ (visited on 2024-12-09).

SSH Agent Integration (2025). URL: https://keepassxc.org/docs/KeePassXC_UserGuide#_database_sharing_with_keeshare (visited on 2025-06-03).

Stallings, William (2013). *Cryptography and Network Security: Principles and Practice*. 6th. USA: Prentice Hall Press. ISBN: 0133354695.

Storage | Bitwarden Help Center (2024). Bitwarden. URL: https://bitwarden.com/help/data-storage/ (visited on 2024-12-09).

strings (2024). URL: https://linux.die.net/man/1/strings (visited on 2024-12-09).

Support for Argon2id on NVIDIA CUDA GPUs · hashcat/hashcat (2025). GitHub. URL: https://github.com/hashcat/hashcat/pull/4284 (visited on 2025-07-12).

System Informer (2025). URL: https://systeminformer.com/ (visited on 2025-03-13).

Tips for secure user authentication | ENISA (2025). URL: https://www.enisa.europa.eu/news/enisa-news/tips-for-secure-user-authentication (visited on 2025-06-16).

VirtualBox (2024). URL: https://www.virtualbox.org/wiki/Downloads (visited on 2024-12-09).

Wash, Rick et al. (2016). "Understanding password choices: how frequently entered passwords are re-used across websites". In: *Proceedings of the Twelfth USENIX Conference on Usable Privacy and Security*. SOUPS '16. Denver, CO, USA: USENIX Association, pp. 175–188. ISBN: 9781931971317.

WebAuthn (2025). URL: https://www.w3.org/TR/webauthn-1/#sctn-authenticator-model (visited on 2025-06-02).

WebAuthn.io (2024). WebAuthn.io. URL: https://webauthn.io (visited on 2024-12-09).

xclip (2025). URL: https://github.com/astrand/xclip (visited on 2025-03-12).

APPENDICES

APPENDIX A

This script was used to extract metadata from the header of the KeePassXC database, the script is available at *KeePass2-kdbx-4-header-extract* (2025).

```
1  import struct
2  import argparse
3  def bytes_to_mib(byte_value):
4      return byte_value / (1024 * 1024)   # Divide by 2^20 (1,048,576)
5  def main():
6      parser = argparse.ArgumentParser(description="Read and parse a KDBX file.")
7      parser.add_argument("filename", type=str, help="Path to the KDBX file to be
   ↪   read.")
8      args = parser.parse_args()
9      data, varientMapData=readFile(args.filename)
10     print_values(data,varientMapData)
11 def read_fixed_size(file, size, unpack_format=None, hex_output=False):
12     data = file.read(size)
13     if unpack_format:
14         return struct.unpack(unpack_format, data)[0]
15     if hex_output:
16         return data.hex()
17     return data
18 def read_fixed_size_string(file,size):
19     data = file.read(size)
20     return data.decode("utf-8")
21 def read_varientMap(file):
22     varientDictionaryFormat=read_fixed_size(file,2,"<H")
23     uuidKDFType= file.read(1) # not used for anything but it still needs to be
   ↪   read
24     uuidKDFNameSize=read_fixed_size(file,4,"<I")
25     uuidKDFName=read_fixed_size_string(file,uuidKDFNameSize)
26     uuidKDFValueSize=read_fixed_size(file,4,"<I")
27     uuidKDFValue=read_fixed_size(file,uuidKDFValueSize,None,True)
28     if uuidKDFValue in
   ↪   ["ef636ddf8c29444b91f7a9a403e30a0c","9e298b1956db4773b23dfc3ec6f0a1e6"]:
29         kdf=1              #this variable is only to help distiguish the KDFs -> 1
   ↪       - argon2d or argon2id and 2 - AES-KDF
30         ######## For entry I (Iterations)   #################
31         argon2IterationType= file.read(1)
32         argon2IterationNameSize=read_fixed_size(file,4,"<I")
33         argon2IterationName=read_fixed_size_string(file,argon2IterationNameSize)
34         argon2IterationValueSize=read_fixed_size(file,4,"<I")
35         argon2IterationValue=read_fixed_size(file,argon2IterationValueSize,"<Q")
```

```
36          ######## For entry M (Memory)  #################
37          argon2MemoryType= file.read(1)
38          argon2MemoryNameSize=read_fixed_size(file,4,"<I")
39          argon2MemoryName=read_fixed_size_string(file,argon2MemoryNameSize)
40          argon2MemoryValueSize=read_fixed_size(file,4,"<I")
41          argon2MemoryValue=read_fixed_size(file,argon2MemoryValueSize,"<Q")
42          ######## For entry P (Parallelism)  #################
43          argon2ParallelismTypeByte= file.read(1)
44          argon2ParallelismNameSize=read_fixed_size(file,4,"<I")
45          argon2ParallelismName=read_fixed_size_string(file,argon2ParallelismNameS
            ↪  ize)
46          argon2ParallelismValueSize=read_fixed_size(file,4,"<I")
47          argon2ParallelismValue=read_fixed_size(file,argon2ParallelismValueSize,"
            ↪  <I")
48          ######## For entry S (Salt)  #################
49          argon2SaltTypeByte= file.read(1)
50          argon2SaltNameSize=read_fixed_size(file,4,"<I")
51          argon2SaltName=read_fixed_size_string(file,argon2SaltNameSize)
52          argon2SaltValueSize=read_fixed_size(file,4,"<I")
53          argon2SaltValue=read_fixed_size(file,argon2SaltValueSize,None,True)
54          ######## For entry V (Version)  #################
55          argon2VersionTypeByte= file.read(1)
56          argon2VersionNameSize=read_fixed_size(file,4,"<I")
57          argon2VersionName=read_fixed_size_string(file,argon2VersionNameSize)
58          argon2VersionValueSize=read_fixed_size(file,4,"<I")
59          argon2VersionValue=read_fixed_size(file,argon2VersionValueSize,"<I",True)
60          ######## Check for end of VariantMap #################
61          checkEnd=read_fixed_size(file,1,None,True)
62          data={
63              "varientDictionaryFormat": hex(varientDictionaryFormat),
64              "uuidKDFValueSize":uuidKDFValueSize,
65              "uuidKDFName":uuidKDFName,
66              "uuidKDFValueSize":uuidKDFValueSize,
67              "uuidKDFValue":uuidKDFValue,
68              "argon2IterationValueSize":argon2IterationValueSize,
69              "argon2IterationName":argon2IterationName,
70              "argon2IterationValue":argon2IterationValue,
71              "argon2MemoryValueSize":argon2MemoryValueSize,
72              "argon2MemoryName":argon2MemoryName,
73              "argon2MemoryValue":argon2MemoryValue,
74              "argon2ParallelismValueSize":argon2ParallelismValueSize,
75              "argon2ParallelismName":argon2ParallelismName,
76              "argon2ParallelismValue":argon2ParallelismValue,
77              "argon2SaltValueSize":argon2SaltValueSize,
78              "argon2SaltName":argon2SaltName,
79              "argon2SaltValue":argon2SaltValue,
80              "argon2VersionValueSize":argon2VersionValueSize,
81              "argon2VersionName":argon2VersionName,
82              "argon2VersionValue":hex(argon2VersionValue),
83              "checkEnd":checkEnd,
84              "kdf":kdf}
85      else:
86          kdf=2
```

```
87          ######## For entry R (Rounds)  #################
88          aesRoundsType= file.read(1)
89          aesRoundsNameSize=read_fixed_size(file,4,"<I")
90          aesRoundsName=read_fixed_size_string(file,aesRoundsNameSize)
91          aesRoundsValueSize=read_fixed_size(file,4,"<I")
92          aesRoundsValue=read_fixed_size(file,aesRoundsValueSize,"<Q")
93          ######## For entry S (Salt/seed)  #################
94          aesSaltType= file.read(1)
95          aesSaltNameSize=read_fixed_size(file,4,"<I")
96          aesSaltName=read_fixed_size_string(file,aesSaltNameSize)
97          aesSaltValueSize=read_fixed_size(file,4,"<I")
98          aesSaltValue=read_fixed_size(file,aesSaltValueSize,None,True)
99          ######## Check for end of VariantMap #################
100         checkEnd=read_fixed_size(file,1,None,True)
101         data={
102             "varientDictionaryFormat": hex(varientDictionaryFormat),
103             "uuidKDFValueSize":uuidKDFValueSize,
104             "uuidKDFName":uuidKDFName,
105             "uuidKDFValueSize":uuidKDFValueSize,
106             "uuidKDFValue":uuidKDFValue,
107             "kdf":kdf,
108             "aesRoundsName":aesRoundsName,
109             "aesRoundsValueSize":aesRoundsValueSize,
110             "aesRoundsValue":aesRoundsValue,
111             "aesSaltName":aesSaltName,
112             "aesSaltValueSize":aesSaltValueSize,
113             "aesSaltValue":aesSaltValue,
114             "checkEnd":checkEnd
115         }
116     return data
117 def readFile(path_file):
118     with open(str(path_file), "rb") as file: #teste123.kdbx - AES
        ↪  Passwords.kdbx- argon2d
119         ######## Read Signtures #################
120         signature1=read_fixed_size(file,4,"<I")
121         signature2=read_fixed_size(file,4,"<I")
122         signature={
123             "signature1":hex(signature1),
124             "signature2":hex(signature2)
125         }
126         ######## Read Version of the kdbx file #################
127         minor_version = read_fixed_size(file, 2, "<H")
128         major_version = read_fixed_size(file, 2, "<H")
129         version = f"{major_version}.{minor_version}"
130
131         ######## Read the ID of the algorithm used to cipher #################
132         CipherIDFlag=read_fixed_size(file,1,"<B")
133         CipherIDSize=read_fixed_size(file,4,"<I")
134         CipherID=read_fixed_size(file,CipherIDSize,None,True)# important usar o
            ↪  .hex() quando so quero extrair o valor hexadecimal
135         cipher={
136             "CipherIDFlag":CipherIDFlag,
137             "CipherIDSize": CipherIDSize,
138             "CipherID": CipherID
```

```
139         }
140         ######## Read the compression algorithm used #################
141         compressionFlag=read_fixed_size(file,1,"<B")
142         compressionSize=read_fixed_size(file,4,"<I")
143         compression=read_fixed_size(file,compressionSize,"<I")
144         compressionAlgorithm={
145             "compressionFlag":compressionFlag,
146             "compressionSize":compressionSize,
147             "compression":compression
148         }
149         ######## Read the salt used for computing the keys #################
150         saltFlag=read_fixed_size(file,1,"<B")
151         saltSize=read_fixed_size(file,4,"<I")
152         salt=read_fixed_size(file,saltSize,None,True)
153         saltData={
154             "saltFlag":saltFlag,
155             "saltSize":saltSize,
156             "salt":salt
157         }
158         ######## Read the initialization vector for the encryption algorithm.
            ↪  #################
159         encryptionIVFlag=read_fixed_size(file,1,"<B")
160         encryptionIVSize=read_fixed_size(file,4,"<I")
161         encryptionIV=read_fixed_size(file,encryptionIVSize,None,True)# means the
            ↪  encryption algorithm is AES meaning this field is 16 bytes long
162         encryptionInitializationVector={
163             "encryptionIVFlag":encryptionIVFlag,
164             "encryptionIVSize":encryptionIVSize,
165             "encryptionIV":encryptionIV
166         }
167         ######## Read the parameters for the key derivation function (KDF)
            ↪  #################
168         kdfParamFlag=read_fixed_size(file,1,"<B")
169         kdfParamSize=read_fixed_size(file,4,"<I")
170         kdfParameters={
171             "kdfParamFlag":kdfParamFlag,
172             "kdfParamSize":kdfParamSize
173         }
174        # print(kdfParameters)
175         ######## Read the varient map field #################
176         varientMapData=read_varientMap(file)
177         ######## Check for end of Headers #################
178         headerEndFlag=read_fixed_size(file,1,"<B")
179         headerEndSize=read_fixed_size(file,4,"<I")
180         headerEnd=read_fixed_size(file,headerEndSize,None,True) # estava fazer
            ↪  unpack ao bytes mas como so queria comparar o valor hex nao era
            ↪  preciso fazer nada
181     data={
182         "signature":signature,
183         "version":version,
184         "cipher":cipher,
185         "compressionAlgorithm":compressionAlgorithm,
186         "saltData":saltData,
187         "encryptionInitializationVector":encryptionInitializationVector,
```

```
188             "kdfParameters":kdfParameters,
189             "headerEndFlag":headerEndFlag,
190             "headerEndSize":headerEndSize,
191             "headerEnd":headerEnd
192         }
193         return data, varientMapData
194     def print_values(data,vmdata):
195         print("Value of the first signature (uint32, little-endian):",
            ↪  data['signature']['signature1'])
196         print("Value of the second signature (uint32, little-endian):",
            ↪  data['signature']['signature2'])
197         print("Format version:",data['version'])
198         if data['encryptionInitializationVector']['encryptionIVSize']==16:
199             print("ID",data['cipher']['CipherIDFlag'],"Encryption algorithm - AES-256
                ↪  (size",data['cipher']['CipherIDSize'],"bytes):",data['cipher']['Ciph
                ↪  erID'],"(hexdecimal)")
200         else:
201             print("ID",data['cipher']['CipherIDFlag'],"Encryption algorithm -
                ↪  ChaCha20 (size",data['cipher']['CipherIDSize'],"bytes):",data['ciphe
                ↪  r']['CipherID'],"(hexdecimal)")
202         if data['compressionAlgorithm']['compression']==1:
203             print("ID",data['compressionAlgorithm']['compressionFlag'],"Compression
                ↪  algorithm (size",data['compressionAlgorithm']['compressionSize'],"by
                ↪  tes):",data['compressionAlgorithm']['compression'], "(using
                ↪  GZIP)")
204         else:
205             print("ID",data['compressionAlgorithm']['compressionFlag'],"Compression
                ↪  algorithm (size",data['compressionAlgorithm']['compressionSize'],"by
                ↪  tes):",data['compressionAlgorithm']['compression'], "(no
                ↪  compression)")
206         print("ID",data['saltData']['saltFlag'],"Master salt/seed (size",data['saltD
            ↪  ata']['saltSize'],"bytes):",data['saltData']['salt'],"(hexadecimal)")
207         print("ID",data['encryptionInitializationVector']['encryptionIVFlag'],"Encry
            ↪  ption IV/nonce
            ↪  (size",data['encryptionInitializationVector']['encryptionIVSize'],"bytes
            ↪  ):",data['encryptionInitializationVector']['encryptionIV'],"(hexadecimal
            ↪  )")
208         print("ID",data['kdfParameters']['kdfParamFlag'],"KDF parameters
            ↪  (size",data['kdfParameters']['kdfParamSize'],"bytes) - Variant
            ↪  dictionary:")
209         print("Format version:",vmdata['varientDictionaryFormat'])
210         print("KDF algorithm:\n entry name ->",vmdata['uuidKDFName'],"\n size
            ↪  ->",vmdata['uuidKDFValueSize'],"\n value ->",vmdata['uuidKDFValue'])
211         if vmdata['kdf'] == 1:
212             print("This means the used KDF ALgorithm used was Argon2")
213             print("Argon2 Iterations:\n entry name
                ↪  ->",vmdata['argon2IterationName'],"\n size
                ↪  ->",vmdata['argon2IterationValueSize'],"\n value
                ↪  ->",vmdata['argon2IterationValue'])
214             print("Argon2 Memory:\n entry name ->",vmdata['argon2MemoryName'],"\n
                ↪  size ->",vmdata['argon2MemoryValueSize'],"\n value
                ↪  ->",vmdata['argon2MemoryValue'],"bytes
                ↪  or",bytes_to_mib(vmdata['argon2MemoryValue']),"MiB")
```

```
215         print("Argon2 Parallelism:\n entry name
            ↪  ->",vmdata['argon2ParallelismName'],"\n size
            ↪  ->",vmdata['argon2ParallelismValueSize'],"\n value
            ↪  ->",vmdata['argon2ParallelismValue'],"threads")
216         print("Argon2 Salt:\n entry name ->",vmdata['argon2SaltName'],"\n size
            ↪  ->",vmdata['argon2SaltValueSize'],"\n value
            ↪  ->",vmdata['argon2SaltValue'],"(hexadecimal)")
217         print("Argon2 Version:\n entry name ->",vmdata['argon2VersionName'],"\n
            ↪   size ->",vmdata['argon2VersionValueSize'],"\n value
            ↪  ->",vmdata['argon2VersionValue'],"(hexadecimal)")
218     else:
219         print("This means the used KDF ALgorithm used was AES-KDF")
220         print("AES-KDF Rounds:\n entry name ->",vmdata['aesRoundsName'],"\n size
            ↪  ->",vmdata['aesRoundsValueSize'],"\n value
            ↪  ->",vmdata['aesRoundsValue'])
221         print("AES-KDF Salt/Seed:\n entry name ->",vmdata['aesSaltName'],"\n size
            ↪  ->",vmdata['aesSaltValueSize'],"\n value ->",vmdata['aesSaltValue'])
222     if vmdata['checkEnd'] =="00":
223         print("Reached the end of the Variant dictionary, continuing to read the
            ↪   headers of the file...")
224     else:
225         print("Something went wrong")
226     print("ID",data['headerEndFlag'],"End of headers reached!
        ↪   (size",data['headerEndSize'],"bytes):",data['headerEnd'])
227 if __name__ == "__main__":
228     main()
```

Listing 21: Code of the script `kdbx4_header_extractor.py`

DECLARATION

I declare, under oath of honor, that the work presented in this Project, entitled *"Forensic Analysis of Password Managers"*, is original and was carried out by Miguel Filipe Cunha Campos (2230465) under the supervision of Miguel Monteiro de Sousa Frade, Miguel Cerdeira Marreiros Negrão and Patrício Rodrigues Domingues.

Leiria, July of 2025

Miguel Filipe Cunha Campos