International Journal Software Engineering and Computer Science (IJSECS)
5 (2), 2025, 633-644
Published Online August 2025 in IJSECS (http://www.journal.lembagakita.org/index.php/ijsecs)
P-ISSN: 2776-4869, E-ISSN: 2776-3242. DOI: https://doi.org/10.35870/ijsecs.v5i2.4207.

RESEARCH ARTICLE **Open Access**

Implementation of Zero-Knowledge Encryption in a Web-Based Password Manager

R. Krisviarno Darmawan *
Informatics Engineering Study Program, Faculty of Information Technology, Universitas Kristen Satya Wacana, Salatiga City, Central Java Province, Indonesia.
Corresponding Email: 672021082@student.uksw.edu.

Ariya Dwika Cahyono
Informatics Engineering Study Program, Faculty of Information Technology, Universitas Kristen Satya Wacana, Salatiga City, Central Java Province, Indonesia.
Email: ariyadc@uksw.edu.

Received: May 7, 2025; Accepted: June 30, 2025; Published: August 1, 2025.

Abstract: The secure management of account credentials presents a considerable challenge in the digital era, as many users continue to engage in unsafe practices such as password reuse. Conventional password managers typically store encrypted data on servers, which introduces risks if those servers are compromised. This study develops a web-based password manager that implements Zero-Knowledge Encryption (ZKE), ensuring that all essential cryptographic operations are executed exclusively on the client side (browser). Employing a client-server architecture (React frontend, Python/FastAPI backend), the system derives encryption keys from the user's master password using Argon2id (4 iterations, 64 MB memory, 1 parallelism), and performs credential data encryption and decryption with AES-GCM entirely on the client side. The server is limited to receiving and storing encrypted data (verifier, salt, data blobs), without ever accessing the master password or plaintext credentials. Network payload analysis conducted with Chrome DevTools confirms that the ZKE implementation effectively prevents the exposure of sensitive data to the server. This approach substantially improves data privacy and security against server-side threats. Nevertheless, the ZKE model lacks an account recovery feature, placing full responsibility on users to protect their master passwords—a trade-off that underscores the need for further investigation into ZKE-compatible recovery mechanisms.

Keywords: Password Manager; Zero-Knowledge Encryption; Client-Side Encryption; Web Security; Argon2id; AES-GCM; Python; FastAPI.

1. Introduction

As daily activities increasingly require multiple application accounts, protecting personal data has become a major concern for internet users. The rise in online services means that people must manage a growing number of account credentials securely and efficiently. Survey results show that most users (60%) handle between one and ten online accounts, while 24% manage ten to twenty. Data also indicates that 74% of users reuse passwords across several accounts, and only 26% regularly change their passwords. This trend suggests that convenience often takes precedence over security. In addition, 46% of users include personal information in their passwords, and 29% rely on common dictionary words, making their passwords more vulnerable to attacks. Although the majority (56%) use passwords with lengths between 8 and 12 characters, frequent reuse and a lack of diversity in password composition continue to be significant weaknesses in digital security.

Password managers have emerged as a way to help users maintain multiple unique and complex passwords in a secure and practical manner [1].

Despite their benefits, conventional password managers typically store encrypted data on servers. If these servers are compromised, there remains a risk that encrypted passwords could be exposed. The main issue lies in the decryption process performed on the server side, which creates a moment when sensitive data may be accessible before further processing. Users also have to place their trust in service providers to manage encryption keys properly. These concerns have led to the adoption of Zero-Knowledge Encryption (ZKE), where even the service provider cannot access user data in its original form, helping to maintain privacy and security even if a server breach occurs. In cryptography, "Zero-Knowledge" refers to two distinct ideas: Zero-Knowledge Proofs (ZKP) and Zero-Knowledge Encryption (ZKE). Knowing the difference between these concepts is essential for understanding their roles in security systems. Zero-Knowledge Proofs (ZKP) allow one party (the prover) to convince another (the verifier) that a statement is true, without revealing anything beyond the validity of the statement itself. This method was first introduced in 1985 by Shafi Goldwasser, Silvio Micali, and Charles Rackoff, in their work "The Knowledge Complexity of Interactive Proof Systems." ZKP is widely used in authentication protocols that require identity verification without exposing sensitive details [2].

Zero-Knowledge Encryption (ZKE), on the other hand, ensures that only the person holding the decryption key—the user—can convert encrypted data back to its original form. All encryption and decryption take place on the client side, such as within the browser, so service providers never have access to the plaintext data. This approach keeps user privacy and security intact, even if the server storing encrypted data is breached. Given this distinction, Zero-Knowledge Encryption (ZKE) has been chosen as the foundation for research and development in building a web-based password manager. To implement ZKE in a web-based password manager, all cryptographic processes are handled within the user's browser. The server only receives and stores ciphertext, not encryption keys or other sensitive information. Even if the server is compromised, the original data remains protected because the keys never leave the user's device. This client-side encryption model, however, requires careful attention to the JavaScript code responsible for encryption and decryption, as well as secure key management, to prevent threats like Man-in-the-Middle (MITM) attacks or code injection. With ZKE's ability to address the limitations of conventional models and its unique challenges on web platforms, this research aims to design, build, and evaluate a web-based password manager that applies Zero-Knowledge Encryption principles throughout.

2. Related Work

Several studies have examined the security and usability challenges of password managers. In "Why People (Don't) Use Password Managers Effectively", researchers identified critical vulnerabilities that affect user data security in password manager applications. One notable issue is the storage of passwords in temporary folders in plaintext, making them easily accessible to third parties. The study also highlights that, without robust end-to-end encryption, sensitive data remains at risk during both processing and storage. To address these concerns, Zero-Knowledge Encryption (ZKE) in web-based password managers is proposed as a relevant solution. With ZKE, passwords and other sensitive information are encrypted on the client side before being transmitted to the server. This ensures that only users hold the encryption keys, preventing servers from accessing user data and thereby increasing user trust in sensitive data management [3].

Beyond technical vulnerabilities, user behavior also plays a significant role in password security. The study "Adopting Password Manager Applications among Smartphone Users" surveyed participants about their daily password management practices. The majority admitted to reusing a single main password across multiple sites or making only minor modifications, a practice that exposes many accounts if the password is compromised. Furthermore, 30% of respondents acknowledged using personal information, such as pet names or birth dates, to make passwords easier to remember. Additionally, 17% stored passwords in browser-based managers, and another 17% recorded them in physical notebooks. These habits reflect a tendency to prioritize convenience over security, underscoring the need for password managers with ZKE to store unique passwords securely, without risking plaintext storage on devices or browsers [4].

Trust issues regarding password manager providers are also significant. In "Password Managers—It's All about Trust and Transparency", 134 out of 247 online survey participants reported not using password managers. The main reason, cited by 41.8% of respondents, was a lack of trust in third-party providers to securely store passwords. Other reasons included insufficient transparency, with 38.1% stating they did not know where their passwords would be stored, and 22.4% unsure of how their online passwords would be processed. Security concerns were also prominent: 35.8% feared all their passwords would be leaked if the database was hacked, and 26.1% worried that if their master password was compromised, all their stored passwords would be exposed [5].

To address technical and trust challenges, recent works have proposed improved security models. For example, "An Enhanced Web Security for Cloud-Based Password Management" introduces a cloud-based password manager that implements cryptographic hash functions (SHA-256) and Diffie-Hellman key exchange to achieve ZKE. By encrypting data on the client side before transmission, the server never gains access to plaintext, significantly reducing the risk of sensitive data theft [6]. A comparative analysis of paid password managers, as presented in "Analisis Komparatif Keamanan Aplikasi Pengelola Kata Sandi Berbayar Lastpass, 1Password, dan Keeper Berdasarkan ISO/IEC 25010", provides insight into the algorithms used by popular desktop password managers. Table 1 summarizes the comparison:

Table 1. Desktop Password Manager Comparison

Name	Lastpass	1Password	Keeper
Algorithm	AES-256 (CBC) + PBKDF2-SHA256 (100,100)	AES-256 (GCM) + PBKDF2 (100,000) - HMAC-SHA256	AES-256 + PBKDF2 (100,000)

All three applications use AES-256 bit encryption for user data. The differences lie in the encryption mode, the number of PBKDF2 rounds, and the hash functions used [7]. For browser-based password managers, "That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers" presents an overview of encryption practices:

Table 2. Browser Password Manager Comparison

System	Encryption	KDF	KDF Rounds	Storage
Chrome	OS Dependent	-	-	File (.sqlite)
Edge	Windows Vault	-	-	Windows Vault
Firefox	3DES	SHA-1	1	File (.json)
IE	Windows Vault	-	-	Windows Vault
Opera	OS Dependent	-	-	File (.sqlite)
Safari	OS X Keychain	-	-	OS X Keychain

According to Table 2, only Firefox performs its own encryption on storage, using 3DES and SHA-1 for key generation. Other browsers delegate encryption to the operating system. On Windows, Chrome and Opera use CryptProtectData, which ties the encryption key to the active user account; on Linux, they use GNOME Keyring or KWallet, and if both fail, passwords are stored as plaintext. On macOS, Apple Keychain is used. Because encryption is managed outside the browser, users of Chrome, Edge, Opera, and Safari are not offered the option to set a master password. The security of the vault is limited to the security of the OS account, while Firefox enables additional protection with a master password managed directly by the browser [8].

3. Research Method

To address the objectives and fill the gaps identified in previous studies, this research focuses on applying Zero-Knowledge Encryption (ZKE) in a web-based password manager developed in Python. The development process follows the Agile Software Development Life Cycle (SDLC), which supports iterative progress and regular improvements based on feedback throughout each stage. Agile is well-suited for projects that require flexibility and quick adaptation [9]. For example, when choosing the encryption algorithm, several options are tested and swapped as needed before settling on a final approach that best fits the project’s requirements. Python was chosen for its extensive cryptography libraries, which simplify the implementation of complex algorithms. Its clean, concise syntax also helps streamline development within the Agile framework.

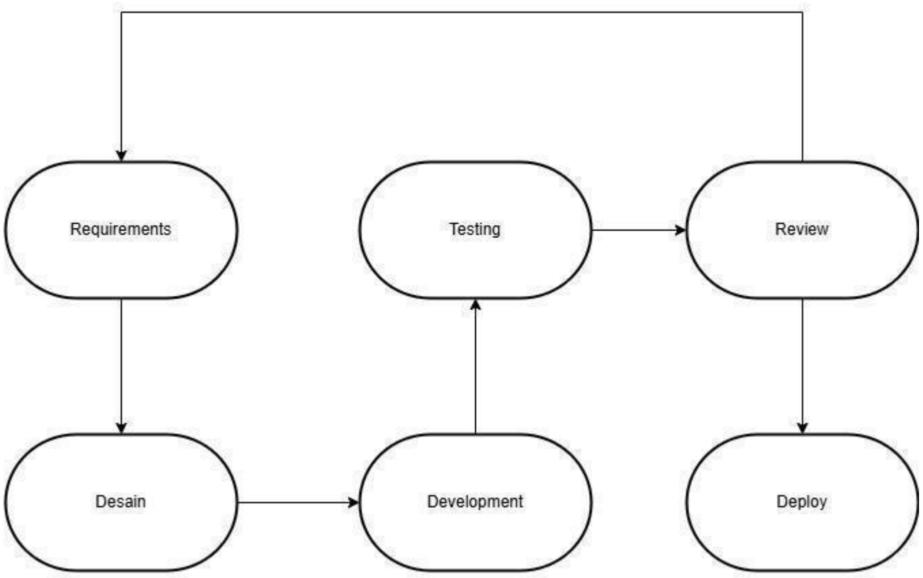


Figure 1. Agile Development Approach

Figure 1 outlines the main stages of application development. Work progresses in sprints, each building on the previous iteration. The initial sprint focuses on setting up the core architecture, using FastAPI for the backend—selected for its speed and ability to build robust APIs—and configuring the database. For the frontend, React and Vite are used. Once the backend, frontend, and database are able to communicate reliably, the next step is to implement data handling with Zero-Knowledge Encryption. Here, passwords are encrypted on the client side before being sent to the server. After the basic configuration is complete, additional security measures are put in place.

Since most cryptographic operations run on the client side, extra care is taken to secure memory. Following techniques described in "Keep Your Memory Dump Shut: Unveiling Data Leaks in Password Managers," sensitive information is removed from memory after processing, encryption (such as AES-256) is applied, and data padding is used to reduce predictable patterns that attackers could exploit [10]. The application also provides a random password generator for users. The logic for this feature is inspired by "An Effective Mechanism For Securing And Managing Password Using AES-256 Encryption & PBKDF2." Passwords are created from a pool of 75 characters: uppercase and lowercase letters, digits (0-9), and special symbols (~!@#$%^&*-_+=). For a password with 12 characters, the total number of possible combinations is 75^12 [11]. Testing is carried out throughout development, not just at the end. Chrome Devtools are used to examine the code, especially the cryptographic functions and ZKE implementation, while black box testing checks functional aspects. OWASP ZAP is used for overall security assessment. OWASP ZAP was selected for penetration testing based on findings from "A Comparative Analysis of Web Application Vulnerability Tools."

Table 3. Penetration Testing Results [12]

	ZAP	Vega	Arachni
SQL Injection	7	6	5
XSS	34	10	29

Table 3 compares the detection of SQL Injection and Cross-Site Scripting (XSS) vulnerabilities among three tools. ZAP leads, identifying 7 SQL Injection cases and 34 XSS cases (including 18 DOM-based). In contrast, Arachni found 29 XSS issues and 5 SQL Injection cases, two of which were Blind SQL Injection. Vega reported 10 XSS and 6 SQL Injection vulnerabilities, plus 3 potential SQL Injection cases. ZAP consistently produced the most findings for both categories [12]. The chosen testing methods help quickly spot and resolve issues during each development cycle. Regular sprints keep reliability, security, and user experience at the forefront. This ongoing process maintains code quality and allows the team to respond promptly to new requirements or bugs, resulting in a password manager that is both secure and easy to use.

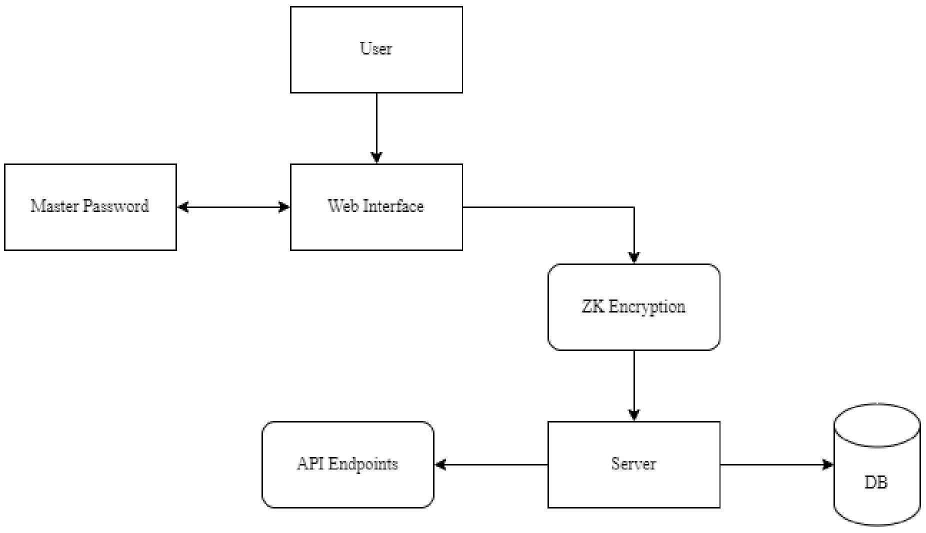


Figure 2. Basic Application Flow

Figure 2 shows the general workflow. When users access the application, they can register or log in. Registration involves creating a master password; login requires entering a username and master password. After successful authentication, users can manage their passwords. All sensitive data is encrypted locally in the browser before being sent to the server, thanks to ZKE. The backend handles requests through API endpoints and interacts with the database, which stores only encrypted passwords and non-sensitive metadata. This approach helps protect user privacy even if the server is compromised.

4. Result and Discussion

4.1 Results

This section explains the implementation and analysis of a web-based password manager that applies Zero-Knowledge Encryption (ZKE). The discussion covers system architecture, core and supporting features, and the cryptographic workflow across the frontend, backend, and database, all designed to align with ZKE principles. The application uses a client-server model. The backend, developed with FastAPI (Python), provides APIs for authentication and encrypted data storage. The frontend, built with React (JavaScript), manages the user interface and executes all cryptographic operations on the client side, including key derivation, encryption, decryption, and hashing. The PostgreSQL database only stores encrypted or hashed information; sensitive data in plaintext never reaches the server. Communication between frontend and backend is protected using JSON Web Tokens (JWT), which authenticate API requests after users log in [13].

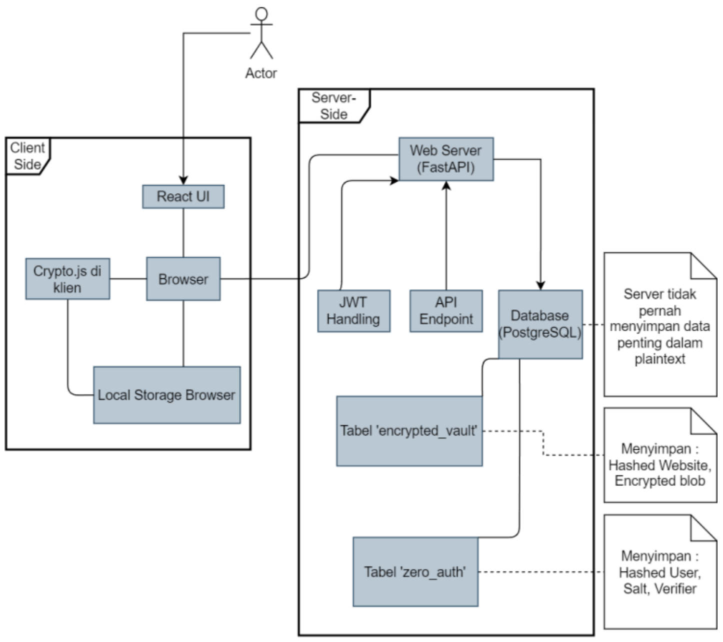
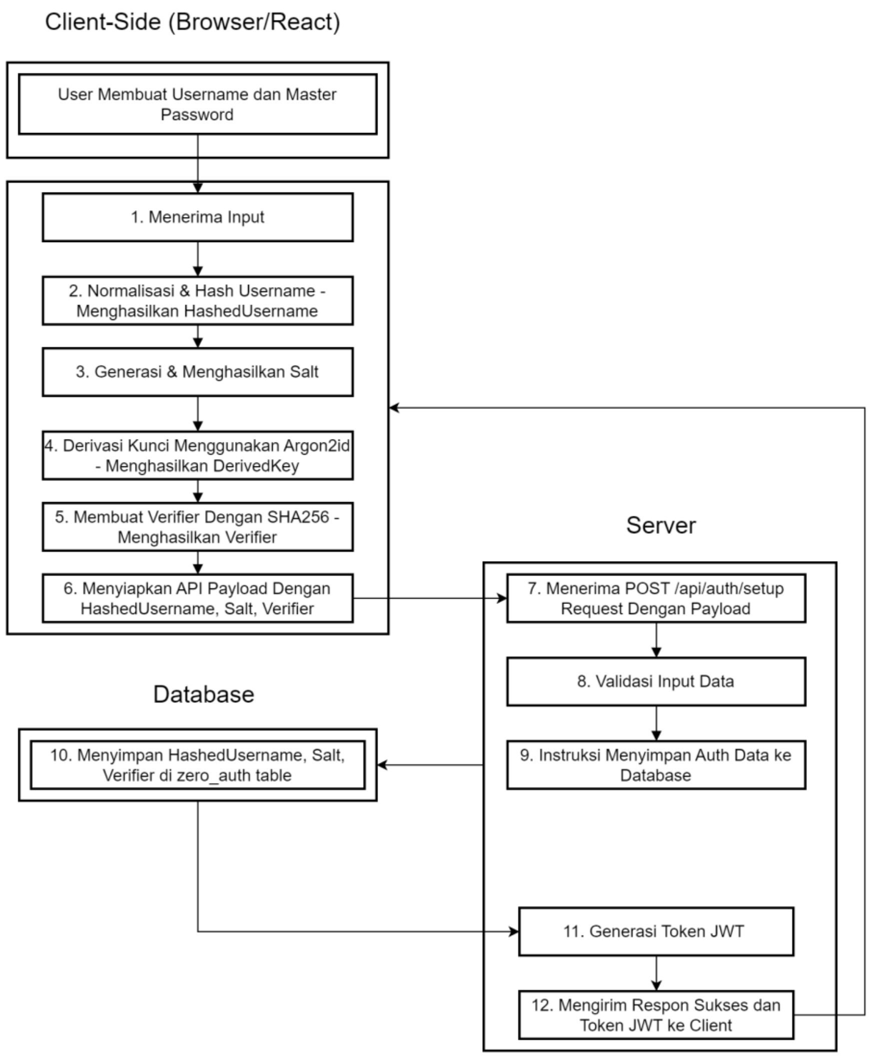


Figure 3. Program Flow

Figure 3 shows the backend, frontend, and database structure. All cryptographic processes—salt generation, key derivation with Argon2id, and AES-GCM encryption—run in crypto.js on the client side. The main database tables are zero_auth (for hashed usernames, salts, and verifiers) and encrypted_vault (for encrypted data blobs and metadata hashes). To understand how ZKE works during registration, refer to Figure 4. All essential cryptographic steps take place locally in the user's browser. First, the client generates a unique salt. This salt acts as a random secret added to the master password, ensuring that even if two users have the same master password, their "fingerprints" (hashes or verifiers) are different. This makes rainbow table attacks much less effective.


Client-Side (Browser/React)
User Membuat Username dan Master Password
1. Menerima Input
2. Normalisasi & Hash Username - Menghasilkan HashedUsername
3. Generasi & Menghasilkan Salt
4. Derivasi Kunci Menggunakan Argon2id - Menghasilkan DerivedKey
5. Membuat Verifier Dengan SHA256 - Menghasilkan Verifier
6. Menyiapkan API Payload Dengan HashedUsername, Salt, Verifier
Server
7. Menerima POST /api/auth/setup Request Dengan Payload
8. Validasi Input Data
9. Instruksi Menyimpan Auth Data ke Database
Database
10. Menyimpan HashedUsername, Salt, Verifier di zero_auth table
11. Generasi Token JWT
12. Mengirim Respon Sukses dan Token JWT ke Client
Figure 4. Program Registration Flow

The client then uses Argon2id (with four iterations and 64 MB memory) to produce a derived key from the master password and salt. This process is intentionally resource-intensive, slowing down brute-force attempts. The derived key is used for cryptographic tasks, such as creating the verifier or encrypting and decrypting vault data, rather than using the master password directly. Usernames are normalized (converted to lowercase and trimmed) and hashed with SHA-256, creating a digital fingerprint that cannot be reversed. This protects the original username from exposure on the server. A verifier is created by hashing the derived key with SHA-256 and encoding it in Base64. Only the hashed username, salt (Base64), and verifier are sent to the /api/auth/setup endpoint. The server stores these values, never receiving the master password or derived key [14]. After registration, the login process (see Figure 5) also relies on client-side cryptography. The client sends a request to /api/auth/salt with the normalized and hashed username. The server retrieves the salt from the zero_auth table and returns it. The browser repeats key derivation using Argon2id with the entered master password and salt, producing the derived key. The client then creates a verifier (SHA-256 hash, Base64) and sends it to /api/auth/verify. The server checks this verifier against the stored value. If they match, the server issues a JWT token (HS256) to confirm successful authentication. Throughout this process, the master password and derived key remain in the browser; the server only sees the hashed username, salt, and verifier, and manages JWT validation and ciphertext storage.

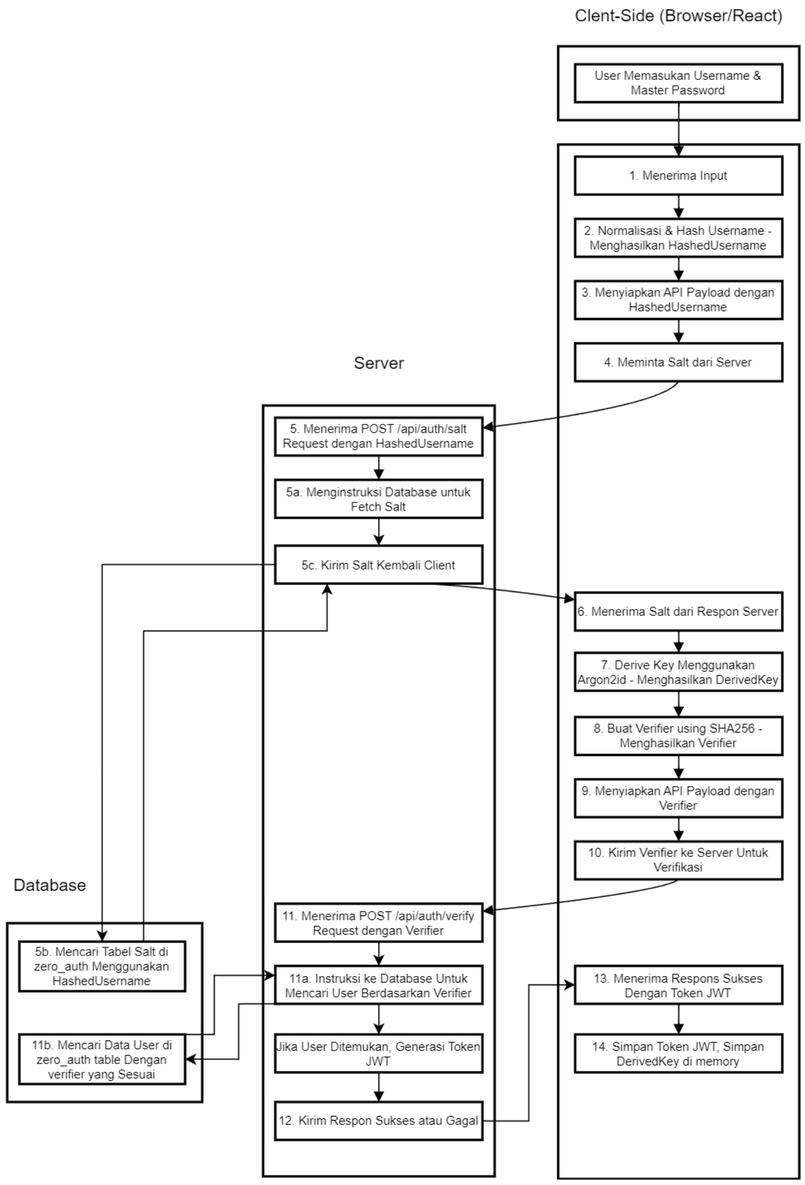

Clent-Side (Browser/React)
User Memasukan Username & Master Password
1. Menerima Input
2. Normalisasi & Hash Username - Menghasilkan HashedUsername
3. Menyiapkan API Payload dengan HashedUsername
4. Meminta Salt dari Server
Server
5. Menerima POST /api/auth/salt Request dengan HashedUsername
5a. Menginstruksi Database untuk Fetch Salt
5c. Kirim Salt Kembali Client
6. Menerima Salt dari Respon Server
7. Derive Key Menggunakan Argon2id - Menghasilkan DerivedKey
8. Buat Verifier using SHA256 - Menghasilkan Verifier
9. Menyiapkan API Payload dengan Verifier
10. Kirim Verifier ke Server Untuk Verifikasi
Database
11. Menerima POST /api/auth/verify Request dengan Verifier
5b. Mencari Tabel Salt di zero_auth Menggunakan HashedUsername
11a. Instruksi ke Database Untuk Mencari User Berdasarkan Verifier
13. Menerima Respons Sukses Dengan Token JWT
11b. Mencari Data User di zero_auth table Dengan verifier yang Sesuai
Jika User Ditemukan, Generasi Token JWT
14. Simpan Token JWT, Simpan DerivedKey di memory
12. Kirim Respon Sukses atau Gagal

Figure 5. Program Login Flow

Beyond registration and login, ZKE is used for credential management. All credential data (site name, username, password, and URL) is encrypted in the browser before being sent to the server. The encryption uses AES-256 GCM IV, chosen for its speed and high security [15]. The encrypted data is encoded in Base64 and sent to the database. The database stores the encrypted blob (in JSON with Base64 strings), together with the hashed site name and username. When users want to view a password, the application requests the encrypted package from the server using the hashed site name. The frontend decrypts it locally using the secret key derived from the master password. This key is kept in local memory during the session and never leaves the browser, ensuring only the user can read the password [16]. The AES-GCM encryption key is not stored directly; it is recreated from the master password when needed. The system's security relies on cryptographic operations performed on the client. Key derivation uses Argon2id with the following parameters:

Code Snippet 1. Argon2id Parameters
```
const params = {
  password: masterPassword,
  salt: saltArray,
  iterations: 4,
  memorySize: 65536, // 64 MB
  hashLength: 32,
  parallelism: 1,
  outputType: "binary",
};
```

These settings balance security and client-side performance. Four iterations provide enough security without slowing down the user experience. The memory size (64 MB) helps defend against GPU and ASIC

attacks. The hash length (32) ensures a 256-bit derived key. Setting parallelism to 1 keeps security consistent across devices, using only one thread. The salt is a random value added to the password before processing. This configuration follows OWASP recommendations for memory size above 48 MiB [17]. Argon2id was selected after testing its resistance to attacks such as rainbow tables, slow function overruns, GPU and FPGA/ASIC attacks, and implementation flaws. Argon2id performed better than other algorithms in these tests [18]. Credential and authentication data must be stored persistently. The application uses PostgreSQL. The zero_auth table stores the hashed username, salt, and verifier. The encrypted_vault table contains the item ID, hashed username, hashed site name, and the encrypted credential blob. Site names are also hashed with SHA-256 on the client before being stored or queried. To display readable site names in the UI, the app maintains a map that links plaintext site names to their hashed versions, updating it as users interact with the vault. CRUD operations for credentials are implemented as follows:
1) Create: Data is encrypted in the browser and sent (along with the hashed site name) to /api/vault/add.
2) Read: The client requests data from /api/vault/credentials using the hashed site name, receives the encrypted blob, and decrypts it locally.
3) Update: Similar to Create, but uses /api/vault/item (PUT) with the item ID.
4) Delete: Sends a request to /api/vault/item (DELETE) with the item ID and hashed site name.

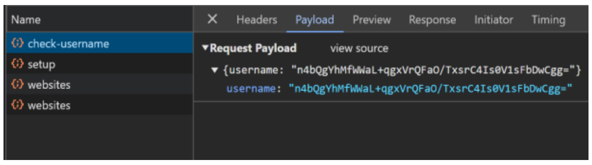
Throughout these operations, the server only stores data and cannot decrypt user information. The application also includes a secure random password generator, running entirely in the browser. In addition to external security evaluation and protocol validation, the application includes a mechanism to protect sensitive data in client memory during active sessions. The clearKeys() function wipes the master password and derived key from JavaScript memory when users log out or their session ends. The main logic is shown below:

Code Snippet 2. Memory Wipe Function
```
clearKeys() {
  if (this._keys?.rawKey instanceof Uint8Array) {
    this._keys.rawKey.fill(0);
  }
  if (this._masterPassword) {
    this._masterPassword = "0".repeat(this._masterPassword.length);
  }
  this._keys = null;
  this._masterPassword = null;
  this._initialized = false;
}
```

This function overwrites the raw key byte array and the master password variable with zeros, then sets their references to null so the browser's garbage collector can reclaim the memory. The goal is to minimize the window during which sensitive data remains accessible in client memory. It's important to distinguish between the mechanisms that define ZKE and standard security practices for password managers. The core ZKE features in this system include:
1) Key derivation exclusively on the client using Argon2id and the master password
2) End-to-end encryption and decryption of credential data using AES-GCM in the browser
3) Authentication using a cryptographic verifier, so the server never receives the master password or raw encryption key
4) Server-side storage limited to salt, verifier, and encrypted data blobs

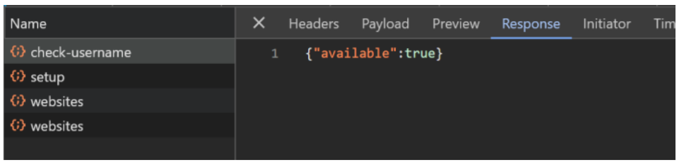
These ZKE-driven features are supported by other best practices, such as using industry-standard cryptography, hashing usernames and site names for extra privacy, session management via JWT, secure password generation, IP-based rate limiting, and input validation. Adopting ZKE in this password manager has both strengths and trade-offs. The main advantage is strong security and privacy: the master password never leaves the browser, so the server has no knowledge of the user's secret. Even if the database is breached, attackers only obtain salts, verifiers (hashes of derived keys), and encrypted data, but not the master password or decryption keys. The verifier allows authentication without exposing the encryption key. However, the most significant limitation is the lack of account recovery if users forget their master password. The application cannot provide a recovery mechanism, placing full responsibility on users to remember their master password and secure their devices, since all crucial cryptographic processes happen locally. This trade-off prioritizes server-side security and user privacy over account recovery convenience.

Testing was carried out on the application using browser developer tools. During these tests, various payloads were observed while performing login, registration, and several actions in the vault.



```
▾Request Payload
 ▾ {username: "n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg="}
    username: "n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg="
```

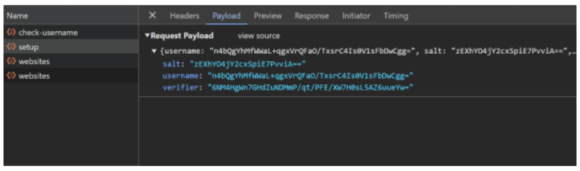
Figure 6. Payload Check Username



```
{"available":true}
```

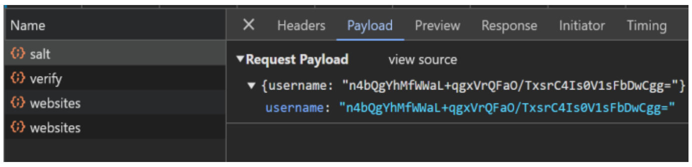
Figure 7. Response Check Username

Figure 6 shows the payload sent to the server during the initial login step, where the system checks whether a username is available. The transmitted data is already protected and not sent as plain text. In Figure 7, if the response is true, the username is still available and has not yet been registered in the database.



```
▾Request Payload
 ▾ {username: "n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg=", salt: "zEXhYO4jY2cxSpiE7PvviA==",…
    salt: "zEXhYO4jY2cxSpiE7PvviA=="
    username: "n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg="
    verifier: "6NM4HgWn7GHdZuNDMmP/qt/PFE/XW7H0sL5AZ6uueYw="
```

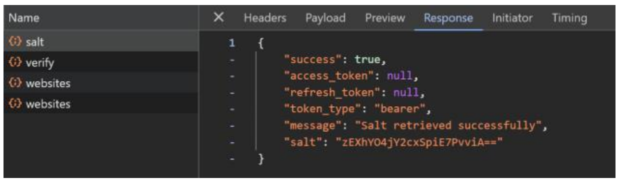
Figure 8. Payload Setup



```
▾Request Payload
 ▾ {username: "n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg="}
    username: "n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg="
```

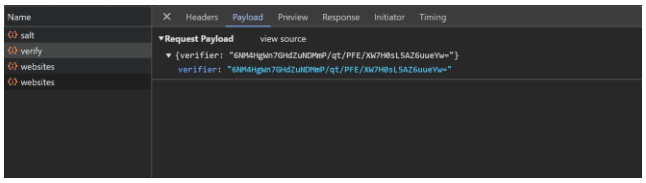
Figure 9. Payload Salt

Figure 8 displays the payload sent to the server after the user sets a master password and proceeds with registration. The username, salt, and verifier are all hashed before storage. Notably, the master password itself is never transmitted to the server. Figure 9 illustrates the login process. The client sends the hashed username to the salt endpoint, and the server's reply is shown in the next figure.



```
{
    "success": true,
    "access_token": null,
    "refresh_token": null,
    "token_type": "bearer",
    "message": "Salt retrieved successfully",
    "salt": "zEXhYO4jY2cxSpiE7PvviA=="
}
```

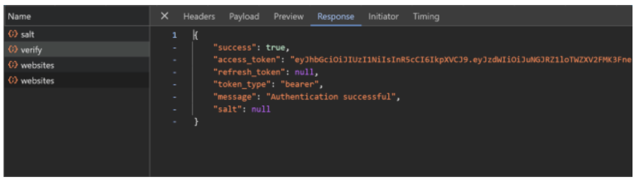
Figure 10. Response Salt



```
▾Request Payload
 ▾ {verifier: "6NM4HgWn7GHdZuNDMmP/qt/PFE/XW7H0sL5AZ6uueYw="}
    verifier: "6NM4HgWn7GHdZuNDMmP/qt/PFE/XW7H0sL5AZ6uueYw="
```

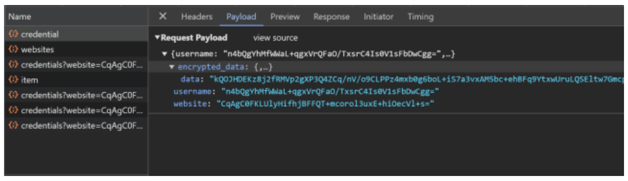
Figure 11. Payload Verify

Figure 10 confirms that the server returns the salt associated with the hashed username. Once the salt is received, the client needs to prove knowledge of the correct master password without actually sending it. To do this, the client computes a verifier using the master password and salt, then sends this verifier to the /verify endpoint. As shown in Figure 11, only the verifier is shared—neither the actual master password nor the secret encryption key ever leaves the client.



```
{
    "success": true,
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJuNGJRZ1loTWZXV2FMK3Fne
    "refresh_token": null,
    "token_type": "bearer",
    "message": "Authentication successful",
    "salt": null
}
```

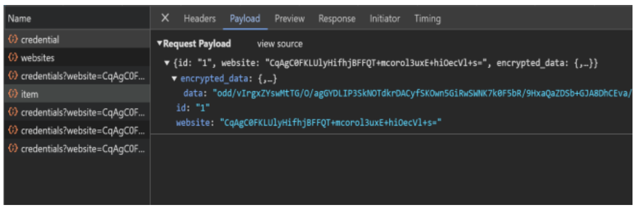
Figure 12. Response Verify


Name
credential
websites
credentials?website=CqAgC0F...
item
credentials?website=CqAgC0F...
credentials?website=CqAgC0F...
credentials?website=CqAgC0F...
Headers
Payload
Preview
Response
Initiator
Timing
view source

```
▾Request Payload
 ▾ {username: "n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg=",…}
  ▾ encrypted_data: {,…}
     data: "kQOJHDEKz8j2fRMVp2gXP3Q4ZCq/nV/o9CLPPz4mxb0g6boL+iS7a3vxAM5bc+ehBFq9YtxwUruLQSEltw7Gmc
    username: "n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg="
    website: "CqAgC0FKLUlyHifhjBFFQT+mcorol3uxE+hiOecVl+s="
```

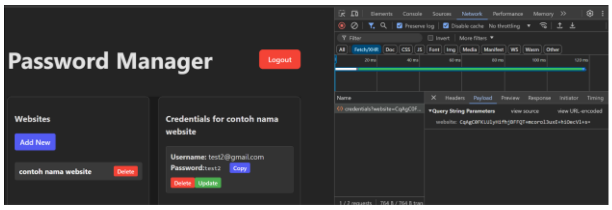
Figure 13. Credential Payload

The server checks whether the provided verifier matches the one stored during registration. If they match, as shown in Figure 12, the server approves the login (success: true), returns a session token (JWT), and grants access—all without ever knowing the master password. When users create credentials for a chosen website, the client sends a payload similar to that in Figure 13. All transmitted data is already hashed.


Name
credential
websites
credentials?website=CqAgC0F...
item
credentials?website=CqAgC0F...
credentials?website=CqAgC0F...
credentials?website=CqAgC0F...
Headers
Payload
Preview
Response
Initiator
Timing
view source

```
▾Request Payload
 ▾ {id: "1", website: "CqAgC0FKLUlyHifhjBFFQT+mcorol3uxE+hiOecVl+s=", encrypted_data: {,…}}
  ▾ encrypted_data: {,…}
     data: "odd/vIrgxZYswMtTG/O/agGYDLIP3SkNOTdkrDACyfSKOwn5GiRwSWNK7k0F5bR/9HxaQaZDSb+GJA8DhCEva/
    id: "1"
    website: "CqAgC0FKLUlyHifhjBFFQT+mcorol3uxE+hiOecVl+s="
```

Figure 14. Item Payload



Username: test2@gmail.com
Password: test2

```
▾Query String Parameters
   website: CqAgC0FKLUlyHifhjBFFQT+mcorol3uxE+hiOecVl+s=
```

Figure 15. Website Name Display

Figure 14 shows the browser log generated when a user updates a credential for a website (CqAgC...). The new data is also encrypted (odd/V...). Figure 15 demonstrates that decryption happens entirely on the client side. The server only provides the hashed version (CqAgC...), but the UI reveals the actual website name after decryption. Analysis of payloads using Chrome DevTools, especially during user interactions (registration, login, and credential management), consistently confirms that sensitive information—including master passwords, derived keys, and website credentials—is never sent to the server in plain text. After validating data integrity using Chrome DevTools, the web application's security was further assessed using OWASP ZAP version 2.16.1. OWASP ZAP was chosen based on comparative testing across 20 tools, where it ranked second but scored equally with the first and third positions. The selection was also influenced by familiarity with OWASP ZAP [19]. Automated scanning targeted the local frontend URL at http://localhost:5173. The primary goal was to evaluate server security settings and the application's HTTP responses. While automated scans with ZAP cannot cover every aspect of the app [20], they offer valuable insights into the overall security posture.

Table 4. ZAP Test Results

Type	Severity	Count
CSP Header Not Set	Medium	2
Hidden File Found	Medium	4
Missing Anti-clickjacking Header	Medium	2
X-Content-Type-Options Header Missing	Low	35
Information Disclosure in URL	Info	1
Information Disclosure - Comments	Info	6
Modern Web Application	Info	2
Tech Detected - Uvicorn	Info	1
Tech Detected - Vite	Info	1

The scan found no high-risk vulnerabilities. Several medium and low-risk issues were detected, which do not undermine the effectiveness of ZKE for server-side data protection, but should be addressed to further strengthen overall web application security.

4.2 Discussion

The primary findings at the 'Medium' risk level relate to the absence of recommended HTTP security headers, such as Content Security Policy (CSP) and anti-clickjacking headers, as well as the identification of hidden files. Without a CSP, the application is more susceptible to Cross-Site Scripting (XSS) attacks. The presence of four hidden files indicates that there are resources accessible to the public that should remain private. The missing anti-clickjacking header allows the application to be embedded within frames on external sites, creating an opportunity for clickjacking attacks, which can result in users unknowingly performing actions. At the 'Low' risk level, the lack of the X-Content-Type-Options header was observed. This omission permits MIME sniffing, where the browser attempts to determine the content type automatically rather than relying on the server's declaration. Such behavior may expose the application to security threats if non-executable content is interpreted as executable scripts, potentially enabling XSS attacks, though the risk remains minor. Additional findings categorized as 'Informational' include the detection of technologies in use (such as Uvicorn and Vite) and several minor disclosures. Overall, the results indicate the absence of critical vulnerabilities on the frontend layer. However, several aspects of standard web security can be improved, particularly by enforcing stricter server configurations and implementing appropriate security headers. These findings do not affect the assessment of the core security mechanism, Zero-Knowledge Encryption (ZKE). Subsequent testing used a blackbox approach to reflect user interactions and actions within the password manager application.

Table 5. Blackbox Testing Results

No	Function	Condition	Expected Outcome	Test Result	Conclusion
1	Registration	Username available	Redirect to master password setup form	Redirected to master password setup form	Match
		Username unavailable	Alert indicating username is taken	Alert indicating username is taken	Match
2	Login	Valid username and password	Access to vault	Access to vault	Match
		Invalid username or password	Alert indicating invalid credentials	Alert indicating invalid credentials	Match
3	Add website	Enter website name and credentials	Website data saved successfully	Website data saved successfully	Match
		Use password generation feature	Password generated	Password generated	Match

4	Update website data	Modify website credentials	Updated data saved successfully	Updated data saved successfully	Match
5	Copy website password	Copy password to Windows clipboard	Password copied	Password copied	Match
6	Delete data	Remove website entry	Website entry deleted	Website entry deleted	Match
		Remove website credentials	Website credentials deleted	Website credentials deleted	Match

Results from blackbox testing demonstrate that all essential features and actions performed as expected. The application's functionality aligns with its intended design and user requirements [21].

5. Conclusion

The project succeeded in developing and evaluating a web-based password manager utilizing Zero Knowledge Encryption (ZKE). Core cryptographic operations—including key derivation from the user's master password via Argon2id (with parameters balanced for client security and performance) and credential data encryption/decryption using AES-GCM—were executed entirely on the client side. The adopted client-server structure, employing React for the frontend and FastAPI for the backend, strictly separates sensitive data management on the client from encrypted data storage (salt, verifier, credential blob) on the server. As a result, the server does not access the master password or user credentials in plaintext form. Network payload analysis with Chrome DevTools confirmed that no sensitive information is transmitted without protection, validating the ZKE implementation. Additional measures, such as clearing cryptographic keys from client memory (clearKeys()), further reinforce security. A fundamental consequence of ZKE is the absence of account recovery options if the master password is forgotten. This trade-off is essential to maximize server-side security and user privacy. Security scanning with OWASP ZAP did not reveal critical vulnerabilities; however, several areas warrant improvement, such as the adoption of HTTP security headers (CSP, X-Content-Type-Options, anti-clickjacking), which would strengthen overall application resilience. The results demonstrate a practical and validated implementation of ZKE for a web-based password manager, offering a substantial increase in protection against server-side threats. Future research is encouraged to focus on designing and assessing account recovery mechanisms that align with ZKE principles, for example, approaches based on Shamir's Secret Sharing.

References

[1] Mannuela, I., Putri, J., & Anggreainy, M. S. (2021, October). Level of password vulnerability. In *2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI)* (Vol. 1, pp. 351-354). IEEE. https://doi.org/10.1109/ICCSAI53272.2021.9609778

[2] Sudiarto, W., Dhian, I., Ratri, E. K., & Susilo, H. (2017, April). Implementasi two factor authentication dan protokol zero knowledge proof pada sistem login. *JUTISI*, 3(1), 127–136. https://doi.org/10.28932/jutisi.v3i1.579

[3] Pearman, S., Zhang, S. A., Bauer, L., Christin, N., & Cranor, L. F. (2019). Why people (don't) use password managers effectively. In *Fifteenth symposium on usable privacy and security (SOUPS 2019)* (pp. 319-338).

[4] Alkaldi, N. A. (2019). *Adopting password manager applications among smartphone users* (PhD thesis, University of Glasgow). https://doi.org/10.5525/gla.thesis.74359

[5] Alodhyani, F., Theodorakopoulos, G., & Reinecke, P. (2020, November). Password managers—it's all about trust and transparency. *Future Internet, 12*(11), 1–50. https://doi.org/10.3390/fi12110189

[6] Oladipupo, R. O., & Olajide, A. O. (2019). An enhanced web security for cloud-based password management. [Online]. Available: www.aujst.com

[7] Aditama, W. Y., Hikmah, I. R., & Priambodo, D. F. (2023, August). Analisis komparatif keamanan aplikasi pengelola kata sandi berbayar Lastpass, 1Password, dan Keeper berdasarkan ISO/IEC 25010. *Jurnal Teknologi Informasi dan Ilmu Komputer, 10*(4), 857–864. https://doi.org/10.25126/jtiik.2023106544

[8] Oesch, S., & Ruoti, S. (2020, August). That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers. In *Proceedings of the 29th USENIX Conference on Security Symposium* (pp. 2165-2182).

[9] Pargaonkar, S. (2023, August). A comprehensive research analysis of software development life cycle (SDLC) agile & waterfall model advantages, disadvantages, and application suitability in software quality engineering. *International Journal of Scientific and Research Publications, 13*(8), 120–124. https://doi.org/10.29322/ijsrp.13.08.2023.p14015

[10] Chatzoglou, E., Kampourakis, V., Tsiatsikas, Z., Karopoulos, G., & Kambourakis, G. (2024, June). Keep your memory dump shut: Unveiling data leaks in password managers. In *IFIP International Conference on ICT Systems Security and Privacy Protection* (pp. 61-75). Cham: Springer Nature Switzerland. https://doi.org/10.48550/arXiv.2404.00423

[11] Khande, R., Ramaswami, S., Naidu, C., & Patel, N. (2021). An effective mechanism for securing and managing password using AES-256 encryption & PBKDF2. *Technology (IJEET)*, *12*(5), 1-7. https://doi.org/10.34218/ijeet.12.5.2021.001

[12] Garcia, S. P. L., Abraham, A. S., Kepic, K., & Cankaya, E. C. (2023). A Comparative Analysis of Web Application Vulnerability Tools. *Journal of Information Systems Applied Research, 16*(2). [Online]. Available: https://conisar.org

[13] Laipaka, R. (2022). Penerapan JWT untuk Authentication dan Authorization pada Laravel 9 menggunakan Thunder Client. In *Seminar Nasional Corisindo*.

[14] Chuah, C. W., Harun, N. Z., & Hamid, I. R. A. (2024). Key derivation function: key-hash based computational extractor and stream based pseudorandom expander. *PeerJ Computer Science*, *10*, e2249. https://doi.org/10.7717/peerj-cs.2249

[15] Susanti, A., Prasetiya, B. A., Pangesti, O. D., Suryawati, L. D., & Saputro, I. A. (2024, December). Perbandingan kinerja dan keamanan algoritma kriptografi modern AES-GCM dengan CHACHA20-POLY1305. *Infomatek, 26*(2), 253–264. https://doi.org/10.23969/infomatek.v26i2.19255

[16] R. S. (2020, October). Navigating client-side storage in modern web applications: Mechanisms, best practices, and future directions. *International Journal For Multidisciplinary Research, 2*(5). https://doi.org/10.36948/ijfmr.2020.v02i05.12096

[17] Tippe, P., & Berner, M. P. (2025, August). Evaluating Argon2 Adoption and Effectiveness in Real-World Software. In *International Conference on Availability, Reliability and Security* (pp. 25-46). Cham: Springer Nature Switzerland. https://doi.org/10.48550/arXiv.2504.17121

[18] Fedorchenko, V., Yeroshenko, O., Shmatko, O., Kolomiitsev, O., & Omarov, M. (2024, November). Password hashing methods and algorithms on the .NET platform. *Advanced Information Systems, 8*(4), 82–92. https://doi.org/10.20998/2522-9052.2024.4.11

[19] Belay, T. E., Gupta, S., & Burisa, E. (2025, April). Perform scanning and comparison of open source web application testing tools: Using strategic holistic approach. *Journal of Posthumanism, 5*(2), 1377–1402. https://doi.org/10.63332/joph.v5i2.512

[20] Maniraj, S. P., Ranganathan, C. S., & Sekar, S. (2024). Securing web applications with owasp zap for comprehensive security testing. *International Journal of Advances in Signal and Image Sciences*, *10*(2), 12-23. https://doi.org/10.29284/ijasis.10.2.2024.12-23

[21] Putri, M., Ginting, A., & Lubis, A. S. (2024). Pengujian aplikasi berbasis web data Ska menggunakan metode black box testing. *Februari, 2*(1), 41–48. https://doi.org/10.55537/cosmic.