Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Pentest-Report Obsidian Sync API, Server & Crypto 09.2024

Cure53, Dr.-Ing. M. Heiderich, M. Pedhapati, C. Luders, Dr. D. Bleichenbacher, Dr. N. Kobeissi

Index

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

“Obsidian is the private and flexible writing app that adapts to the way you think. Obsidian stores notes on your device, so you can access them quickly, even offline. No one else can read them, not even us. Obsidian uses open, non-proprietary files, so you're never locked in, and can preserve your data for the long term.”

From https://obsidian.md/

This report describes the results of a security assessment of the Obsidian Sync software, with an explicit focus on the server components and cryptographic implementations within Obsidian Sync. The project, which included a penetration test and a dedicated source code audit, was conducted by Cure53 in September 2024.

The audit, registered as *DYL-04*, was requested by Dynalist Inc. in August 2024. It should be clarified that this is not the first security-centered cooperation between Cure53 and Obsidian. In fact, the Sync component was a target to a parallel examination (*DYL-03),* with a different focus specifically angled towards security of the client component. Moreover, the Obsidian Sync was a target of a previous examination conducted by Cure53 in November 2023 and tracked as *DYL-01.*

In terms of the exact timeline and specific resources allocated to *DYL-04*, Cure53 has completed the research in CW38 In order to achieve the expected coverage for this task, a total of eight days were invested. In addition, it should be noted that a team consisting of five senior testers was formed and assigned to the preparation, execution, documentation, and delivery of this project.

Given the nature of the tasks envisioned for *DYL-04*, the assessment was split into two work packages (WPs):

- **WP1**: Crystal-box pentests & code audits against Obsidian Sync API & server
- **WP2:** Crystal-box pentests & cryptography reviews of Obsidian Sync crypto code

As the titles of the WPs indicate, the so-called crystal-box methodology was used for both WPs, with cryptographic reviewing methods featured in WP2. Cure53 was provided with invites to the Obsidian GitHub repository, links to the relevant documentation, as well as all further means of access required to complete the tests.

The project was completed without any major issues. To facilitate a smooth transition into the testing phase, all preparations were completed in CW37. Throughout the engagement, communications were conducted through a private, dedicated, and shared Discord channel. Stakeholders - including Cure53 testers and internal staff from Obsidian - were able to participate in discussions in this space.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cure53 did not need to ask many questions, and the quality of all project-related interactions was consistently excellent. The continuous exchange contributed positively to the overall results of this project. Significant roadblocks were avoided thanks to clear and careful preparation of the scope. Cure53 provided frequent status updates on the examination and emerging findings, but live reporting was not specifically requested for *DYL-04*.

The Cure53 team achieved very good coverage of the WP1 and WP2 objectives. Of the six security-related discoveries, five were classified as security vulnerabilities and one was classified as a general weakness with lower exploitation potential.

The backend functionalities offered by the Obsidian Sync software left Cure53 with a rather positive impression. Compared to the previous iteration, Obsidian has made significant strides in improving its cryptography.

Nevertheless, there are still some areas that require attention. Key rotation should be more flexible, password hashing should be strengthened, and additional hardening in the authentication is necessary. Sensitive data should be encrypted, and the *managed keys* mode should be improved to provide true end-to-end encryption.

Additionally, the absence of a technical specification for encryption mechanisms leveraged by Obsidian creates uncertainty for both security auditors and users. Hence, it is highly recommended to provide clear design documentation to make the overall processes and intentions more transparent.

The following sections first describe the scope and key test parameters, as well as how the work packages were structured and organized. Next, all findings are discussed in grouped vulnerability and miscellaneous categories. The vulnerabilities assigned to each group are then discussed chronologically. In addition to technical descriptions, PoC and mitigation advice is provided where applicable.

The report ends with general conclusions relevant to this September 2024 project. Based on the test team's observations and the evidence collected, Cure53 elaborates on the overall impressions and reiterates the verdict. The final section also includes tailored hardening recommendations for the Obsidian Sync software complex, more specifically the Obsidian Sync server components and cryptographic implementations.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Penetration tests & source code audits against Obsidian Sync software & new cryptography**
  - **WP1:** Crystal-box pentests & code audits against Obsidian Sync API & server
    - **Obsidian sources:**
      - **Branch:**
        - obsidian-master
      - **Commit ID:**
        - 220b580d4fd4ab13250703a3efd36310d1b7f0f2
    - **Obsidian static sources:**
      - **Branch:**
        - obsidian-static-master/
      - **Commit ID:**
        - fbf3d1ab4fb01047e36e0b571a5f020268137650
      - **Documentation:**
        - *Local_dev_environment_guide.md*
  - **WP2:** Crystal-box pentests & cryptography reviews of Obsidian Sync cryptographic code
    - Sources & documentation included in the repositories from WP1
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, every finding has been given a unique identifier (e.g., *DYL-04-001*) to facilitate any follow-up correspondence in the future.

DYL-04-001 WP1: Password hashing not memory-hard *(Low)*

Obsidian currently employs *bcrypt* for hashing user passwords on the server-side. While *bcrypt* has been a reliable choice for password hashing in the past, it pales in comparison to some of the more modern algorithms like *scrypt*. Limitations are particularly visible in resisting hardware-accelerated attacks.

Specific risks that might need to be addressed in relation to *bcrypt* over *scrypt* are discussed next.

- **Limited resistance to parallel attacks.** *bcrypt is* computationally intensive but not memory-hard. This means it primarily consumes CPU resources but minimal memory. Attackers using GPUs, FPGAs, or ASICs can exploit this by parallelizing computations, which would significantly speed up brute-force attacks.
- **Lack of memory hardness:** The absence of substantial memory requirements in *bcrypt* makes it more susceptible to attacks that leverage specialized hardware designed for high-throughput computations.

In contrast, *scrypt* is specifically designed to be both CPU-intensive and memory-hard[1]. To that end, it fosters:

- **Enhanced security through memory hardness.** *scrypt* requires a large amount of memory for its computations. This design choice makes it considerably more resistant to hardware-accelerated attacks because scaling up memory is more costly and less feasible than scaling CPU cores alone.
- **Reduced efficiency of specialized hardware attacks**: The memory-hard nature of *scrypt* limits the effectiveness of attackers using specialized hardware, as they cannot significantly speed up the hashing process without incurring substantial memory costs.

Furthermore, Obsidian already utilizes *scrypt* elsewhere in the codebase for the vault encryption's key generation process. Transitioning to *scrypt* for password hashing would not only enhance security but also maintain consistency within the codebase, potentially simplifying maintenance and reducing the learning curve for developers working with the cryptographic functions.

[1] https://eprint.iacr.org/2016/989

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

It is recommended to replace *bcrypt* with *scrypt* as a way of hashing user passwords on the server-side. When implementing *scrypt*, appropriate parameters (*N, r, p*) need to be carefully chosen to balance security and performance. This should be based on current best practices and the anticipated server load:

- **N (CPU/memory cost parameter):** a power of two greater than 2^{14} (e.g., 2^{16} or 2^{18}), depending on acceptable performance.
- **r (block size parameter):** typically 8.
- **p (parallelization parameter):** typically 1, unless multi-threaded processing is desired.

By adopting *scrypt* for password hashing, Obsidian will significantly enhance its defense against password cracking attempts, especially those utilizing recent specialized hardware.

DYL-04-002 WP1: Internal comms method relies on single hard-coded secret *(Low)*

***Fix note**: The issue was fixed by the Obsidian team after the audit, and the patch was successfully verified by the Cure53 team.*

The current implementation of a specific type of internal authentication tokens between Obsidian servers relies on a single hard-coded secret, *ACCOUNT_SERVER_PASSWORD*, which is embedded directly in the source code. This could lead to a number of different problems in at least three areas of hard-coding, weak generation mechanism and suboptimal management of secrets.

In terms of hard-coding, exposure risk stems from storing secrets directly in the source code. This approach increases the risk of unauthorized access, especially if the codebase is shared, leaked, or improperly secured. Moreover, it has direct effects on inflexibility, since hard-coded secrets cannot be easily rotated or changed without modifying the source code and redeploying the application. This complicates secrets' management, as also discussed below.

Regarding the weak token generation mechanism, the problems relate to predictability, replay attacks and lack of entropy. The authentication token is generated using a SHA-256 hash of the hard-coded secret concatenated with a timestamp. An attacker who knows or can guess the timestamp (which is often predictable) could generate valid tokens if they obtain the secret. Since the token is valid within a ±30-minute window, intercepted tokens could be reused by an attacker within this timeframe. This signifies easier replay attacks. Moreover, the token generation does not utilize sufficient randomness or entropy, making it more susceptible to brute-force or guessing attacks.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Finally, Cure53 would argue that management of secrets is inadequate, especially since it has a single point of failure. The secret is not stored in an environment variable or a secure secrets management system, violating best practices for sensitive data handling. Relying on a single secret for all token generation and validation immediately creates a critical vulnerability if that secret is compromised.

Affected files:

- *web/main/core/validation.ts*
- *web/sync/server.ts*

Affected code:

```
const ACCOUNT_SERVER_PASSWORD = '[REDACTED]';

export async function isValidAuthToken(token: string): Promise<boolean> {
   const parts = token.split(':');
   if (parts.length !== 2) {
       return false;
   }
   const [digest, timestamp] = parts;
   // Timestamp must be within ±30 minutes of now
   if (Math.abs(Date.now() - parseInt(timestamp)) > 1800000) {
       return false;
   }
   const expectedDigest = crypto.createHash('sha256')
       .update(ACCOUNT_SERVER_PASSWORD + timestamp)
       .digest()
       .toString('hex')
       .toLowerCase();
```

This issue can be addressed by implementing recommendations corresponding to the discussed problems.

Firstly, hard-coded should be removed. It is recommended to:

- **Use environment variables:** Store *ACCOUNT_SERVER_PASSWORD* in an environment variable or a dedicated secrets management service (e.g., AWS Secrets Manager, HashiCorp vault).
- **Secure access control:** Ensure that only authorized personnel and processes can access the secret; implement strict access controls and auditing.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Moreover, generation of *auth* tokens should be improved by using cryptographically secure random tokens. Obsidian should generate tokens using a cryptographically secure pseudorandom number generator (CSPRNG), such as *crypto.randomBytes* in Node.js. Tokens need to also be stored securely on the server-side. By this it is also meant that they must be associated with the user session and metadata, like creation time and expiration.

Cure53 recommends adoption of standard libraries in this context. Established authentication frameworks or libraries should be deployed to handle token generation, storage, and validation securely (e.g., JSON Web Tokens (JWT) with proper signing and verification).

There are also certain ways to enhance token validation. Cure53 proposes to:

- Shorten token lifespan by reducing the validity window of tokens to the minimum necessary duration to limit exposure in case of interception.
- Implement refresh mechanisms and token rotation strategies to enhance security.
- Include additional claims. If JWTs are used, include claims like *issuer (iss), subject (sub)* and *audience (aud)* to prevent token misuse.

A revised approach would offer enhanced security through less predictable tokens, proper secret management and compliance with best practices.

DYL-04-003 WP1: Hard-coded secrets give access to development tools *(Low)*

Obsidian stores sensitive private secrets in plaintext within a single configuration file on the server-side (*web/main/config.ts*). This file contains various secrets, including API keys, private keys, and tokens, which are hard-coded directly into the source code. The Obsidian team has clarified that these secrets are used during development and not in the final product. Nevertheless, hardcoding these secrets can result in the following security risks:

Exposing secrets:

- **Source code leakage:** If the source code is ever leaked, shared, or improperly secured, all embedded secrets become immediately compromised.
- **Version control vulnerabilities:** Storing secrets in files tracked by version control systems (e.g., Git) increases the risk of accidental exposure through code commits, pushes to public repositories, or forks.
- **Single point of failure:** Keeping all secrets in one file creates a single point of failure, increasing the impact if that file is accessed by unauthorized individuals.
- **Violation of security guidelines:** Keeping all secrets in one file creates a single point of failure, increasing the impact if that file is accessed by unauthorized individuals.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Inflexible rotation of secrets:

- **Difficulty in rotating secrets:** Changing secrets requires modifying the source code and redeploying the application, which is cumbersome and prone to errors.
- **Delayed response to compromise:** In the event of a suspected breach, rotating secrets quickly is critical. Hard-coded secrets hinder rapid response.

Affected file:
obsidian/web/main/config.ts

Affected code:
```
let config = {
	env: 'dev',
	// Cookie secret used to sign cookies
	[REDACTED]
};
```

This issue can be addressed by adopting recommendations pertinent to managing secrets, especially rotation and monitoring.

In the context of improving secrets' management, environment variables should be used to secure injection. Specifically, secrets should be loaded from environment variables set in the deployment environment. Configuration should also be separated from the code. Cure53 recommends removal of hard-coded secrets from the codebase to prevent accidental exposure.

In addition, dedicated services for management of secrets should be utilized. In this realm, dedicated tools like HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, or Google Secret Manager can be relied upon. Attention is also needed regarding access control and auditing, as these provide fine-grained control, reviewing capabilities, and secure storage mechanisms.

Obsidian is advised to implement secret rotation policies:

- **Regular rotation schedule:** Establish a schedule for regular secret rotation to minimize the window of opportunity for compromised secrets.
- **Immediate rotation on compromise:** Have procedures in place to rotate secrets immediately if a compromise is suspected.

Last but not least, security monitoring should be enhanced by:

- **Access logging:** Monitor access to secret storage locations to detect unauthorized access attempts.
- **Alert mechanisms:** Set up alerts for unusual activities related to secret access or changes.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

DYL-04-005 WP1: Key mgmt. confusion in managed vault encryption mode (*Medium*)

Note: After sufficient discussion with the client, the Obsidian team promptly updated the UI/UX to clearly outline how managed and end-to-end encryption modes work, ensuring there is no room for confusion.

It was noticed that Obsidian offers two vault encryption modes:

- **End-to-end encryption (default)**: Users locally generate encryption keys and never share them with servers.
- **Managed encryption (opt-in)**: Obsidian servers generate, manage, and store encryption keys for clients.

While the first mode provides true end-to-end encryption, the "*managed*" keys mode is fundamentally insecure and, by only storing the encryption keys on a different server instead of using a dedicated hardware-based security solution, fails to provide meaningful encryption at rest. In this mode, the Obsidian service generates and holds the encryption keys on a separate machine from the one holding the vault ciphertext, nevertheless still having full access to the encrypted vault-contents at any time. This undermines the primary purpose of encryption, which is to protect data from unauthorized access, including from the server itself.

Furthermore, in all of the available front-facing Obsidian documentation and marketing materials that could be found at the time of audit, "managed encryption" and "end-to-end encryption" modes might look like they are offering similar security guarantees.[2][3] This further increases the confusion between both encryption modes, which in reality offer vastly different security guarantees: end-to-end encryption modes provide true resilience against a dishonest server, whereas managed encryption is not secure even in the face of an "honest-but-curious" server and is only secure in highly restricted contexts, such as partial server compromise or a partial compromise at the level of the transport layer (eg. TLS certificate compromise).

Specific problems are listed next.

- **Server access to encryption keys:**
  - No true encryption at rest: Since the server holds both the encrypted data and the encryption keys, it can decrypt the vault contents at will.
  - Increased attack surface: If the server is compromised, attackers can obtain both the encrypted data and the keys, rendering encryption ineffective.

[2] https://help.obsidian.md/Obsidian+Sync/Security+and+privacy
[3] https://obsidian.md/blog/verify-obsidian-sync-encryption/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

- **Lack of end-to-end encryption:**
  - Compromised user-privacy: Users may believe their data is securely encrypted and inaccessible to the server but, in reality, the server can access all plaintext data.
  - Misleading security claims: Advertising this mode as secure encryption may mislead users into a false sense of security.
- **No hardware security measures:**
  - Absence of HSMs or TEEs: The server does not utilize Hardware Security Modules (HSMs) or Trusted Execution Environments (TEEs) to securely manage keys without exposing them to the server's operating system or administrators.
  - Vulnerability to insider threats: Server administrators or malicious insiders could access the keys and decrypt user data.

- **Insufficient separation of servers:**
  - *Sync* server vs. *account* server: Although vault contents are sent only to the *sync* server and not the *account* server, this separation does not mitigate the fundamental issue because Obsidian, the company, still has access to both the encrypted data and the keys without any of the aforementioned hardware-based security measures being implemented.
  - No security guarantees: Physical or logical separation of servers without proper key management provides negligible security benefits.

With the current *managed keys* design, users' sensitive data is at risk of unauthorized access by the server or attackers who find a way to compromise the Obsidian servers.

The following recommendations could help address these issues:

- **Eliminate *managed keys* mode:**
  - Deprecate *managed* keys mode: Consider phasing out this mode entirely due to its inherent security weaknesses.
- **If the *managed keys* mode remains, enhance key management:**
  - Implement Hardware Security Modules (HSMs):
    - Use HSMs to generate, store, and manage encryption keys securely, ensuring keys are not exposed to the server's software environment.
  - Utilize Trusted Execution Environments (TEEs):
    - Employ TEEs to perform encryption and decryption operations without exposing keys or plaintext data to the rest of the system.
  - Strict access controls and auditing:
    - Enforce least-privilege access policies for administrators.
    - Implement comprehensive logging and monitoring of all key-management activities.
    - Regularly audit access to encryption keys and vault contents.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

- **Improve server architecture:**
  - Separate key management services: Isolate key management functions from data storage services to minimize the risk of key compromise.
  - Encrypted key storage: Encrypt keys at rest using a master key stored in secure hardware.
- **Transparent communication with users:**
  - Update documentation and marketing materials: Clearly explain the security implications of each encryption mode.
  - Inform users about the risks: Ensure users are aware that the *managed* keys mode does not provide the same level of security as end-to-end encryption.

DYL-04-006 WP1: Vault auth reveals information about encryption key *(Medium)*

***Fix note**: The issue was fixed by the Obsidian team after the audit, and the patch was successfully verified by the Cure53 team.*

Some observations were made about the current key derivation method for vault encryption in Obsidian, particularly in relation to using true end-to-end encryption where the user chooses an encryption password that is not managed by the Obsidian service. This, in fact, reveals information about the encryption key to the server. The existing implementation derives the encryption key directly from the user's password using *scrypt*, and then generates an authentication key by hashing the encryption key with SHA-256.

This process inadvertently exposes information about the encryption key to the server during authentication, since the hashed encryption key is shared with the server for the purpose of authentication. Best practices dictate that servers should not have any knowledge of encryption keys or passwords, as only this approach can maintain end-to-end encryption integrity. Furthermore, using the same key (or derived keys without proper key separation) for both encryption and authentication violates cryptographic best practices.

To enhance security and ensure that the server does not gain any information about the user's encryption key or password, it is recommended to revise the key derivation and authentication mechanisms in accordance with the recommendations presented next.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

- **Implement key separation using HKDF:**
  - Intermediate key derivation. This means using *scrypt* to derive an intermediate key from the user's password (instead of directly deriving the vault encryption key, as is done currently). This intermediate key serves as a high-entropy source for further key derivation.
  - Use HKDF for key separation. Specifically, apply HKDF-SHA-256 in order to derive separate keys for encryption and authentication from the intermediate key. The encryption key is then used to encrypt the vault with AES-SIV, while the separate authentication key is used to authenticate with OPAQUE, as detailed below. This offers two benefits. The first entails indistinguishability, which ensures that the derived keys are cryptographically independent. The second concerns security, meaning that compromise of one key would no longer affect the security of the other.
- **eUse OPAQUE for authentication:**
  - Implement the OPAQUE password-authenticated key exchange:
    - Utilize the OPAQUE protocol, a modern PAKE that allows secure authentication without revealing passwords or derived keys to the server. The client and server engage in the OPAQUE protocol using the authentication key derived via HKDF as the client's OPAQUE *"password".* Upon successful authentication, the server grants access without ever receiving the authentication key, and therefore learning no information about the encryption key.
    - This approach has benefits in the realm of server blindness (server remains oblivious to any encryption keys or passwords) and key independence (separate, blinded keys for encryption and authentication prevent cross-compromise or information leakage).
  - Server-side adjustments:
    - Eliminate KeyHash storage by removing storage of any hashes or representations that could expose key material, as well as by storing only the necessary, blinded values required by the OPAQUE protocol.
    - Session management: Upon successful authentication via OPAQUE, establish a secure session using standard practices.

Implementing the recommended solutions will significantly enhance security by ensuring that the server has no access to any key material, thereby greatly improving the confidentiality and integrity of user data. This approach reduces the risk of data breaches by limiting the potential damage from server-side compromises. The new design showcases that attackers cannot obtain passwords or keys from compromised servers. Moreover, it demonstrates a strong commitment to user privacy and data protection, which will enhance users' trust in Obsidian.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible.

DYL-04-004 False Positive: Ineffective string validation function *(Info)*

Note: After discussion with the Obsidian team, the issue can be seen as a false positive finding. The utility function is introduced to facilitate dynamic type checking of strings, which the Obsidian team clarified that TypeScript cannot handle on its own.

Obsidian utilizes a *validateString* function to perform runtime type checking of inputs to ensure they are strings. While runtime validations can be beneficial in certain contexts, in this case, the use of *validateString* is likely redundant due to TypeScript's robust compile-time type checking. Relying on both TypeScript's type system and additional runtime checks for string validation leads to unnecessary code complexity and potential maintenance challenges.

Affected file:
obsidian/web/main/core/validation.ts

Affected code:
```
export function validateString(input: any, key: string): void {
	if (typeof input !== 'string') {
		throw new HandleError(`Invalid Request: "${key}" must be a
string.`);
	}
}
```

To optimize the validation process, Obsidian should leverage TypeScript's compile-time type-checking by removing redundant *validateString* calls and ensuring comprehensive type definitions throughout the codebase. It is recommended to implement runtime validations selectively, retaining them only for external inputs and APIs where TypeScript's type guarantees do not apply, and utilize TypeScript's type guards and assertions to maintain type safety without unnecessary functions.

Moreover, it would be beneficial to enhance error handling by centralizing validation logic. The goals should be to reduce duplication and adopt established validation libraries like Joi or Yup for more maintainable solutions. Additionally, developer awareness and adherence to best practices can be fostered through rigorous code reviews, clear development guidelines, and comprehensive training and documentation that emphasize the effective use of TypeScript's features, thereby minimizing reliance on redundant runtime validations.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

As noted in the *Introduction,* several positive changes have been observed between the previous and current assessment of the Obsidian Sync server and cryptography. On the one hand, Cure53 wishes to praise the Obsidian team on bringing the security posture to a better composition overall. On the other hand, certain aspects still need attention, as evidenced by the presence of six security problems on the list of findings from this September 2024 inspection.

At the center of *DYL-04* is one of the main changes that Obsidian made in their code, namely the addition of a new ciphersuite for the vaults. Each vault uses one ciphersuite that is defined during the vault's construction. This translates to a clean design that allows to deprecate cryptographic primitives in a transparent manner by migrating vaults.

Cure53 has reviewed the new ciphersuite using AES-GCM and AES-SIV as underlying encryption modes. No flaws were observed in the design and usage of this ciphersuite. Alterations therefore improve upon the previous version and remove the weaknesses found there.

An important feature of storage encryption is the ability to rotate keys. The goal is to limit the period during which each encryption key can be used. The current implementation allows key rotation through vault migration. However, such a rotation may be somewhat inflexible. Cure53 recommends considering a feature that would allow periodic key rotation.

DYL-04-001 shows that Obsidian currently utilizes *bcrypt* for hashing user passwords on the server-side. While historically this was reliable, it lacks the memory-hard properties necessary to effectively resist modern hardware-accelerated brute-force attacks. *Bcrypt* primarily consumes CPU resources without significant memory usage, making it vulnerable to parallelization attacks using GPUs, FPGAs, or ASICs. In contrast, *scrypt* is designed to be both CPU and memory-intensive, substantially enhancing resistance against such attacks.

In another tickets, DYL-04-002 and DYL-04-003 highlight a security vulnerability in Obsidian’s source code where secrets and tokens were hard-coded within the code. This practice poses multiple risks, including increased exposure to unauthorized access if the source code is leaked or improperly secured, and inflexibility in rotating secrets without modifying and redeploying the application. However, it was later confirmed with the Obsidian team that the secrets are non-critical.

Furthermore, the token generation method in DLY-04-002, which uses a SHA-256 hash of the hard-coded secret concatenated with a timestamp, is predictable and lacks sufficient entropy, making it susceptible to brute-force and replay attacks within the token’s validity window of ±30 minutes.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

A key management deficiency is discussed in DYL-04-005. Obsidian's *managed keys* vault encryption mode in the Obsidian documentation and marketing materials that could be found at the time of audit. In these, the described "managed encryption" and "end-to-end encryption" modes might look as if they are offering similar security guarantees. It was recommended to modify the documentation to imply clearly.

The managed encryption fails to provide true end-to-end encryption, as the server itself can access and decrypt user data, negating the primary purpose of encryption to protect data from unauthorized access, including presumed maliciousness of the server. The absence of hardware security measures such as Hardware Security Modules (HSMs) or Trusted Execution Environments (TEEs) exacerbates this vulnerability, making the system susceptible to insider threats and server compromises.

DYL-04-006 addresses issues in Obsidian's vault encryption key derivation process, where the current method inadvertently exposes information about the encryption key to the server during authentication. The existing implementation derives the encryption key directly from the user's password using *scrypt* and then generates an authentication key by hashing the encryption key with SHA-256. This results in the server receiving a hashed version of the encryption key (KeyHash).

It is recommended to implement key separation using HKDF to derive distinct encryption and authentication keys from an intermediate key. Moreover, Cure53 proposes adoption of the OPAQUE Password-Authenticated Key Exchange protocol to enable secure authentication without revealing information about the encryption key or password to the server.

Overall, Obsidian's cryptography, while acceptable, could be improved, especially by addressing current shortcomings in the Obsidian vaults with *managed keys* in the documentation, as discussed in DYL-04-005. Furthermore, the lack of any technical specification for Obsidian's encryption mechanisms is not ideal, since it results in subpar clarity for both security auditors and Obsidian users.

Cure53 would like to thank Erica Xu, Steph Ango, Shida Li, and Tony Grosinger from the Dynalist Inc. team for their excellent project coordination, support and assistance, both before and during this assignment.