Obsidian Sync

Security Assessment

December 15, 2025

Prepared for:
Steph Ango
Obsidian

Prepared by: **Joe Doyle and Marc Ilunga**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Kimberly Espinoza, Project Manager
kimberly.espinoza@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Marc Ilunga, Consultant
marc.ilung@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 30, 2025	Delivery of draft comprehensive report
October 20, 2025	Delivery of final comprehensive report
November 21, 2025	Delivery of final comprehensive report with fix review
December 4, 2025	Delivery of updated final comprehensive report with fix review
December 15, 2025	Delivery of updated final comprehensive report with fix review

Executive Summary

Engagement Overview

Obsidian engaged Trail of Bits to review the security of Obsidian Sync, a service that enables end-to-end encrypted backup of users' notes on Obsidian servers and backups with encryption keys managed by Obsidian.

A team of two consultants conducted the review from September 22 to September 26, 2025, for a total of two engineer-weeks of effort. Our testing efforts focused on identifying severe issues in the security of the end-to-end encrypted and managed sync functionalities against several attacker models. We also focused on assessing the strength of cryptographic primitives and the protection of metadata. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The Obsidian Sync service aims to provide end-to-end encryption and managed backups. The system uses a bespoke backup protocol because there are currently no standardized protocols for encrypted backups. However, we have identified several serious issues in the Sync protocol and implementation that undermine Obsidian Sync's security guarantees.

We identified a high-severity issue due to the use of weak randomness, which could allow an attacker to recover encrypted files (TOB-OBSYNC-1).

We also identified several authentication-related issues in the overall Obsidian system, which can threaten user accounts and also the security of data stored in Sync. We found insecure comparison of attacker-controlled data against secrets (TOB-OBSYNC-3); insecure storage of sensitive data, including passwords and TOTP codes (TOB-OBSYNC-4); and actions that can be performed without proper authentication (TOB-OBSYNC-2, TOB-OBSYNC-6). We have also identified instances of secrets stored in the codebase (TOB-OBSYNC-7, TOB-OBSYNC-8).

At the protocol level, we found that the Sync protocol does not sufficiently achieve the expected security guarantees of end-to-end encrypted protocols. In particular, it leaks information about encrypted data (TOB-OBSYNC-9) and does not appropriately protect metadata nor provide sufficient integrity guarantees (TOB-OBSYNC-10).

We identified several recommendations for hardening the cryptographic design of the system, which can be found in appendix D.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Obsidian take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts.
- **Deprecate and require migration of all weak or potentially exposed keys and passwords.** Some vault passwords and all vault seeds generated by this implementation are insecure and should be replaced. In addition, current multifactor authentication (MFA) secrets and MFA recovery codes are potentially exposed and should be rotated once a more secure storage solution is implemented.
- **Consider implementing in-depth mitigations.** We provide several recommendations for strengthening the Obsidian Sync system in appendix D. These recommendations aim to strengthen the security of Obsidian Sync against various attackers.
- **Formalize the security guarantees of Obsydian Sync.** A recent study on the security of end-to-end encrypted backup systems revealed that they are not trivial to build and require more than just simple encryption. Therefore, a strong formalization of the expected security guarantees is a prerequisite to ensure that the implemented protocol achieves meaningful security guarantees.
- **Perform a comprehensive application security review.** While we focused this review on the cryptographic components of the Sync system, we found several serious application security flaws. The Obsidian team should seek out a comprehensive application security evaluation of their system, since we believe more issues may be present.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	5
Medium	3
Low	2
Informational	1
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Authentication	3
Cryptography	4
Data Exposure	3

Project Goals

The engagement was scoped to provide a security assessment of the Obsidian Sync. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are end-to-end encrypted vaults securely encrypted?
- Are end-to-end encrypted secure against malicious servers, networking adversaries, and adversaries that break into the server?
- Does the Obsidian Sync system use security cryptographic primitives?
- Are cryptographic secrets stored securely on the server?
- Is access to the vault protected by appropriate authentication?
- Is metadata sufficiently protected?

Project Targets

The engagement involved reviewing and testing the following target.

Obsidian Sync

Repository	obsidian
Version	`a675feb5da040d3a444db82ef98309e493ae4a5d`
Type	TypeScript
Platform	Multiple

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Static analysis:** We ran Semgrep on the codebase and reviewed the results.
- **Manual review:** We manually reviewed the following components of the system:
    - **Sync client**: We reviewed the main components of the client-side sync, including file uploads and downloads, and the primitives for file encryption and authentication. We checked that the file contents are secure against various threat models, including malicious servers, and that metadata is appropriately protected.
    - **Sync server:** We reviewed the server components relevant to the sync functionality. We checked that cryptographic secrets are stored securely, and we checked for the use of strong cryptographic primitives. We assessed whether the server authenticates access to vaults and whether managed vaults are secure against break-in adversaries.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Application security of Obsidian Sync.** We focused on evaluating the cryptography and key management used in Obsidian Sync. While we did evaluate several other parts of the Sync system in the process, we focused primarily on identifying the most severe issues and providing effective recommendations.
- **Correctness of the synchronization algorithm**. Obsidian Sync's synchronization algorithm represents a fairly complex state machine whose correctness is not trivial to establish. During the audit, we performed a high-level manual review, but we did not comprehensively evaluate it, especially in the case of a malicious server.
- **Security of the overall Obsidian system.** Our efforts were focused on the Obsidian Sync functionality. While we incidentally discovered several serious security flaws with the Obsidian authentication system, we did not comprehensively assess any other component; consequently, we did not review potentially dangerous interactions between other components and the sync functionality.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix C

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- General code quality issues and unidiomatic code patterns

Test Results

The results of this focused testing are detailed below.

Semgrep

We ran Semgrep Pro on the codebase using both the default configuration and our internal rulesets. This analysis did not identify any security issues in the codebase.

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Use of Math.random for salt and password generation	Cryptography	High
2	Logged-out clients can look up and trigger deletion of vaults with inactive subscriptions	Access Controls	Informational
3	Variable-time comparison of secrets	Data Exposure	High
4	Secrets are stored in plaintext	Data Exposure	High
5	TOTP codes can be used multiple times	Authentication	High
6	Password reset does not require MFA authentication	Authentication	Medium
7	Fixed-password authentication allows unauthorized use of /size endpoint	Authentication	Low
8	Repository contains hard-coded credentials	Data Exposure	Medium
9	Deterministic encryption of file hash endangers file confidentiality	Cryptography	Medium
10	General lack of cryptographic binding between file content and metadata	Cryptography	High
11	The authentication protocol does not guarantee proof of possession of the vault key	Cryptography	Low

Detailed Findings

1. Use of Math.random for salt and password generation	
Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-OBSYNC-1
Target: `web/main/routes/vault.ts`, `src/app/plugins/sync/sync-modal.ts`	

Description

The encryption and authentication keys used for vault sync are computed from a password and a salt. In "standard" mode, the server generates both during vault creation and stores them, as shown in figure 1.1:

```
[REDACTED]
```

Figure 1.1: Non-E2EE vault passwords and salts are generated via `generatePassword` (`obsidian/web/main/routes/vault.ts#151–152`)

The `generatePassword(20)` call generates a 20-character password using the `Math.random` function, as shown below in figure 1.2.

```
[REDACTED]
```

Figure 1.2: `generatePassword` uses `Math.random` to select characters (`obsidian/web/main/util.ts#52–62`)

In E2EE mode, the salt sent to the server is generated using a modified version of the `generatePassword` function, shown below in figure 1.3.

```
[REDACTED]
```

Figure 1.3: The client-side equivalent, used for salt generation only (`obsidian/src/shared/util/data.ts#101–109`)

Using `Math.random` to generate passwords and salts is insecure because it has low seed entropy and is vulnerable to state recovery attacks. Node and Electron are both based on the V8 JavaScript engine, which uses the XorShift128 RNG that is initialized based on a 64 bit seed, as shown below in figure 1.4.

```
int RandomNumberGenerator::Next(int bits) {
  DCHECK_LT(0, bits);
```

```
  DCHECK_GE(32, bits);
  XorShift128(&state0_, &state1_);
  return static_cast<int>((state0_ + state1_) >> (64 - bits));
}


void RandomNumberGenerator::SetSeed(int64_t seed) {
  initial_seed_ = seed;
  state0_ = MurmurHash3(base::bit_cast<uint64_t>(seed));
  state1_ = MurmurHash3(~state0_);
  CHECK(state0_ != 0 || state1_ != 0);
}
```

Figure 1.4: Random number generation and seeding in V8's `Math.random()` (v8/src/base/utils/random-number-generator.cc#212–225)

An attacker who observes a generated salt can take advantage of this PRNG's linearity to compute the internal state, and from there learn passwords and salts generated by that system nearby in time. There are existing tools to calculate the internal state, such as the scripts in "Hacking the JavaScript Lottery."

Even without direct access to an output of the PRNG, the 64-bit seed size is small enough that a well-resourced attacker who receives an encrypted vault can practically brute-force "standard-encryption" vaults and E2EE vaults with weak passwords. The small seed space for the salt also allows an attacker to use precomputation to build a rainbow table for weak passwords, significantly reducing the value of using a salt. Since keys are derived via `scrypt`, this attack is more difficult than a straightforward 64-bit hash preimage attack, but it is still tractable.

Exploit Scenario

Alice compromises a sync server and retrieves the encrypted vaults stored on it. She brute-forces the seed used to generate the password and salt used for each non-E2EE vault and decrypts the vaults without authorization.

Recommendations

Short term, replace the use of `Math.random` with a secure PRNG such as `crypto.getRandomValues`, and deprecate passwords and salts generated with `Math.random`.

Long term, ensure that cryptographic keys are always generated from secure sources of randomness.

References

- Hacking the JavaScript Lottery

2. Logged-out clients can look up and trigger deletion of vaults with inactive subscriptions

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Access Controls	Finding ID: TOB-OBSYNC-2
Target: `web/main/routes/vault.ts`	

Description

Using Obsidian Sync requires a subscription, and 30 days after a vault owner's subscription expires, the server-side copy of the synchronized vault can be deleted. However, this is triggered by the `/access` endpoint, which looks up the vault and can trigger deletion without requiring the client to be logged in, as shown below in figure 2.1.

```
[REDACTED]
```

Figure 2.1: The user's login status is checked after the vault is processed for expiration (`web/main/routes/vault.ts#570–602`)

Since this applies only to vaults that are already due for deletion, it does not appear to create any security concern. However, allowing an unauthenticated client to trigger data deletion could become a component in an exploit chain.

Recommendations

Short term, require that clients are logged in when calling the `/access` endpoint, and consider moving vault expiration handling to a scheduled job instead of triggering it when a user accesses an expired vault.

Long term, ensure that all client actions are appropriately authenticated.

3. Variable-time comparison of secrets

Severity: **High**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-OBSYNC-3
Target: `web/main/{routes/user.ts,core/user.ts}`	

Description

Comparison functions can reveal a significant amount of information through the amount of time that they take. For example, string-comparison functions such as `memcmp` exit after finding the first unequal position in the strings being compared. By observing how long a comparison takes, an observer can learn how long the common prefix is between two strings; if one string is attacker-controlled and the other is a secret such as a password, the attacker can learn the secret by guessing it one character at a time, and inferring that the guess is correct for values that cause the comparison to take a longer time.

The Obsidian Sync system compares password-reset tokens via the `!==` operator, as shown in figure 3.1.

```
[REDACTED]
```

Figure 3.1: The password reset key is compared using the `!==` operator (`web/main/routes/user.ts#381–396`)

In addition, MFA recovery codes are checked against entries in the recovery code database via SQL queries with the submitted code in a `WHERE` clause, as shown in figure 3.2.

```
[REDACTED]
```

Figure 3.2: MFA recovery codes are stored in a database and looked up via a `WHERE` clause (`web/main/routes/user.ts#72–81`)

Finally, the OTP secrets are used as the key of an associative array to enforce a rate limit on TOTP authentication, as shown in figure 3.3.

```
[REDACTED]
```

Figure 3.3: OTP secrets are used as the key of the `totpRateLimit` associative array (`web/main/core/user.ts#254–279`)

None of these methods use constant-time comparison, and each one potentially leaks information about the underlying secrets through timing information.

We have not yet found a direct way to exploit either comparison, so we have given this finding a “High” difficulty rating.

In the specific case of the password-reset tokens, we do not believe that this is currently exploitable; V8’s string comparison uses a SIMD-optimized implementation for comparing strings below 32 bytes, which does not appear to have an exploitable timing leak. We have not investigated what timing information is revealed by the SQL query or the OTP timeout associative array.

We have not determined whether these are the only possible timing leaks present.

Exploit Scenario

Alice develops a technique for extracting the timing information from these comparisons. She triggers a password reset on an administrator account, performs a timing attack against the reset-code comparison to reset the account’s password, then performs a timing attack against the MFA recovery code lookup to remove the existing MFA, gaining full administrator control over Obsidian’s systems.

Recommendations

Short term, ensure that comparisons are constant-time and that they cannot reveal partial information about the secrets involved; for example, hash the strings to be compared and then compare them with a timing-safe comparison such as Node’s `crypto.timingSafeEqual` function. Use the user ID instead of the OTP secret as the key for the OTP rate-limit array.

Long term, always perform constant-time operations when handling secrets, and avoid performing *any* direct computation including plaintext secrets unless strictly necessary.

4. Secrets are stored in plaintext	
Severity: **High**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-OBSYNC-4
Target: Various in `web/main/`	

Description
The Obsidian user database stores MFA recovery codes, the OTP secret, and the active password reset key in plaintext. In the event of a database exposure, MFA is trivial to bypass. If the attacker has active read access to the database, they can use the password reset flow to log in to any account at will.

Password reset keys and MFA recovery codes should be treated similarly to passwords, and should be stored exclusively as hashes, to protect against database exposure.

MFA authentication should be performed via schemes such as WebAuthn that do not require the service to store secrets in the same way that TOTP does. However, if TOTP must be used, it should be handled via a separate system than the main user database, with a strict API that does not reveal the secrets of current users.

In addition to the database itself, OTP secrets are stored in a long-lived in-memory table to implement rate limiting. Since memory contents can be exposed in artifacts such as crashdumps, this presents another path through which secrets can be leaked.

Exploit Scenario
Alice discovers a vulnerability that gives her read-only access to the Obsidian user database. She triggers a password reset for an administrator account, reads the password recovery key and OTP secret from the database, and uses them to log in and take over Obsidian's system.

Recommendations
Short term, only store salted hashes of MFA recovery codes and password recovery keys in the user database. Move MFA authentication and OTP key storage into a separate system, to ensure that implementation errors in the normal API do not compromise user accounts.

Long term, factor user authentication into a separate service with clear security boundaries from the rest of the system.

References
- MDN: Web Authentication API

5. TOTP codes can be used multiple times

Severity: **High**	Difficulty: **Medium**
Type: Authentication	Finding ID: TOB-OBSYNC-5
Target: `web/main/routes/vault.ts`	

Description

Time-based OTP authentication (TOTP) uses a shared key to generate a unique code for a given window of time. As an authentication method, it relies on the assumption that anyone able to provide the code must have also generated it, and thus has the shared key. However, if a code can be used more than once while it is still valid, an attacker who sees a TOTP code (for example, by observing a user's screen) can use it immediately after to bypass MFA.

The only restriction on TOTP usage in the Obsidian system is a rate-limit that requires subsequent TOTP authentications to be two seconds apart, as shown below in figure 5.1.

```
[REDACTED]
```

Figure 5.1: TOTP rate-limiting logic (`web/main/core/user.ts#258–263`)

Since the OTP timestamp quantization used is 30 seconds, and the authentication allows for one time period before or after, this gives an attacker over one minute in which they can potentially reuse an OTP code.

Exploit Scenario

Alice wishes to take over Bob's account. Posing as tech support, she manipulates Bob into installing a remote access trojan, and is able to learn Bob's password. She does not have access to Bob's MFA device, but when Bob logs in the next time, she sees the TOTP code and is able to reuse it two seconds later, compromising Bob's account.

Recommendations

Short term, in addition to rate-limiting TOTP authentication to prevent brute-force attacks, record which time-steps have had successful TOTP logins, reject logins reusing codes, and alert the user of suspicious activity if code reuse is detected.

Long term, ensure that any one-time-use material can be used only once.

6. Password reset does not require MFA authentication

Severity: **Medium**	Difficulty: **Medium**
Type: Authentication	Finding ID: TOB-OBSYNC-6
Target: `web/main/routes/vault.ts`	

Description
Major changes to a user account that can affect login should always require MFA. Obsidian's password reset requires only that the emailed password reset key is present, at which point a user's password can be changed. The `/resetpass` endpoint is shown below in figure 6.1.

```
[REDACTED]
```

Figure 6.1: The password reset endpoint, which does not require MFA authentication (`web/main/routes/user.ts#381–413`)

If a user's email is compromised but their MFA is not, this allows an attacker to permanently lock their target out of their Obsidian account.

Exploit Scenario
Alice compromises Bob's email account and locks him out of it. She then performs an Obsidian password reset and changes Bob's password to something he does not know. Without access to his email account, Bob is unable to reset the password himself, and he is locked out of his Obsidian account.

Recommendations
Short term, require MFA authentication to reset the password of an account with MFA activated.

Long term, ensure that an attacker with only partial access to a user's credentials cannot lock that user out of their account.

7. Fixed-password authentication allows unauthorized use of /size endpoint	
Severity: **Low**	Difficulty: **Medium**
Type: Authentication	Finding ID: TOB-OBSYNC-7
Target: `web/main/routes/vault.ts`	

Description

The `/vault/size` API endpoint is used for reading and updating the size metadata for a vault, to enforce usage limits. This is intended to only be called internally, and is authenticated via a `token` parameter provided by the caller, as shown below in figure 7.1.

```
[REDACTED]
```

Figure 7.1: Access to the `/size` endpoint is restricted by the isValidAuthToken function (`web/main/routes/vault.ts#668–675`)

The token is formatted as "`<hash>:<timestamp>`, and is considered valid if the timestamp is within five minutes of the current time and the hash is the result of hashing a fixed password with the given timestamp. This check is shown below in figure 7.2.

```
[REDACTED]
```

Figure 7.2: Auth token validation, with the liveness check highlighted (`web/main/core/validation.ts#14–34`)

In addition to the potential risk of leaking the hard-coded password, there is also a flaw in this function that allows the creation of indefinitely valid tokens. Since JavaScript numbers are floating-point, the special value `NaN` is a possible output of the `parseInt` function, and in fact is the result of the expression "`parseInt("")`". Since `NaN` is guaranteed to fail all comparisons, the comparison in `isValidAuthToken` fails, and a valid auth token with an empty timestamp never expires.

In addition, the use of the variable-time "`===`" operator potentially allows a client to use a side-channel attack to learn the correct hash for a given timestamp without learning the password itself. As discussed in TOB-OBSYNC-3, we do not know of a viable timing attack on this operator for short strings such as hashes, but we would recommend using explicitly constant-time comparison functions regardless.

Exploit Scenario

Alice learns the hard-coded password and disrupts Obsidian by changing the recorded sizes of various vaults. She sets her own vault's size to 0, allowing her to bypass Obsidian's storage usage limits.

Recommendations

Short term, replace this authentication method with certificate-based authentication, such as mTLS.

Long term, avoid ad-hoc authentication schemes and use well-vetted public key authentication protocols whenever possible.

8. Repository contains hard-coded credentials	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Exposure	Finding ID: TOB-OBSYNC-8
Target: Various	

Description

The Obsidian source repository includes multiple hard-coded credentials, shown below in figure 8.1, as reported by the `trufflehog` tool.

```
✅ Found verified result 🐷🔑
Detector Type: [REDACTED]
Decoder Type: PLAIN
Raw result: [REDACTED]
Commit: [REDACTED]
Email: [REDACTED]
File: [REDACTED]
Line: 94
Timestamp: 2023-12-18 16:41:18 +0000

✅ Found verified result 🐷🔑
Detector Type: [REDACTED]
Decoder Type: PLAIN
Raw result: <...>
Commit: [REDACTED]
Email: [REDACTED]
File: [REDACTED]
Line: 177
Timestamp: 2020-09-30 18:10:20 +0000
```

Figure 8.1: Trufflehog output describing the credentials in the repository

Exploit Scenario

Alice gains access to the Obsidian source repository and discovers these credentials. She then uses them to disrupt Obsidian's operations.

Recommendations

Short term, remove these credentials from the repository, and rotate them.

Long term, add a credential-detecting tool such as Trufflehog to the integration test suite.

9. Deterministic encryption of file hash endangers file confidentiality	
Severity: **Medium**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-OBSYNC-9
Target: `app/plugins/sync/sync.ts`	

Description

During the synchronization process, files are uploaded along with associated metadata. The metadata is encrypted with a deterministic encryption scheme using the SIV construction. File hashes are also uploaded as SIV-encrypted metadata. Since the hashing and encryption scheme are deterministic, an adversary can learn information about an uploaded file by exploiting the determinism of the file encryption.

Figure 9.2 shows that files are hashed before they are uploaded. Figure 9.3 shows that the hash is deterministically encrypted before and uploaded to the server along with other data.

```
[REDACTED]
```

Figure 9.2: Files are hashed before being uploaded. (app/plugins/sync/sync.ts#L2131–L2144)

```
[REDACTED]
```

Figure 9.3: File hashes are deterministically encrypted. (app/plugins/sync/sync.ts#L2781–L2810)

The expected security guarantee of an end-to-end encrypted system essentially demands that an attacker cannot learn meaningful information about encrypted files other than the length of the message. In particular, an attacker who chooses two files of the same size and sees the encryption of one of the files (chosen at random) should be unable to tell which file has been encrypted, guaranteed by usual encryption schemes that use nonces or IVs. On the other hand, deterministic processing of plaintext usually fails to provide this guarantee. Because SIV construction is deterministic, it leaks information about encrypted messages; it is therefore not suitable for encrypting data that is not guaranteed to have high entropy.

Obsidian's implementation uploads encrypted hashes to detect file duplicates. However, such functionality is at odds with the goals of end-to-end encryption because it reveals potentially harmful information to the server.

Exploit Scenario

Alice, a political activist, receives a confidential note from Bob and decides to upload it. Later, Bob's file is leaked, and a compromised server wishes to learn what Alice is engaged in. The server, who is posing as an activist, lures Alice into uploading different files from Bob. The server observes the uploaded data and compares the encrypted metadata to determine which file Alice uploaded.

Recommendations

Short term, use a randomised encryption scheme to encrypt the file hash and other metadata. Such a change would require extensive changes to the file data structures. For example, the encrypted file data structure could contain a random ID, a file version, and a data blob. The data Blob can be an encrypted data structure of the file content and all required metadata. All fields, including the ID and file version, should be cryptographically tied as associated data with the encrypted data Blob. Although the server loses all capabilities to mitigate file duplicates, this task is arguably not the server's responsibility. A user who maliciously uploads duplicate files is acting against their own interest. On the other hand, duplicated uploads due to a software bug should be addressed by fixing the issue on the client.

Long term, formalize the expected security guarantee of the sync system, and ensure that the protocol and choice of primitives provide these guarantees.

10. General lack of cryptographic binding between file content and metadata

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-OBSYNC-10
Target: `web/sync/server.ts`, `app/plugins/sync/sync.ts`	

Description
In Obsidian Sync, files are uploaded along with associated metadata. However, the file path is not tied to the path content, allowing a server to switch path content during a "sync-down" operation without detection. Other metadata, including paths and file hashes, is not cryptographically tied to the file content and is deterministically encrypted.

Figure 10.1 shows that metadata, such as the path, is encrypted independently of the file content.

```
[REDACTED]
```

Figure 10.1: Metadata is encrypted separately from file content (`app/plugins/sync/sync.ts#2752–2778`)

The lack of cryptographic binding breaks the expected security guarantees and opens up a large attack surface for the adversary, potentially allowing a break of end-to-end security.

Exploit Scenario
Alice has `file1` and `file2` stored on `path1` and `path2`, respectively, on her primary device. She uploads them to the server in an end-to-end encrypted folder. The server sends notifications with switched paths when she sets up and syncs down. Alice now downloads and decrypts the content on the new device and has the file content switched around.

Recommendations
Short term, tie the paths and all other metadata to the file encryption and add them as associated data to the file ciphertext.

Long term, formalize the integrity requirements for the sync system and ensure that the implementation guarantees them.

11. The authentication protocol does not guarantee proof of possession of the vault key

Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-OBSYNC-11
Target: `src/app/plugins/sync/{sync.ts, sync-encryption.ts}`, `web/sync/server.ts`	

Description
Vaults are protected by a password chosen by the user. The password is then used to authenticate access to a vault or to encrypt files. The authentication method demands only that the user upload a hash derived from the password, and then the server checks the provided hash against the expected hash. As a consequence, the vault authentication mechanism does not guarantee that access to the vault data is predicated on knowledge of the password, and furthermore, it does not protect against adversaries that can break into the server.

```
[REDACTED]
```

Figure 11.1: Authentication compares the provider keyhash against the expected one. (`web/main/routes/vault.ts#561–666`)

Exploit Scenario
The attacker compromises the user's main account and temporarily gains access to the sync server to retrieve the `keyhash`. Now, they can download the encrypted vault and do offline brute-forcing.

Recommendations
Short term, derive a value from the keyhash using `HMAC(keyhash, global_authentication_label)` and store that value. During authentication, require that the user send a `keyhash`, recompute `HMAC(keyhash, global_authentication_label)`, and compare the values in constant-time.

Long term, consider using a Password Authenticated Key Exchange (PAKE) protocol like the OPAQUE protocol to restrict the adversary to online only brute-forcing and provide fresh authentication guarantees.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Quality Findings

This appendix contains recommendations for findings that do not have immediate or obvious security implications. However, implementing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- **Functions with long bodies.** Several functions have long bodies. For example, the `_sync` function has over 700 lines of code. Functions with long bodies degrade the quality of the codebase; they are harder to read and make it harder to write unit tests.

C. Automated Testing

Semgrep

The community version of Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be a single rule, a directory of rules, or the name of a ruleset hosted on the Semgrep registry. To use the default configuration, use the command-line argument `--config auto`. Trail of Bits' public ruleset can be used by running `semgrep --config p/trailofbits`.

For more details on how to run Semgrep, including how to set up automatic scanning as part of your CI/CD pipeline, see Trail of Bits' Testing Handbook section on Semgrep.

D. Cryptographic Hardening Recommendations

End-to-end encrypted systems are, by nature, expected to protect against a variety of attacker models. Users expect end-to-end encryption to guarantee that even a server cannot access any data. In addition, the usual expectations of traditional server-based applications still need to be met. In summary, the attacker models under consideration are: malicious server, network adversaries, and temporary server break-ins. We provide recommendations below to strengthen the Sync system against these attackers.

Authentication

Access to a vault is guarded by an authentication step. The current implementation uses a variant of traditional password-based authentication that does not guarantee actual knowledge of the password (TOB-OBSYNC-11). Furthermore, authentication is vulnerable to offline attacks against a network adversary or one who temporarily breaks into the server. Our main recommendation is to restrict the network and prevent adversaries from launching online attacks. Unlike a password brute-force attack, an online attack requires the adversary to interact with the server to guess the password. Consequently, the server can implement appropriate rate limiting. We recommend the following to improve the authentication to the vault:

- **Authentication using OPRF**. An Oblivious Pseudorandom Function (OPRF) allows a client who holds a password to interact with a server that knows a cryptographic secret. At the end of the protocol, only the client learns a random value associated with the password, while the client learns nothing about the server secret. The obtained random secret can be used as a drop-in for a high-entropy password. OPRFs are best used for authentication through a Password-Authenticated Key Exchange, like the OPAQUE protocol.
- **Security against malicious servers.** Protection against a malicious server requires the user to store a high-entropy secret. Management of high entropy presents a number of challenges for users. Solutions like the PRF extension of passkeys provide a reasonable solution to these issues. The PRF extension allows deriving a random secret for specific uses. In short, given a base, high-entropy secret, subkeys can be derived from the base secret and a unique (non-secret) input string. Passkey's PRF extension relies on HMAC.

File Encryption

File encryption is expected to prevent anyone who does not know the secret from learning any information about the message and provide integrity guarantees. However, these fundamental guarantees are not enough in an end-to-end encrypted system where the attacker is afforded more power. It is also expected that learning about the decryption key

of a given file does not lead to learning information about other files. We recommend implementing the following file encryption methods:

- **Per-file key encryption.** It is best practice to sample unique keys for each file that is encrypted. Keys freshly sampled from a good randomness source ensure that a single file encryption key does not reveal information about other files. Furthermore, it is recommended to limit the usage of a single encryption key to not weaken its security.
- **Strengthen vault sharing.** Sharing vaults with other users raises another consideration for the encryption scheme. A vault owner should not be able to share a file with two users in a way that decrypts the file to different content. This property is known as key commitment. One scheme that provides key commitment is the DNDK construction of Gueron. The construction allows for the use of larger nonces while providing key commitment security.

Metadata Protection

Metadata is built from auxiliary content related to the underlying file. In the Obsidian Sync system, examples of metadata are paths and file hashes. Although metadata is different from the file content, it can reveal information about the file content and undermine the system's guarantees. Therefore, metadata must also be adequately protected. For optimal security guarantees, the visible metadata should be unrelated to the file content and other metadata. We recommend the following to improve the security of metadata:

- **Metadata encryption**. Consider encrypting metadata along with the file content, following the recommendations in the File Encryption recommendations.
- **Metadata integrity.** Metadata that remains publicly accessible must be cryptographically tied to the encrypted content. Cryptographic binding can be achieved by using the associated data field of an authenticated encryption scheme.

Key Hierarchies

In the current implementation, a single key derived from the vault password is used for several purposes. As a consequence, password rotation is an expensive operation; furthermore, leakage of a file encryption key would enable decryption of all files. An attacker who holds a ciphertext of a deleted file would be able to decrypt this file if a file encryption key leaks. Cryptographic best practices recommend using keys for a single purpose and limiting the lifetime or usage of a given key. Using appropriate key hierarchies ensures that keys are used for a single purpose and are largely independent. We recommend implementing the following to strengthen key usage in Obsidian Sync:

- **File encryption keys.** File encryption keys should be generated uniformly at random for each file. They should be encrypted by a key encryption key and stored in a header next to the encrypted file.

- **Key encryption key.** A global key encryption key (KEK) can be used to encrypt (or wrap) individual file keys using key-committing Authenticated Encryption. The KEK can be encrypted with a master key derived from the password.
- **Master key**. The master key, `mk`, is derived from the password and a high-entropy value. It can be used to derive a key that encrypts the KEK and to derive another cryptographic secret used for authentication, as described in Authentication. As a consequence, key rotation is fairly inexpensive and requires only re-encryption of the KEK and derivation of a new authentication secret.
- **Password:** A low-entropy and memorable secret chosen by the user
- **Base secret:** A high-entropy secret used along with the password to derive the master key

Secret Management

The system's security depends on the good management of cryptographic secrets. Encryption keys and tokens must be stored securely to ensure that an attacker who breaks into the server does not learn valuable information on encrypted files or become capable of breaking authentication. We recommend the following to improve management of secrets:

- **Encryption of sensitive information**. All information that should be known only to the server and a specific user must be encrypted. This data includes managed passwords, OTP secrets, and any data that grants special privileges. Data should be encrypted with an AEAD scheme. All context information must be cryptographically tied to the ciphertext using the Associated Data field.
- **Secure storage on HSM.** Encryption keys should be stored in a dedicated HSM, which can be local or from a cloud provider such as CloudHSM. Encryption keys must be appropriately backed. Additionally, a robust key rotation plan must be in place and regularly tested.
- **Storage-less key management.** The storage burden of key management can be significantly lowered by using appropriate key derivation schemes. In particular, given a master key, several child keys can be derived from the master key using a pseudorandom function like HMAC and a unique label.

References

- Hofmann & Truong: End-to-End Encrypted Cloud Storage in the Wild: A Broken Ecosystem
- Backendal et al.: A Formal Treatment of End-to-End Encrypted Cloud Storage

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On November 4, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Obsidian team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 11 issues described in this report, Obsidian has resolved five issues, has partially resolved three issues, and has not resolved the remaining three issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Use of Math.random for salt and password generation	High	Resolved
2	Logged-out clients can look up and trigger deletion of vaults with inactive subscriptions	Informational	Unresolved
3	Variable-time comparison of secrets	High	Resolved
4	Secrets are stored in plaintext	High	Partially Resolved
5	TOTP codes can be used multiple times	High	Resolved
6	Password reset does not require MFA authentication	Medium	Resolved
7	Fixed-password authentication allows unauthorized use of /size endpoint	Low	Partially Resolved
8	Repository contains hard-coded credentials	Medium	Resolved
9	Deterministic encryption of file hash endangers file confidentiality	Medium	Unresolved

10	General lack of cryptographic binding between file content and metadata	High	Unresolved
11	The authentication protocol does not guarantee proof of possession of the vault key	Low	Partially Resolved

Detailed Fix Review Results

TOB-OBSYNC-1: Use of Math.random for salt and password generation
Resolved in commit 214698e and commit 90fcea9. The function `generatePassword` uses `crypto.randomInt` to generate a random password from a specified character set. Furthermore, the sync plugin now uses `crypto.getRandomValues` to generate salts.

TOB-OBSYNC-2: Logged-out clients can look up and trigger deletion of vaults with inactive subscriptions
Unresolved. The Obsidian team decided to keep the endpoint as is to enable the sync server to clean up data after the grace period.

TOB-OBSYNC-3: Variable-time comparison of secrets
Resolved in commit 6457077. Password recovery tokens and recovery codes are hashed with HKDF using a random salt. Constant-time comparisons are performed using `crypto.timingSafeEqual`. Finally, TOTP secrets are no longer used as indexes to look up the rate-limiting state.

TOB-OBSYNC-4: Secrets are stored in plaintext
Partially resolved in commit 6457077. Password reset tokens and recovery codes are no longer stored in plaintext. Instead, hashes of these values are stored in the database. However, several sensitive values are still stored plaintext, including TOTP secrets and managed vault passwords. The Obsidian team has stated that they plan to explore server-side encryption options.

TOB-OBSYNC-5: TOTP codes can be used multiple times
Resolved in commit 6457077. TOTP attempts are limited both per user and per token. Users' attempts are limited to one attempt every two seconds, and a single TOTP may not be reused in a time window larger than two minutes. The dual-rate-limiting system ensures that a valid OTP code cannot be reused.

TOB-OBSYNC-6: Password reset does not require MFA authentication
Resolved in commit ee59321. MFA authentication is now enforced during password reset.

TOB-OBSYNC-7: Fixed-password authentication allows unauthorized use of /size endpoint
Partially resolved in commit ee59321. The API secret has been changed and is no longer stored in a repository. Instead, the API secret is loaded at runtime from Ansible. Furthermore, timestamps are ensured not to be `NaN`, and validation of tokens during validation is performed using constant-time comparison. However, the API secret is stored on the server side in Ansible, and the authentication protocol does not fully guarantee desirable properties such as freshness.

TOB-OBSYNC-8: Repository contains hard-coded credentials
Resolved in commit ee59321. All credentials have been changed. Sensitive credentials are no longer stored in the repository; instead, they are loaded from Ansible. We recommend using git-filter-repo to remove any remaining credentials from the repository.

TOB-OBSYNC-9: Deterministic encryption of file hash endangers file confidentiality
Unresolved. The Obsidian team has stated that they plan to address the core issue at a later stage. The team added a limitations section to their public documentation to warn users about this issue in commit ec32a5c.

TOB-OBSYNC-10: Deterministic encryption of file hash endangers file confidentiality
Unresolved. The Obsidian team has stated that they plan to address the core issue at a later stage. The team added a limitations section to their public documentation to warn users about this issue in commit ec32a5c.

TOB-OBSYNC-11: The authentication protocol does not guarantee proof of possession of the vault key
Partially resolved in commit ee59321. User-provided keyhash values are now verified using constant-time comparison. However, the Obsidian team has stated that they plan to address the core issue at a later stage.

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.