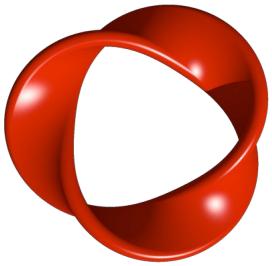

Secfault Security

SSO Client Application 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	2022-10-01	First Draft
0.2	Gregor Kopf	2022-10-12	Internal Review
0.3	Jennifer Gehrke	2022-12-01	Retest Status
0.4	Gregor Kopf	2022-12-05	Added Customer Feedback
0.5	Gregor Kopf	2023-03-01	Additional Customer Feedback after the Retest
1.0	Gregor Kopf	2023-03-03	Final Version
1.1	Gregor Kopf	2023-03-16	Additional Grammar and Layout Corrections

Table of Contents

1 Executive Summary

Secfault Security was tasked by Agilebits with a security review of selected components of the 1Password ecosystem, namely the beta SSO integration along with its accompanying changes to the codebase.

The review has been performed in the time frame from 2022-09-19 to 2022-10-14. This document describes the results of the project.

During the review a number of issues, which are described in detail in section 4 of this document, have been identified. The most severe issues included a design problem that enabled the b5 server to partially compromise users' vault data. Furthermore, issues have been identified in the session handling functionality, which could be used to bypass the requirement of having to authenticate with an SSO provider in regular intervals. During a retest performed in November 2022, these issues have been found to be fixed.

Overall, the reviewed codebase left a positive impression. The code is well-structured and readable and has been implemented with security in mind.

After having received a draft version of this document, Agilebits provided feedback on the identified issues, which can be found in section 6 of this document.

In November 2022 a retest of the majority of previously identified security issues was commissioned by Agilebits. In total, the status of five issues was inspected. All issues have been found to be addressed. The issues related to bypassing the SSO session duration requirements have been discussed with Agilebits, who provided a more in-depth description of the desired semantics of the session duration mechanism. Additional details are provided in sections 4.7 and 4.5 of this document.

2 Overview

1Password is a password manager product developed and maintained by AgileBits Inc. The solution provides a secure place for customers to store various passwords, software licenses, and other sensitive information in virtual vaults.

Agilebits tasked Secfault Security with a review of the SSO integration, which Agilebits recently added to the solution. In section 2.1 of this document, a description of the project's scope is provided. Section 2.2 provides details on the test procedures.

2.1 Target Scope

Secfault Security performed a combination of dynamic and static security testing within this project. Agilebits

provided Secfault Security with the source code and necessary binary builds for the Server API, Web Application (1Password.com), 1Password CLI, 1Password in the Browser, and 1Password for Desktop and Mobile.

Agilebits specified the following issues for the retest in November 2022:

- MITM Attack against Encrypted Credentials
- MITM Attack during Device Enrollment
- Prolonging Sessions by Nested Delegation
- MacOS Desktop App not using Secure GUI Restore
- SSO Login Bypass by Session Extension

2.2 Test Procedures

The overall project followed a white-box approach, which means that Agilebits provided the source code, the compiled binaries and technical documentation for the solution. Therefore, the solution has been analyzed by performing a source code review, as well as targeted dynamic testing.

The source code review has been performed in a manual fashion, i.e., without relying on automated vulnerability scanners or similar tools. Besides identifying possible classical implementation weaknesses, one main focus of the review was the identification of potential logic problems. This requires an in-depth understanding of the solution's inner workings, which is best achieved by a manual process.

The dynamic tests have been performed in a targeted fashion. On the one hand, this served the purpose of validating issues identified during the source code review. On the other hand, dynamic tests were also performed to obtain a better understanding of the overall solution and the interplay

of its individual components.

2.3 Project Execution

The project has been executed in the time frame from 2022-09-19 to 2022-10-14.

The consultants assigned to this projects were:

- Jennifer Gehrke
- Maik Münch
- Gregor Kopf

A retest was performed in conjunction with further tests of the 1Password8 client in the time frame of 2022-11-14 to 2022-11-25.

The inspections were performed by the consultants:

- Jennifer Gehrke
- Gregor Kopf

3 Result Overview

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

Description	Chapter	Type	Severity	Status
MITM Attack against Encrypted Credentials	4.1	Design	High	Closed
MITM Attack during Device Enrollment	4.2	Design	High	Closed
Lax Authorization Concept for Delegating Sessions	4.3	Design	Medium	Won't Fix
Automatic Restoring of Devices	4.4	Observation	Low	Won't Fix
Prolonging Sessions by Nested Delegation	4.5	Design	Medium	Closed
MacOS Desktop App not using Secure GUI Restore	4.6	Code	High	Closed
SSO Login Bypass by Session Extension	4.7	Code/Design	Medium	Closed

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 MITM Attack against Encrypted Credentials

Summary

Type	Location	Severity	Status
Design	Encrypted Credential Storage	High	Closed

Technical Description

While analyzing the overall design changes introduced for the SSO feature, Secfault Security identified a weakness that would allow the b5 backend to possibly compromise information stored in users' vaults.

The entries in a vault are not stored in plain text on the backend servers. Instead, they are encrypted with a key that is ultimately only known to the user. While this design is recommendable in terms of privacy and security, it is not directly compatible with SSO-based logins: after logging in with their IDP, the users would have to enter their "original" password again, in order to be able to access the key material. In order to address this issue, Agilebits decided to implement a new process for SSO-based sign-in. The core idea is to store the user's key material (their `CredentialBundle`) on the b5 backend, encrypted with a key that is tied to the user's device (e.g., a key that is stored in the operating system's key chain). This way, clients could retrieve the key material from the server, locally decrypt it, and then subsequently use it to access their vaults' contents.

The original version of this design made use of asymmetric encryption: during the initial device enrollment phase, a client would generate an asymmetric key, store it in the system's keychain and then use this key to encrypt their `CredentialBundle`. The encrypted `CredentialBundle` would then be uploaded to b5, along with the public key used by the client. As b5 does not receive the secret key, it is not able to directly decrypt the `CredentialBundle`.

However, as the public key **is** known to b5, it can generate its own version of an encrypted `CredentialBundle`, containing cryptographic key material that it has access to. A client that would use such a `CredentialBundle` could not access their vault entries (as the keys in the `CredentialBundle` would not be the ones originally set by the user). However, when creating new entries, the client would use the server-generated key material, and subsequently the server would

be able to decrypt the stored vault entries.

Recommended Action

The issue has been discussed with Agilebits during the engagement. A fixed version of the design has already been provided to Secfault Security, so no further action is required.

Retest Status

During the retest performed in November 2022, this issue has been found to be fixed. The used encryption scheme has been changed from asymmetric encryption to symmetric encryption. This prevents the backend from encrypting (or decrypting) `CredentialBundles`, as the used symmetric encryption key is not known to the backend.

4.2 MITM Attack during Device Enrollment

Summary

Type	Location	Severity	Status
Design	Device Enrollment	High	Closed

Technical Description

As described in section 4.1 of this document, the SSO feature required a number of changes to the way clients log in on the b5 backend. As users can have multiple client devices, each client device now has to have access to the user's `CredentialBundle`. This is achieved by the new device enrollment process. During this process, a user first signs in on a new device using SSO. If the sign-in succeeds and the backend finds no encrypted `CredentialBundle` for the device, the user is prompted to open the OnePassword application on one of their existing devices. The existing device and the new device then cooperate in order to make the user's `CredentialBundle` accessible to the new device.

In a first step, the new device generates an asymmetric keypair and sends the public key to the existing device via the b5 backend. Now the existing device generates a one-time code and a random salt. The salt is encrypted with the new device's public key and sent to the new device via b5. The one-time code is displayed on the existing device's screen and has to be manually typed into the new device. The new device then computes an HMAC over the one-time code and the provided salt and sends the result back to the existing device (again, via b5). If the HMAC matches the existing device's expectations, the existing device locally decrypts its `CredentialBundle` and re-encrypts it with the new device's public key. The encrypted data is then uploaded to b5.

The problem with this process is similar to the one described in section 4.1: the b5 backend could maliciously replace the new device's public key with a key that it generated on its own. It could then decrypt all information sent by the existing device, including the random salt. The random salt could be forwarded to the legitimate new device, simply by re-encrypting it with the device's actual public key. The new device would then perform the correct HMAC calculation, so that the existing device would encrypt its `CredentialBundle` with the public key that has been generated by the b5 backend. This would result in the ability to compromise the user's key material.

It should be noted that this issue has been independently identified by Agilebits during their analysis of the issue described in section 4.1 of this document.

Recommended Action

The issue has been discussed with Agilebits during the engagement. A fixed version of the design has already been provided to Secfault Security, so no further action is required.

Retest Status

During the retest performed in November 2022, this issue has been found to be fixed. The overall scheme for provisioning new devices has been reworked, and is now based on setting up a secure channel between the two involved devices using CPace. This prevents the backend from modifying data in transit during the provisioning.

4.3 Lax Authorization Concept for Delegating Sessions

Summary

Type	Location	Severity	Status
Design	Delegated Sessions	Medium	Won't Fix

Technical Description

A review of the server-side implementation of the new delegated sessions feature revealed that the authorization controls in place focus on assigning the session to the correct user and account only. No checks were found that apply to restricting clients for which a delegated session can be requested.

Consequently, the current design should allow any authenticated application to request delegated sessions for any other device. Due to routines automatically registering unknown devices in this process, even new clients can be specified.

Based on this result, further considerations were made regarding the necessity and practicability of additional restrictions. Since the feature is currently only supported by the desktop applications to request sessions for the CLI client, local attacks can be reduced to attacks on the native message communication between these two clients.

However, another question in this context is whether all client types have the same permissions. In case any device type exists with less permissions than others, a delegated session could be utilized to elevate those. Such a situation was actually found to be present by supporting device access restrictions that can be applied on a vault basis, as illustrated in Figure 1.

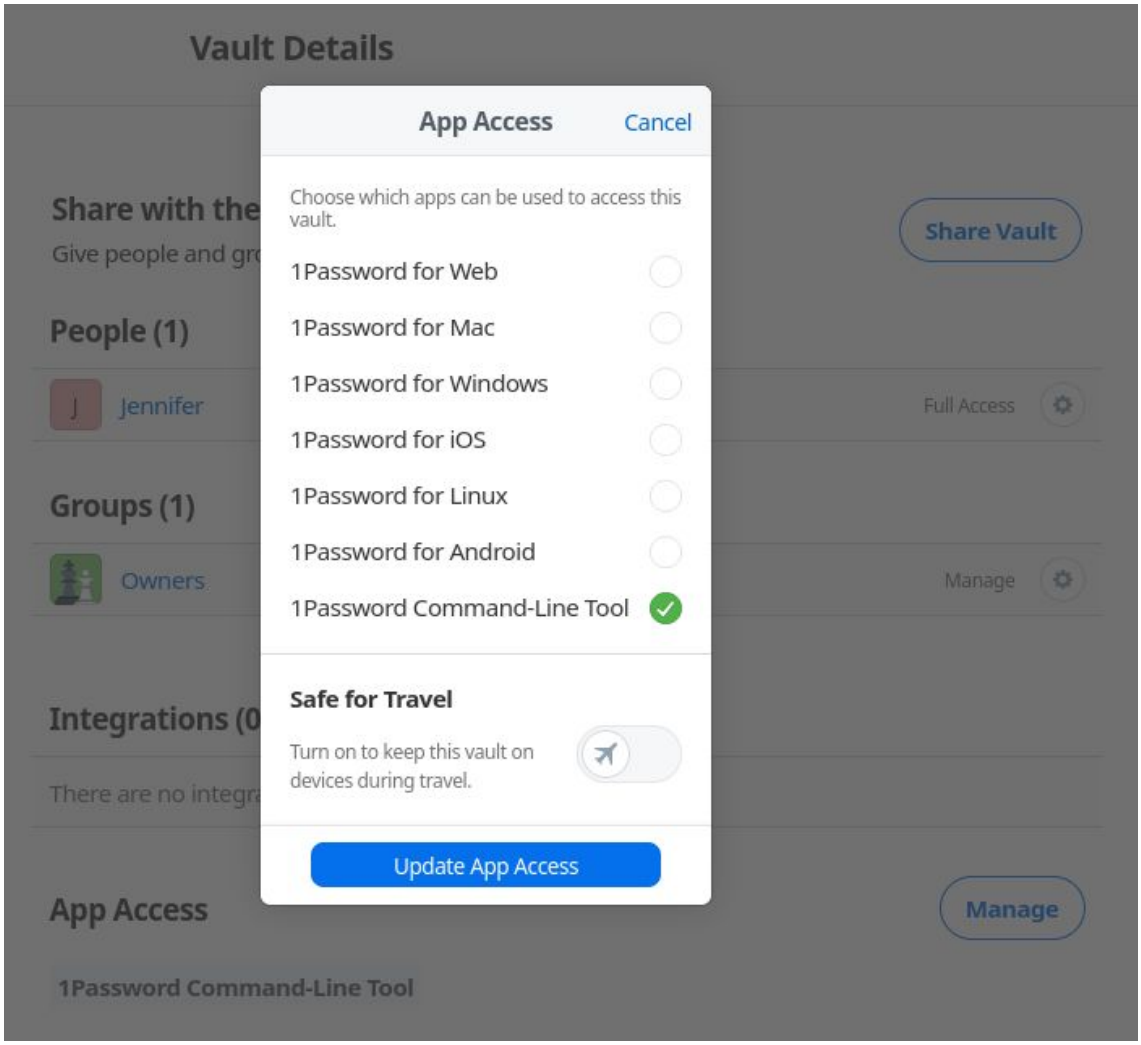

Figure 1 - Vault App Access

This option suggests that specially critical vaults can be protected from being accessed by clients running on less trusted devices, e.g. mobile phones. However, in the current design a hijacked mobile phone session should still be able to requests sessions for other device types, such as desktop applications, resulting in access to the vault.

Recommended Action

There are multiple options for addressing this issue. One option could be to accept the emerging risk and to educate users about the fact that the device restriction feature is not intended to be used for security-related decisions.

Another option would be to consider the introduction of more strict client restrictions to the current design. While it is a known problem to reliably identify the device type in a client to server communication, the issue might be addressed by enforcing specific device enrollment steps.

By implementing a mandatory device enrollment, which seems to be in place for several client

types already, the issuing of delegated sessions could be restricted to existing devices. In this process or during a later "direct" session, a client should be required to enable that delegated sessions can be requested for it and optimally provide a list of devices authorized to do so. During these steps, the chosen procedures must ensure that the user agrees with these settings. It must be remarked at this point, that the requesting party must no longer be able to change any server-side information related to the target device, as currently supported.

To achieve an even higher level of security, cryptographic means could be introduced during this enrollment on a client instant basis. Those could allow to validate that a delegated session request originates from the target client, e.g. employing an end-to-end challenge-response authentication mechanism based on signatures. Further the response could be wrapped in a way that the contents remain confidential with respect to the intermediary client.

Retest Status

This security issue was not included in the retest in November 2022. Please see comments from 1Password in section 6 of this document, which explain the situation.

4.4 Automatic Restoring of Devices

Summary

Type	Location	Severity	Status
Observation	Delegated Session	Low	Won't Fix

Technical Description

During the inspection of the routines creating delegated sessions on the server side, it was observed that deleted devices requesting such a session are automatically restored. Please note the following excerpt from `b5-main/server/src/logic/action/auth.go` that contains the main procedures related to this new feature:

```
1988 func DelegateSession(acs *access.VerifiedAccess, device *api.Device)
(*api.DelegateSessionResponse, api.Status) {
1989    acs.MustBegin("DelegateSession")
1990    defer acs.AutoRollback()
[..]
2021    existingDevice, err := acs.UnsafeFindDeviceByUUID(account.ID, user.ID,
device.UUID, lock.None)
2022    if err != nil {
2023       acs.LogError("DelegateSession failed to UnsafeFindDeviceByUUID for
uuid", device.UUID, "and user", user, ":", err)
2024       return nil, api.Status{Code: api.StatusInternalServerError}
2025    }
2026
[..]
2033
2034    if existingDevice.IsDeleted() {
2035       acs.LogInfo("DelegateSession: Device", existingDevice.UUID, "for
user", user.ID, "exists but is deleted, reauthorizing in cache")
2036       err = acs.ReauthorizeDeviceInCache(existingDevice)
2037       if err != nil {
2038          acs.LogError("DelegateSession failed to
ReauthorizeDeviceInCache:", err)
2039          return nil, api.Status{Code: api.StatusInternalServerError}
2040       }
2041    }
```

Here, the device UUID that is part of the session request is taken to perform a lookup in the device cache. In case the device's state is found to be `DeviceStateDeleted` it is automatically set to `DeviceStateReauthorized`.

Due to time limitations, the use-cases that lead to the deletion of a device were not retraced in the b5 code base. From the impression gained by using the Web UI, it is assumed to be related to the

device deauthorization option offered at the user's profile.

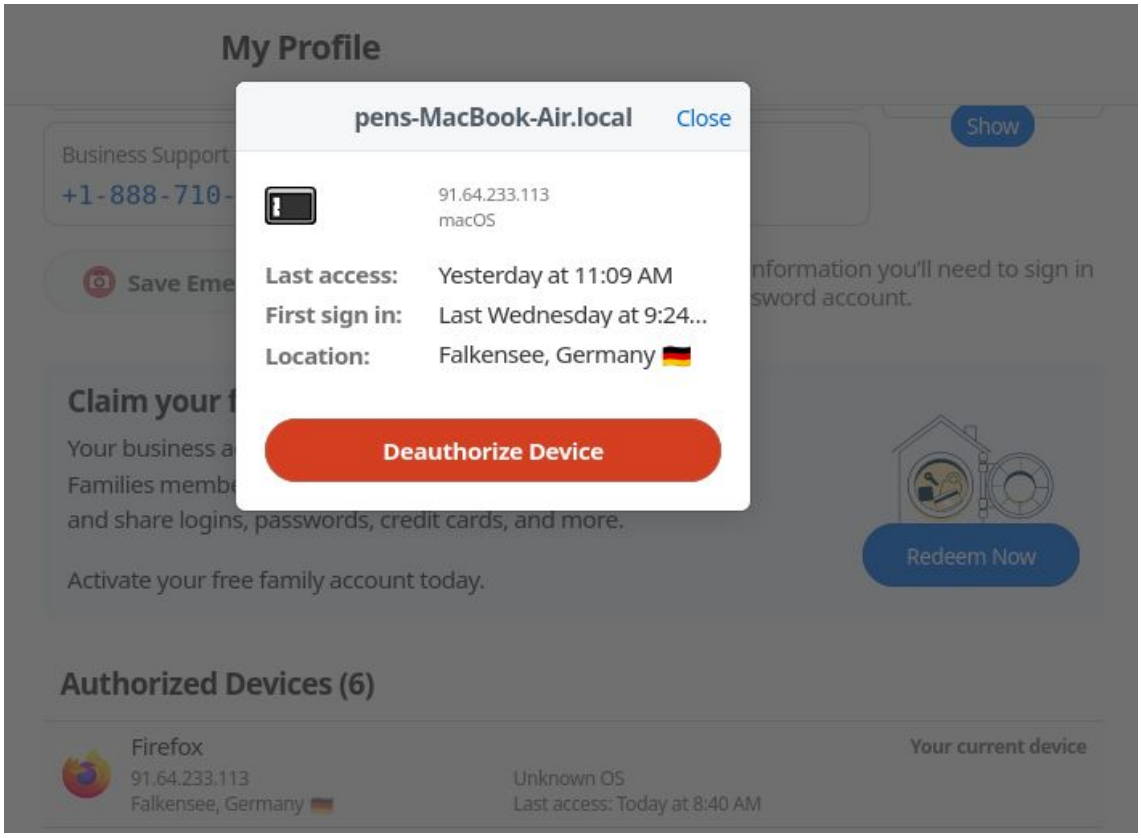

Figure 2 - Deauthorize Device via Web UI

Dynamic tests showed, that a deauthorized CLI client will be added again to the list of authorized devices in case a delegated session is requested for it afterwards. During this process, no differences to the regular flow were observed. Since no special malicious crafting of the delegated session request was performed, this behavior might possibly not meet user security expectations.

The actual advantage an attacker can gain from it, however, is considered very limited to not-existent. As outlined in section 4.3, the delegated session feature currently allows to automatically register new devices anyway. Therefore, an attacker could add new devices instead of restoring deleted ones, without decreasing their abilities.

Further, this attack scenario has the precondition of getting access to a valid b5 session or sending crafted native messages to a desktop application.

Recommended Action

The issue should be addressed in conjunction with the current design limitations described in section 4.3, after defining the security level that should be applied for the delegated sessions feature. In case it is decided to restrict the set of clients that a delegated session can be requested for, no automated restoring of a client should take place during session creation.

Reproduction Steps

In order to reproduce this observation, please proceed as follows:

1. Setup a test machine utilizing a desktop client and CLI application both supporting delegated session.
2. Add the CLI to the list of authorized device by requesting a delegated session for it.
3. Use the Web UI to deauthorize the CLI application for the respective user.
4. Request a new delegated session for the CLI.
5. Check the list of authorized devices in the Web UI and find the CLI application to be present again.

Retest Status

This security issue was not included in the retest in November 2022. Please see comments from 1Password in section 6 of this document, which explain the situation.

4.5 Prolonging Sessions by Nested Delegation

Summary

Type	Location	Severity	Status
Design	Delegated Sessions	Medium	Closed

Technical Description

With the introduction of the new SSO feature, it should be possible to force users to periodically authenticate via SSO to be able to communicate with b5. Any ability of creating sessions of infinite validity could therefore be considered a bypass of this protection measure. This might not only be achieved by prolonging one session (please refer to section 4.7), but also by requesting new sessions by means of an existing one.

While reviewing the details of delegated session entries created by the server, it was observed that those are added to the session cache with a static expiry period. Therefore, it does not depend on the expiry date of the parent session. Considering the new validity requirements mentioned above, the idea of creating nested delegated sessions to renew the session time-frame arises. Nested delegated sessions are explicitly supported by the implementation and are also described in the provided feature documentation. Those are therefore considered an intended characteristic.

Further inspection of the server implementation was performed, to identify whether nested sessions are automatically removed from the cache in case the root or parent session expires. No such mechanisms were found. Consequently, it is assumed that the SSO login requirement can be bypassed by requesting delegated sessions in an automated nested manner, generating a new session shortly before the respective parent session expires. As a result, one obtains a valid b5 session at any point in time.

Attempts to bypass the SSO login by the account user itself are regarded a valid attack scenario, access to an initial valid session can therefore be presumed.

Recommended Action

In order to address this issue, routines should be added to the server implementation that ensure the expiry of all nested sessions on the expiry of the root parent session.

Reproduction Steps

To retrace that delegated sessions are generated with a static expiry time-frame, one can inspect the implementation of the function `DelegateSession` defined inside `b5-main/server/src/logic/action/auth.go`:

```
1988 func DelegateSession(acs *access.VerifiedAccess, device *api.Device)
```


```
(*api.DelegateSessionResponse, api.Status) {
1989    acs.MustBegin("DelegateSession")
1990    defer acs.AutoRollback()
[..]
2069    // save after signin succeeded so that session is given an ID.
2070    acs.UnsafeSaveSession(delegatedSession)
```

This adds the generated session to the session cache using the function `UnsafeSaveSession` implemented in `b5-main/server/src/access/unverifiedaccess.go` that must be considered a wrapper for the function `SaveUserSession` inside `b5-main/server/src/cache/usersession.go`:

```
 70 // SaveUserSession saves the user session in the cache
 71 func (c *Cache) SaveUserSession(userSession *model.UserSession) error {
[..]
 80    ttl := UserSessionUnverifiedTTL
 81    if userSession.IsLoggedIn() {
 82       ttl = UserSessionTTL
 83    }
 84
 85    buf, err := json.Marshal(userSession)
 86    if err != nil {
 87       l.Error(nil, "SaveUserSession failed to Marshal:",
userSession.SessionUUID, err)
 88       return err
 89    }
 90
 91    key := sessionKey(userSession.SessionUUID)
 92    l.Info(nil, "SaveUserSession saving to cache",
util.ObfuscateCacheKey(key))
 93
 94    if err := c.SessionCache.Send("SET", key, buf, "EX", ttl); err != nil {
 95       l.Error(nil, "SaveUserSession failed to SET",
util.ObfuscateCacheKey(key), ":", err)
 96       return err
 97    }
 98
 99    key = sessionsForUserKey(userSession.UserID)
100    if err := c.SessionCache.Send("SADD", key, userSession.SessionUUID); err
!= nil {
101       l.Error(nil, "SaveUserSession failed to SADD",
userSession.SessionUUID, " for user", util.ObfuscateCacheKey(key), ":", err)
102       return err
103    }
104
105    if err := c.SessionCache.Send("EXPIRE", key, ttl); err != nil {
106       l.Error(nil, "SaveUserSession failed to EXPIRE",
util.ObfuscateCacheKey(key), ":", err)
```

```
107       return err
108    }
```

The excerpt shows that the `ttl` variable is set independent of potentially existing parent sessions.

A textual search for the session fields `RootSessionUUID` and `ParentSessionUUID` used to track the delegated session relationships reveals that no automated deletion from the cache is performed on parent session expiration.

Further the general code flow for generating a new delegated session does not show any checks related to the nesting depth or a limitation of target devices.

Retest Status

The server implementation was adjusted to link the lifetime of a delegated session to the authentication time of the root session. This is achieved by recursively inheriting the `LoginTime` field of the parent session, as defined in `server/src/db/model/usersession.go`:

```
108 func (s *UserSession) NewDelegatedSession(device *Device, clientIP string)
(*UserSession, error) {
109    randomBytes := random.NBytes(32)
[...]
141    delegatedSession.LoginTime = s.LoginTime
142    currentTime := time.Now().UTC()
143    delegatedSession.DelegationTime = &currentTime
```

The `UnsafeSaveSession` function, which is called to store the delegated session in the respective cache afterwards, does now contain a check whether the session is allowed to be valid for the time-frame define as `UserSessionTTL`. It is refusing to store the session, in case its expiry would be outside a configured period (currently seven days) starting at the `LoginTime` field value.

The implementation can be found in the file `b5-main/server/src/access/unverifiedaccess.go`:

```
 701 func (acs *UnverifiedAccess) SessionCanLiveForAdditionalTTL(session
*model.UserSession, ttl int) bool {
 702    anticipatedEndTime := time.Now().Add(time.Duration(ttl) * time.Second)
 703    latestAcceptableStartTime := anticipatedEndTime.Add(-
constraints.UserSessionMaxAgeSeconds * time.Second)
 704
 705    return session.LoginTime.After(latestAcceptableStartTime)
 706 }
[...]
 728 func (acs *UnverifiedAccess) UnsafeSaveSession(session *model.UserSession)
error {
 729    ttl := cache.UserSessionUnverifiedTTL
 730
 731    if session.IsLoggedIn() {
```

```
 732       ttl = cache.UserSessionTTL
 733
 734       if !acs.SessionCanLiveForAdditionalTTL(session, ttl) {
 735          return errors.New("UnsafeSaveSession failed: session has reached
hard limit")
 736       }
 737    }
 738
 739    if err := acs.Cache.UnsafeSaveUserSession(session, ttl); err != nil {
 740       return err
 741    }
 742    return nil
 743 }
```

The issue was found to be mitigated by this code changes. Delegated sessions can no longer be used to infinitely create fresh sessions, thereby keeping the user authenticated arbitrarily long.

It could be noted that the current session length is still not aligned with the user's SSO settings. However, Agilebits explained that this is not one of their security goals. Rather, the limitations to the session duration are intended to apply only to **new** sign-ins and to preventing the use of biometric authentication after a certain amount of time has elapsed.

The issue is therefore considered to be fixed.

4.6 MacOS Desktop App not using Secure GUI Restore

Summary

Type	Location	Severity	Status
Code	MacOS Desktop App	High	Closed

Technical Description

MacOS offers a feature to automatically restore windows open on machine shutdown or for windows sent to the background in situations of resource shortages. For this purpose, the window data is serialized and stored in the file system. A flaw allowing for unsafe object deserialization was identified in this context that allows for code execution in the context of the attacked process[1].

The issue was addressed by Apple by restricting the classes for which objects will get deserialized. Since applications can implement their own serialization and restore routines for their UI elements, the protection measure cannot generally be applied. Instead, applications have to explicitly enable it by implementing the method `applicationSupportsSecureRestorableState` to return `true` in their `NSApplicationDelegate`.

Such a method implementation could not be found in the provided source code. It is therefore assumed that the MacOS Desktop client is prone to the serious attack.

Recommended Action

An implementation of the `applicationSupportsSecureRestorableState` returning `true` should be added to all MacOS applications. It must further be mentioned that this does not provide full protection against the named attack class. The fact that objects still get deserialized, despite limited to specific classes, might still lead to exploitable conditions - however, this greatly depends on implementation details. Therefore, the general recommendation to properly address such flaws is considering to refrain from deserializing objects at all. However, this could only be achieved by disabling the feature for the respective application.

Retest Status

The issue was addressed in a two-fold manner. On the one hand the main application and thereby the framework helper applications, which is based on Electron, was configured to depend on Electron version 21. By this update it makes use of Chromium in version not less than 106.0.5249.51[2], which includes adjustments to mitigate the risk of unsafe object deserializations on restoring the application UI. The performed changes were not audited in detail, since this would exceed the scope of the assessment.

1 https://sector7.computest.nl/post/2022-08-process-injection-breaking-all-macos-security-layers-with-a-single-vulnerability/
2 https://www.electronjs.org/releases/stable#21.0.0

Besides that, secondary macOS applications, as used for launching or updating the application, were modified to implement the `applicationSupportsSecureRestorableState` method as recommended.

It should be noted that a risk of exploits in the context of this mechanism could remain, since generic object deseralization is complex to secure thoroughly. Nevertheless, the issue was closed, since the applications follow current security best practices.

4.7 SSO Login Bypass by Session Extension

Summary

Type	Location	Severity	Status
Code/Design	Session Handling	Medium	Closed

Technical Description

As described in section 4.5, the repeatable prolonging of an active b5 session can be considered a bypass of the mandatory periodic SSO login option. Due to this, it was investigated whether the expiry of an existing session can be extended. While the function `TouchUserSession` (defined inside `b5-main/server/src/cache/usersession.go`), that serves exactly this purpose, seem to only be used on upgrading unverified to verified sessions, the function `SaveUserSession` that saves its argument to the session cache with a static expiry period is more broadly used. It is executed on calls to the function `MustSaveUserSession` (implemented in `b5-main/server/src/main/web/app.go`). This is used during issuing responses, in cases where the user's session has changed as indicated by the `acs.isChanged` flag.

As one example, this flag is set by the function `SetDeviceUUID` (inside `b5-main/server/src/access/unverifiedaccess.go`), which was found to be used on issuing a delegated session. It is called inside the function `registerDeviceForUserInCache` and `activateDeviceForUser`, which are called explicitly in the function `DelegateSession` or implicitly via the function `handleSigninSucceeded` (both defined in `server/src/logic/action/auth.go`).

Based on this, dynamic tests were performed to verify that the parent session is extended on requesting a delegated session. During this, it was noted that the parent session did frequently not expire after the expected time-frame of approx. 30 minutes after authentication, caused by other unknown conditions. For this reason, the planned test-case could not be performed, since the effect of the delegated session request on the session expiry could not be clearly differentiated from that of other requests. This, however, leads to the impression that the session expiry is updated regularly during the general use of the desktop application. This impression was confirmed by Agilebits stating that use-cases exist that prolong the session time-frame.

As noted above, the user's ability to arbitrarily extend the session validity can be considered a security issue, when periodic SSO authentication should be guaranteed.

Recommended Action

In order to address this issue, the session management procedures would need to be reworked to ensure that sessions are not prolonged whenever a user is required to perform a new SSO

authentication.

Reproduction Steps

In order to reproduce this issue, please proceed as described below. Set up a MacOS desktop application to make use of an intercepting proxy for HTTPS connections. Lock and unlock the account and note the time of the user authentication by searching for the recent requests towards `POST /api/{v1,v2,v3}/auth`. Now intercept the next request before those are submitted to the server and record it including all headers for later use. Drop the same and all following requests to ensure that those are not send to the server.

After the session should have expired, send the recorded request and observe that the server is responding with a `HTTP 401 Unauthorized` error message.

Repeat the process, but change some entries via another client after half of the session expiry time-frame. This should automatically trigger some requests sent by the intercepted client. Intercept those again and forward the first one or two to the server and record the next one without forwarding. Drop it and all further requests until the session should have expired. Send the recorded request and observe that a valid response is returned.

Check in both tests that the client does not issue any authentication related requests in between. Since the conditions on that the session is prolonged are not fully known, the second procedure might need to be repeated to obtain the desired result.

Retest Status

Both the `TouchUserSession` and `UnsafeSaveSession` functions, where the latter is internally called by `SaveUserSession`, were found to be modified. For sessions that are considered logged in, a check is performed that no session prolongation beyond a configured time-frame (currently seven days) will take place. For more details on the implemented routine, please refer to the reproduction steps of issue 4.5.

It could be noted that the current session length is still not aligned with the user's SSO settings. However, Agilebits explained that this is not one of their security goals. Rather, the limitations to the session duration are intended to apply only to **new** sign-ins and to preventing the use of biometric authentication after a certain amount of time has elapsed.

The issue is therefore considered to be fixed.

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 Android Password Field Allows Pasting HTML

It was noted that the android client (version 8.9.3) allowed to paste HTML into the password field upon login. When viewing the pasted password, the HTML was actually rendered. This could lead to self-XSS attack. Most software, e.g. Browsers, will apply some form of XSS sanitization to HTML pasted from the clipboard, but the past has shown that these sanitizers might be bypassed.[3] To reproduce this observation, please go to any website and copy some text with notable html content e.g., **bold** text and paste it into the password field. Note that it also suggests "Paste as plain text". Now please click on the eye symbol to view the rendered password.

5.2 Cursory Review of new SSO Design

During the engagement, Secfault Security discussed the findings described in sections 4.1 and 4.2 with Agilebits. The issues have been addressed by Agilebits and a new design has been provided. This new design has been subject to a cursory inspection by Secfault Security. The results of this inspection are described in this section.

Two major changes have been introduced in order to address the identified issues. The first change is that a client's `CredentialBundle` is now no longer encrypted using an asymmetric key. Instead, symmetric cryptography is used now. This prevents the backend from simply generating its own `CredentialBundle` and encrypting it with the client's key.

The second change is in the device enrollment process. The process has been changed, so that it now relies on using a PAKE protocol to confirm that both, the new and the existing device share the same one-time code. After the PAKE has been performed, both devices also share a strong cryptographic key, which is then used to establish a secure channel between the devices. Over this secure channel, the devices can now exchange the `CredentialBundle`.

Overall, the changes address the identified problems. It should be noted, however, that during the review of the new design, one potential issue has been identified:

The backend could still influence a client's behavior with respect to handling `CredentialBundles`; suppose a client has already been fully enrolled and this client now signs in on the backend. The backend could tell the client that it has no stored `CredentialBundle` (even though it actually does), and so the client would re-use its existing device key to encrypt its `CredentialBundle`. While this does not appear to be exploitable in the current design, it might still be advisable to let the client

3 See Copy & Pest by Mario Heiderich as example https://www.slideshare.net/x00mario/copypest

generate a fresh device key every time it uploads encrypted `CredentialBundles`. This issue has been found to be addressed properly during the review of the updated implementation performed in November 2022.

During the retest performed in November 2022, an update to the design document has been provided, which has been subject to review. The overall concept of relying on symmetric encryption for the `CredentialBundle` s, and of using CPace for negotiating a session key between the two devices during additional device enrollment is still in place. However, the key management process has been updated. The new scheme involves adding a "bundle version specifier" to the additional authenticated data of encrypted `CredentialBundle` s; this addresses possible problems with future changes in the format of the encrypted blob. Furthermore, the PAKE handshake now also includes the identifiers of both involved devices in its channel identifier.

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.

6.1.1 MITM Attack against Encrypted Credentials (Finding 4.1)

We were notified of this finding at the beginning of the test window and immediately started working on a fix. The finding has been retested by the vendor during a November pentest.

6.1.2 MITM Attack during Device Enrollment (Finding 4.2)

We were notified of this finding at the beginning of the test window and immediately started working on a fix. The finding has been retested by the vendor during a November pentest.

6.1.3 Lax Authorization Concept for Delegating Sessions (Finding 4.3)

We already acknowledge in the 1Password Security Design white paper that client side controls are the weakest control and are easily circumvented. This finding will not be fixed.

6.1.4 Automatic Restoring of Devices (Finding 4.4)

The design for delegated sessions required that devices that hadn't even been registered before are registered as part of the delegation. Given that a "malicious" device could just say it was a new device, not sure we gain much by adding a restriction here. This finding will not be fixed.

6.1.5 Prolonging Sessions by Nested Delegation (Finding 4.5)

We accepted this finding and implemented a fix that has since been retested by the vendor during a November pentest.

6.1.6 MacOS Desktop App not using Secure GUI Restore (Finding 4.6)

We accepted this finding and implemented a fix that has since been retested by the vendor during a November pentest.

6.1.7 SSO Login Bypass by Session Extension (Finding 4.7)

We accepted this finding and implemented a fix that has since been retested by the vendor during a November pentest.

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
API	Application Programming Interface
CLI	Command Line Interface
GUI	Graphical User Interface
HMAC	Keyed-Hash Message Authentication Code
HTML	Hypertext Markup Language
HTTPS	HTTP over SSL
IDP	Identity Provider
MITM	Man-In-The-Middle
PAKE	Password Authenticated Key Exchange
SSO	Single Sign-On
UUID	Universally Unique Identifier
XSS	Cross Site Scripting