Bitwarden Core App & Library Report

ISSUE SUMMARIES, IMPACT ANALYSIS, AND RESOLUTION

BITWARDEN, INC

Table of Contents

Summary

In August 2023, Bitwarden engaged with cybersecurity firm Cure53 to perform penetration testing and a dedicated audit of the Bitwarden core application and library. A team of two senior testers from Cure53 were tasked with preparing and executing the audit over two days to reach total coverage of the system under review.

Thirteen issues were discovered during the audit. Eleven issues were resolved post-assessment. One issue was determined not feasible to address. One issue is under planning and research.

This report was prepared by the Bitwarden team to cover the scope and impact of the issues found during the assessment and their resolution steps. For completeness and transparency, a copy of the report delivered by Cure53 has also been attached to this report.

Issue

BWN-08-007 WP1: Open redirect on SSO via returnUrl parameter (Low)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/server/pull/3696

The return URL had some susceptibility to undesired characters and paths being allowed. The URL has been sanitized if it includes spaces, tabs, etc. to prevent phishing attempts.

BWN-08-010 WP1: Admin XSS via name of provider organization (Medium)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/server/pull/3218

The organization name was being placed into alerts as a variable. This injection – and therefore the ability to send data into the alert – was removed to prevent HTML or anything else from being added.

BWN-08-015 WP1: HTML injection in Freshdesk support tickets (Low)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/server/pull/3219

Organization names could contain an expanded character set. Organization names are now sanitized and any HTML entities removed.

BWN-08-018 WP1: User-hijacking via confusion in Sustainsys RelayState (Critical)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/server/pull/3215

The maintainers of the vulnerable library were notified. As a temporary mitigation, a custom handler for the SSO callback was written that validated the `RelayState` parameter until the library was updated, and the Bitwarden reference then updated to use it.

BWN-08-003 WP5: Insufficient minimum parameters for KDF (Medium)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/clients/pull/6440

The minimum PBKDF2 iterations was updated to 600k. Previously, only a warning was shown to users with low KDF iterations; now users are prevented from changing their settings to anything less than 600k.

BWN-08-004 WP5: Insecure storage of access and refresh tokens (Medium)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/clients/pull/7975

Secure storage for desktop and mobile clients is utilized when available according to platform. This experience does differ depending on what the platform is capable of, but more secure storage capabilities now exist and continue to make it more difficult for rogue processes to capture contents on disk.

BWN-08-005 WP5: XXE declarations permitted on importing XML vaults (Info)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/clients/pull/6918

Logic was added to detect if external entities exist within an XML upload. An error message is displayed upon detection.

BWN-08-006 WP1: Access token in notification API URL (Info)

Status: Accepted as an upstream limitation.

Bitwarden must provide a SignalR authentication token in query strings for notifications over WebSockets to work as expected. This cannot be avoided and is required by SignalR, as noted in Microsoft documentation. Exposure is minimal and limited to basic information sent in the push payload.

BWN-08-008 WP1: Unmaintained IdentityServer4 dependency (Info)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/server/pull/3185

The platform was upgraded to use IdentityServer v6, and later v7. Both are supported and licensed.

BWN-08-009 WP1: Excessive lifetime of refresh tokens for IdentityServer4 (Low)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/server/pull/3697

Lifetimes of tokens were shortened. Furthermore, Bitwarden plans to implement improved refresh token usage in clients in the future.

BWN-08-012 WP1: HTML injection in passwordless login emails (Low)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/server/pull/3623

The return URL parameter of the passwordless admin login was encoded to prevent HTML injection.

BWN-08-016 WP1: DoS risk in attachments feature via max size (Info)

Status: Issue under planning and research.

Bitwarden is already streaming large file uploads directly and securely to online storage systems. For self-hosted installations, solutions are being assessed for improvements to the same streaming capabilities, including a chunk-based upload mechanism.

BWN-08-017 WP1: Leakage of encrypted private keys of organizations (Medium)

Status: Issue was fixed post-assessment.

Pull requests:

- https://github.com/bitwarden/server/pull/3195

Encrypted private keys were removed from the API payload. Logic was adjusted for this to no longer be needed.

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, each ticket has been given a unique identifier (e.g., *BWN-08-007*) to facilitate any future follow-up correspondence.

BWN-08-007 WP1: Open redirect on SSO via *returnUrl* parameter (*Low*)

During the examination of the server-side code, it was found that the *returnUrl* parameter is insufficiently validated in the SSO component. Using an arbitrary domain inside this parameter, followed by tricking a victim to rely on the seemingly trustworthy URL, ultimately leads to a redirect to an attacker-controlled domain. From that domain a phishing attempt could be started. As a prerequisite for exploitation, the attacker must know the scheme UUID of the SSO provider and the victim must be logged in to the IdP. Further, this issue is only exploitable if the victim uses the Chrome browser. The following URL illustrates the injection point of the malicious parameter.

Affected URL:
https://bitwarden.example.com/sso/Account/ExternalChallenge?scheme=0498a4f8-c173-44b6-85f9-b05700b2df21&returnUrl=%2F%09%2fexample.com%2Fconnect%2Fauthorize%2Fcallback

The code excerpt demonstrates the insufficient validation at the point in question. This issue is indirectly caused by the *IsLocalUrl* check returning *true* for the value */%09/example.com,* which the Chrome browser interprets as an absolute URL. To pass the *IsValidReturnUrl,* it is sufficient to append the path */connect/authorize/callback* to the URL.

Affected file:
server-2023.7.1/bitwarden_license/src/Sso/Controllers/AccountController.cs

Affected code:
```
if (!Url.IsLocalUrl(returnUrl) && !_interaction.IsValidReturnUrl(returnUrl))
   {
       throw new Exception(_i18nService.T("InvalidReturnUrl"));
   }
```

Since a proper URL should only contain URL-encoded whitespaces, it is recommended to include additional validation that fails if the user-supplied *returnUrl* parameter contains plain whitespaces.

BWN-08-010 WP1: Admin XSS via *name* of *provider organization* (*Medium*)

The Bitwarden admin backend utilizes Razor templates to render HTML code. By default, the rendering engine escapes HTML metacharacters to prevent the injection of malicious HTML code. However, since single-quotes are not escaped, user-input within the JavaScript context has to be handled with special care.

It was found that *organization* names are insecurely embedded in the single-quoted JavaScript context of the *Resend Setup Invite* feature, leading to a stored XSS vulnerability. However, this issue can only be exploited by malicious or compromised users of customer support, who could leverage it to hijack sessions of accounts equipped with higher privileges. What is more, this vulnerability is limited to the *cloud admin* portal, since the required feature is not available in self-hosted environments.

Steps to reproduce:

1. Log in to the internal *admin* portal with a user who has the *CS* role.
2. Navigate to *Providers* and select a *Reseller.*
3. Click on *New Organization* and place the XSS payload (e.g., *'-alert(1)'-*) in the *Organization Name* field.
4. Fill in the required fields and click on *Save.*
5. Lure a higher-privileged user into clicking on the *Resend Setup Invite* (blue envelope icon).
6. Observe that the XSS is triggered.

The following code excerpt illustrates that the *organization name* is embedded in between single-quotes in the *onclick* event handler of an *anchor* tag, which essentially allows escaping of the JavaScript string.

Affected file:
server-2023.7.1/src/Admin/Views/Providers/Organizations.cshtml

Affected code:
```
@if (org.Status == OrganizationStatusType.Pending)
  {
 <a href="#" class="float-right" onclick="return 
resendOwnerInvite('@org.OrganizationId', '@org.OrganizationName');">
       <i class="fa fa-envelope-o fa-lg" title="Resend Setup Invite"></i>
   </a>  }
```

It is recommended to properly escape the *organization name* for the JavaScript context. This could be done, for example, via the *HttpUtility.JavaScriptStringEncode*[1] function.

[1] https://learn.microsoft.com/en-us/dotnet/api/system.web.httputility.javascriptstringencode?view=net-7.0

BWN-08-015 WP1: HTML injection in Freshdesk support tickets *(Low)*

A source code review of the *server-master* repository revealed that the Bitwarden backend contained an endpoint for creating new Freshdesk support tickets. After discussing this with the Bitwarden team, it became clear that the Freshdesk application automatically creates a new support ticket on each new email to the *support@bitwarden.com* email address.

To that end, the Freshdesk application invokes an endpoint on the Bitwarden backend. This endpoint of the Bitwarden backend looks up the *organization* associated with the sender's email address within messages to *support@bitwarden.com* and constructs a HTML document involving the *organization name* without escaping or sanitizing this information.

As a result, an attacker who owns an *organization* on *vault.bitwarden.com* can use an *organization name* containing HTML tags. For instance, the attacker could craft a malicious name with a link, for example to attempt phishing attacks. However, dynamic testing determined that no XSS attack was feasible as a consequence of this flaw.

Steps to reproduce:

1. Create a new account on *vault.bitwarden.com*.
2. Activate premium subscription for *organizations*.
3. Create a new *organization* with the *name* indicated below.

   Organization name:
   `<a href="https://attacker.com">Click me</a>`

4. Create a new support ticket for Bitwarden by sending a message using the email address of the account at *vault.bitwarden.com*.
5. The operator at Bitwarden opening the ticket sees the link with the "*Click me*" message, in accordance with the payload in the *organization name* item.

Affected file:
server-master/src/Billing/Controllers/FreshdeskController.cs

Affected code:
```
public async Task<IActionResult> PostWebhook([FromQuery, Required] string key,
    [FromBody, Required] FreshdeskWebhookModel model)
{
    [...]
    try
    {
        [...]
        if (user != null)
        {
            [...]
            var orgs = await 
_organizationRepository.GetManyByUserIdAsync(user.Id);

            foreach (var org in orgs)
            {
                var orgNote = $"{org.Name} ({org.Seats.GetValueOrDefault()}): " 
+ 
$"{_globalSettings.BaseServiceUri.Admin}/organizations/edit/{org.Id}";
                note += $"<li>Org, {orgNote}</li>";
                [...]
            }
            [...]
            var noteBody = new Dictionary<string, object>
            {
                { "body", $"<ul>{note}</ul>" },
                { "private", true }
            };
            [...]
        }
        [...]
    }
    [...]
}
```

To mitigate this issue, Bitwarden should sanitize *organization names* fully, making sure these items are free from all HTML tags.

BWN-08-018 WP1: User-hijacking via confusion in Sustainsys *RelayState* (*Critical*)

The Single Sign-On logic can be tricked into accepting the SAML assertion for one *organization* of an identity provider belonging to a foreign *organization.* This effectively lets attackers with an *enterprise organization* authenticate as users of *organizations* that have Single Sign-On enabled. Moreover, these attackers would be able to perform arbitrary API calls. Thus, this issue was rated as *Critical*.

During the SSO login, Bitwarden instructs the Sustainsys2 SAML library to relay the targeted *organization* through the identity provider back to the login callback of the service provider. The library does so by associating the claim with a session cookie that has a unique name which is relayed through the SAML2 *RelayState* parameter. Upon receiving a successful authentication result, the library will determine the *organization* on the basis of the *RelayState* parameter, so as to log the user in.

However, the problem lies in the fact that the Sustainsys2 library insufficiently validates that the state identified by the *RelayState* parameter passed along a SAML assertion is associated with the authentication request. This means adversaries can swap the *RelayState* parameter with a SAML assertion, effectively getting the capacity to alter the authenticated *organization*.

Steps to reproduce:

1. Log in as an *organization* administrator.
2. Setup a valid Single Sign-On SAML configuration for the attacker's *organization* and add a user to the attacker's identity provider with the email address of the targeted Bitwarden victim.
3. Using a suitable HTTP proxy software like Burp Suite, prepare to intercept all HTTP requests containing either the parameter *SAMLRequest* or the parameter *SAMLResponse*[2]. As such, the software should be capable of decoding SAML traffic[3].
4. Start an SSO authentication flow for the victim:
5. Visit the *login* at the web vault URL.
6. Enter the victim's email address and proceed.
7. At the password prompt, click on *Enterprise single sign-on.*
8. Enter the identifier of the victim's *organization* and click on *Log in.*
9. An HTTP request to the victim's identity provider should be intercepted, like in the following.

HTTP request:

```
GET /app/trial-1883647_bwn08samlidp_1/exk6wadvczGldAIiZ697/sso/saml?
SAMLRequest=jJJBb9swDIXv
%2FRWG7rZkRbUdIQlgNNgQoBuGdtuhl0KRaFSoLHmi3HT79VPdblgOG3Yl
%2BfHhPXKDanR8kv2cHvwNfJsBU%2FE8Oo%2FytbMlc
%2FQyKLQovRoBZdLytv9wLXnF5BRDCjo48ifzb0QhQkw2eFIc9ltiDWsEY9wMrdatqHW3NgNT
vAPR8VVrgJPiK0TMwJZkPlOIMxw8JuVTLjG
%2BKllX1uIz45IJuWJ3pNhnG9artFAPKU0oKU3RKlfWXbdqRFuFx6QqHUaqpum8dX88eda9WL
Fmuq8pPD82J2We9I
%2F3zvQHe9esW4oY6MsIKfpfhq6Cx3mEeAvxyWr4cnP9qp2l83R1tOmkogFfuaCVk5d1x
%2Fmyg1PdmI5xEKVSoi7FsG7L4yCa8sguNatzHII1tNdIdhdFsVkYueQQd%2F
%2BnsKFn0O8tk%2FyYL3TYfwrO6u
%2FFuxBHlf5%2BwLqql4o15bCMShiVdb0xERBzFs6F01UElWBLUpyBFHR38SZ%2B
%2FmW7nwAAAP%2F%2FAwA%3D&RelayState=jVUg1u1svLVT-prh310dYXg5 HTTP/2
[...]
Host: trial-1883647.okta.com
```

[2] https://portswigger.net/burp/documentation/desktop/tutorials/using-interception-rules

[3] https://portswigger.net/bappstore/c61cfa893bb14db4b01775554f7b802e

10. Note both the *RelayState* parameter and the ID attribute of the decoded SAML authentication request before dropping the HTTP request.

    Decoded SAML request:

    ```
    <saml2p:AuthnRequest xmlns:saml2p="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
    ID="id064002df7cc741c89df0a28e48237de2" Version="2.0" [...]>
      <saml2:Issuer>http://sso.bitwarden.local:51822/saml2</saml2:Issuer>
      <saml2p:NameIDPolicy Format="urn:oasis:names:tc:SAML:1.1:nameid-
    format:emailAddress" AllowCreate="true" />
    </saml2p:AuthnRequest>
    ```

11. Use the same browser session to start an SSO authentication flow for the attacker's *organization.* This can be done by adding the Bitwarden SSO URL and the attacker's *organization* GUID within the following URL template. The URL should then be visited.

    URL template:

    ```
    <BitwardenSSOUrl>/Account/ExternalChallenge?scheme=<attackersOrgId>
    ```

12. An HTTP request to the attacker's identity provider should be intercepted. It should look similar to the HTTP request from *Step 5*. Replace the *RelayState* parameter and the *ID* attribute with the values from *Step 5* and forward the HTTP request to the attacker's identity provider.
13. The attacker should now be successfully authenticated as the victim-user, being prompted for the master-password in order to decrypt it. The authenticated attacker can perform any API actions in the name of the victim.

The following source code snippet shows that the Bitwarden account controller stores the scheme parameter - which is equivalent to the *organization* ID - to the authentication properties of the authentication challenge.

Affected file:
server/bitwarden_license/src/Sso/Controllers/AccountController.cs

Affected code:

```
        public IActionResult ExternalChallenge(string scheme, string returnUrl,
string state, string userIdentifier)
        {
        [...]
        var props = new AuthenticationProperties
        {
                RedirectUri = Url.Action(nameof(ExternalCallback)),
                Items =
```

```
        {
        // scheme will get serialized into `State` and returned back
        { "scheme", scheme },
        { "return_url", returnUrl },
        { "state", state },
        { "user_identifier", userIdentifier },
        }
    };

    return Challenge(props, scheme);
```

It must be noted that this behavior is present in the latest version of the Sustainsys2 library and, therefore, cannot be addressed by simply updating the library. It is advisable to hot-fix this issue by using the middleware class *SsoAuthenticationMiddleware* in order to identify the organization used during a successful SAML authentication instead of relying on the *RelayState* parameter.

The revised approach would prevent attackers from abusing this parameter to cause confusion. As a long-term solution, it is advisable to wait until the issue is reported by Cure53 and fixed by the maintainers of the Sustainsys2 library. Once this is done, the patched release should be deployed by Bitwarden.

Miscellaneous Issues

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

BWN-08-003 WP5: Insufficient minimum parameters for KDF *(Medium)*

Dynamic testing of the settings deployed for the encryption key of the Bitwarden web application revealed support for two KDF algorithms, namely *PBKDF2 SHA-256* and *argon2id*.

The former has been in focus of recent security breaches due to the increase of computing power, and recommendations regarding the minimum number of iterations have increased recently to *600.000*[4]. It must be noted that the KDF settings page provides the user with a warning: "*Higher KDF iterations can help protect your master password from being brute forced by an attacker. We recommend a value of 600,000 or more.*"

The latter KDF algorithm, namely *argon2id,* allows the user to adjust three values, specifically the *KDF iterations*, *KDF memory (MB)* and *KDF parallelism* parameters. These three parameters determine, in an abstract sense, the security that the *argon2id* KDF achieves. In the context of Bitwarden appt, it was found that the KDF settings could be reduced to insecure values by the user.

Testing confirmed that the minimum number of iterations for *PBKDF2 SHA-256* can be reduced to *5000,* whereas for the *argon2id* KDF the *KDF memory (MB)* value can be decreased to *16 MiB*. This is the case despite the officially recommended setting of 19 MiB for *iterations* set to 3 and 1 *degree of parallelism*.

Affected file:
clients-master/libs/common/src/platform/services/crypto.service.ts

Affected code:

```
async makeKey(
  password: string,
  salt: string,
  kdf: KdfType,
  kdfConfig: KdfConfig
): Promise<SymmetricCryptoKey> {
```

[4] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2

```
  let key: ArrayBuffer = null;
  if (kdf == null || kdf === KdfType.PBKDF2_SHA256) {
    if (kdfConfig.iterations == null) {
      kdfConfig.iterations = 5000;
    } else if (kdfConfig.iterations < 5000) {
      throw new Error("PBKDF2 iteration minimum is 5000.");
    }
    key = await this.cryptoFunctionService.pbkdf2(password, salt, "sha256", 
kdfConfig.iterations);
  } else if (kdf == KdfType.Argon2id) {
    if (kdfConfig.iterations == null) {
      kdfConfig.iterations = DEFAULT_ARGON2_ITERATIONS;
    } else if (kdfConfig.iterations < 2) {
      throw new Error("Argon2 iteration minimum is 2.");
    }

    if (kdfConfig.memory == null) {
      kdfConfig.memory = DEFAULT_ARGON2_MEMORY;
    } else if (kdfConfig.memory < 16) {
      throw new Error("Argon2 memory minimum is 16 MB");
    } else if (kdfConfig.memory > 1024) {
      throw new Error("Argon2 memory maximum is 1024 MB");
    }
    [...]
}
```

To mitigate this issue, Cure53 advises making it impossible for the user to provide insecure KDF settings. These values affect the resulting security of the vaults, so it is recommended to adjust the minimum values for KDF functions in a manner compliant with the officially recommended values[5].

BWN-08-004 WP5: Insecure storage of *access* and *refresh* tokens (*Medium*)

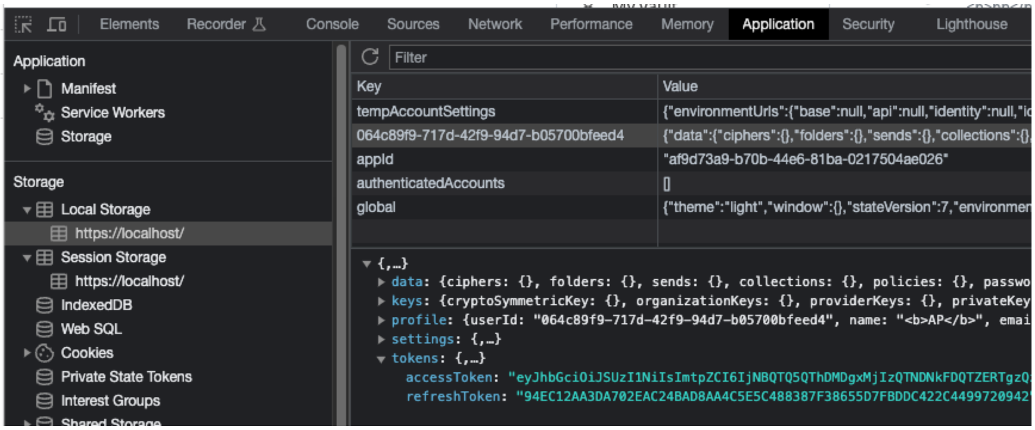
Dynamic tests of the Bitwarden web application revealed that the application persisted both *access* and *refresh* tokens of a user for the Bitwarden backend to the local storage and session storage of the browser. The web application utilizes these tokens to either authorize access to the Bitwarden backend, or to acquire new *access* tokens by using the *refresh* token.

Persisting sensitive information in the local storage of the browser is considered bad practice from a security standpoint, since it fosters extraction of tokens and similar items through XSS vulnerabilities in the affected application. For an attacker who manages to mount an XSS attack against a victim, it means easier extraction of *access* and *refresh* tokens. Consequently, the attacker can impersonate the victim with regard to the Bitwarden backend.

[5] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Steps to reproduce:

1. Open the Bitwarden web application and complete the login.
2. Open the developer tools of the browser and navigate to the *Application* section.
3. Open the *local storage* of the web application. It can be seen that the entry with the current user ID contains both *access* and *refresh* tokens of the victim, as demonstrated by the figure below.



Key	Value
tempAccountSettings	{"environmentUrls":{"base":null,"api":null,"identity":null,"id
064c89f9-717d-42f9-94d7-b05700bfeed4	{"data":{"ciphers":{},"folders":{},"sends":{},"collections":{},
appId	"af9d73a9-b70b-44e6-81ba-0217504ae026"
authenticatedAccounts	[]
global	{"theme":"light","window":{},"stateVersion":7,"environmen

```
▼ {,…}
  ▶ data: {ciphers: {}, folders: {}, sends: {}, collections: {}, policies: {}, passwo
  ▶ keys: {cryptoSymmetricKey: {}, organizationKeys: {}, providerKeys: {}, privateKey
  ▶ profile: {userId: "064c89f9-717d-42f9-94d7-b05700bfeed4", name: "<b>AP</b>", emai
  ▶ settings: {,…}
  ▼ tokens: {,…}
      accessToken: "eyJhbGciOiJSUzI1NiIsImtpZCI6IjNBQTQ5QThDMDgxMjIzQTNDNkFDQTZERTgzQ
      refreshToken: "94EC12AA3DA702EAC24BAD8AA4C5E5C488387F38655D7FBDDC422C4499720942
```

Fig.: The Bitwarden web application stores user-tokens in browser local storage

Affected file:

clients-master/libs/common/src/auth/services/token.service.ts

Affected code:

```
async setToken(token: string): Promise<void> {
  await this.stateService.setAccessToken(token);
}
[...]
async setRefreshToken(refreshToken: string): Promise<any> {
  return await this.stateService.setRefreshToken(refreshToken);
}
```

Cure53 advises keeping tokens in memory only, ceasing their persistence to the local storage. In case the current handling is absolutely necessary, the tokens should be encrypted with the user's password before being written to storage. Alternatively, tokens could also be sent in via cookies with appropriate security flags.

BWN-08-005 WP5: XXE declarations permitted on importing XML vaults *(Info)*

The Bitwarden web application allows its users to import vaults from other password/secret manager applications. Moreover, the app supports a large number of different vendors and their formats. The majority of such imports use either JSON or CSV, but some vendors utilize HTML or XML as underlying data formats.

The import done with XML format allows usage of XML external entities (XXE). Such entities pose an inherent security risk in server applications[6], as they can lead to local file disclosure, request forgery, Denial-of-Service situations or even RCE vulnerabilities.

Client-side XXE vulnerabilities are much less impactful due to the sandboxed nature of browsers. However, the exploitability heavily depends on the browser used. Past research nevertheless confirmed that such vulnerabilities also affected browsers[7]. Cure53 confirmed that the *DOMParser* class used by the tested app permitted processing of simple XXE payloads.

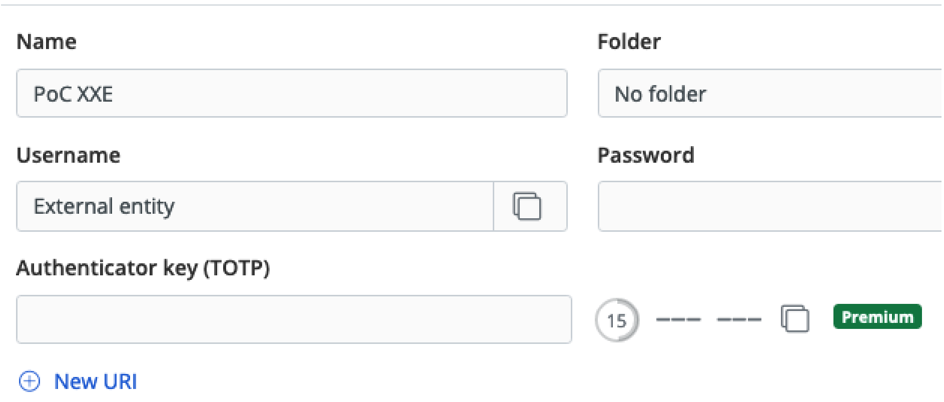
Steps to reproduce:

1. Open the Bitwarden web application and log in.
2. Navigate to the *Tools* section and open the *Import* page.
3. Select *Password Safe (xml)* as input-type.
4. Paste the payload shown below in the *text* field for the import and complete the import.

   XXE payload:
   ```
   <?xml version="1.0" encoding="ISO-8859-1"?>
   <!DOCTYPE replace [
     <!ELEMENT replace ANY>
     <!ENTITY xxe "External entity">
   ]>
   <passwordsafe delimiter=";">
   <entry><title>PoC XXE</title><username>&xxe;</username></entry>
   </passwordsafe>
   ```

5. Navigate to the vault of the user. Open the new *login entry* called *PoC XXE*. The *Username* field contains the external entity's value, as demonstrated in the figure below.

[6] https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html
[7] https://www.blackhat.com/docs/us-15/materials/us-15-Wang-FileCry-The-New-Age-Of-XXE.pdf



Name: PoC XXE

Folder: No folder

Username: External entity

Password:

Authenticator key (TOTP):

Fig.: XXE payload processed successfully by the web application.

Affected file:
clients-master/libs/importer/src/importers/base-importer.ts

Affected code:

```
protected parseXml(data: string): Document {
  const parser = new DOMParser();
  const doc = parser.parseFromString(data, "application/xml");
  return doc != null && doc.querySelector("parsererror") == null ? doc : null;
}
```

To mitigate this issue, Cure53 recommends disabling external entities unless they are absolutely required. The NPM package *libxmljs* appears to be a viable alternative to XML parsing[8].

BWN-08-006 WP1: *Access* token in notification API URL (*Info*)

After successful login, the Bitwarden web application sends a request to the */notifications/hub* endpoint of the Bitwarden backend. This request corresponds to a *GET* request and includes an HTTP parameter containing the *access* token of a user. Providing secret information, for instance *access* tokens or other authorization credentials through HTTP query parameters, is considered insecure, since many web servers, proxies or similar components log URLs of requests by default.

In case there are any of such components between the victim's browser and the Bitwarden backend, and an attacker has access to them, the sensitive information from within URLs could be leaked to the attacker. Such information could help adversaries in mounting further, more sophisticated attacks, generally aiming at impersonation of the victim.

[8] https://www.npmjs.com/package/libxmljs

Steps to reproduce:

1. Open an interception proxy like Burp and make sure all traffic between the browser and the Bitwarden backend is recorded by the interception proxy.
2. Open the Bitwarden *login* web page and authenticate with an arbitrary account known to the Bitwarden backend.
3. Observe the action triggering a request similar to the one shown below within Burp.

   Request:
   ```
   GET /notifications/hub?access_token=eyJhbGc<REDACTED>oXqI HTTP/2
   Host: localhost
   [...]
   ```

4. From the URL path, it can be seen that the Bitwarden web application sends an *access* token through the HTTP query parameter called *access_token* to the Bitwarden backend.

To mitigate, sensitive information like *access* tokens, passwords or other credentials should exclusively be sent within request bodies or within HTTP headers.

BWN-08-008 WP1: Unmaintained *IdentityServer4* dependency (*Info*)

While reviewing the dependencies utilized by the Bitwarden server component, it was found that the *IdentityServer4* library was no longer being maintained. The official documentation states that the library was only checked and equipped with security updates until November 2022[9]. This introduces the risk of potential future vulnerabilities persisting in the Bitwarden application without any security patches available.

It is recommended to utilize the successor library, namely *Duende IdentityServer*[10], to ensure that the application is provided with security patches in the future.

[9] https://identityserver4.readthedocs.io/en/latest/index.html
[10] https://duendesoftware.com/products/identityserver

BWN-08-009 WP1: Excessive lifetime of *refresh* tokens for *IdentityServer4* (*Low*)

In the Bitwarden backend application, it was noted that *access* and *refresh* tokens were used to authorize requests after a successful login by a user. The backend relies on the *IdentityServer4* solution for handling tokens and their generation (see also issue BWN-08-008).

The Bitwarden web application sets the HTTP *Authorization* header for each request, making sure it contains the currently valid *access* token of the user. The *refresh* token is used to fetch new *access* tokens for a user in case the original *access* token expires.

It was found that *access* tokens expire after one hour, whereas the lifetime of a *refresh* token depends on the application of the user. For example, for mobile applications the refresh token expires after 90 days, a cutoff date of 30 days is used for the web application. Besides a long lifetime, *refresh* tokens can be replayed multiple times to acquire new *access* tokens.

As a result of the token settings, an attacker who has a *refresh* token of a user can impersonate the victim for a rather long time. The attacker can use the *refresh* token multiple times to acquire new *access* tokens upon their expiration, without being asked to reauthenticate.

Affected file #1:
server-master/src/Identity/IdentityServer/StaticClientStore.cs

Affected code:
```
public StaticClientStore(GlobalSettings globalSettings)
{
    ApiClients = new List<Client>
    {
        new ApiClient(globalSettings, BitwardenClient.Mobile, 90, 1),
        new ApiClient(globalSettings, BitwardenClient.Web, 30, 1),
        new ApiClient(globalSettings, BitwardenClient.Browser, 30, 1),
        new ApiClient(globalSettings, BitwardenClient.Desktop, 30, 1),
        new ApiClient(globalSettings, BitwardenClient.Cli, 30, 1),
        new ApiClient(globalSettings, BitwardenClient.DirectoryConnector, 30,
24)
    }.ToDictionary(c => c.ClientId);
}
```

Affected file #2:
server-master/src/Identity/IdentityServer/ApiClient.cs

Affected code:

```
public ApiClient(
    GlobalSettings globalSettings,
    string id,
    int refreshTokenSlidingDays,
    int accessTokenLifetimeHours,
    string[] scopes = null)
{
    [...]
    RefreshTokenUsage = TokenUsage.ReUse;
    SlidingRefreshTokenLifetime = 86400 * refreshTokenSlidingDays;
    AbsoluteRefreshTokenLifetime = 0; // forever
    [...]
}
```

To mitigate this issue Cure53 advises revisiting the lifetime of *refresh* tokens in the utilized *IdentityServer4* server. Setting the *RefreshTokenUsage* parameter to *TokenUsage.OneTimeOnly*[11] is advised.

BWN-08-012 WP1: HTML injection in passwordless login emails *(Low)*

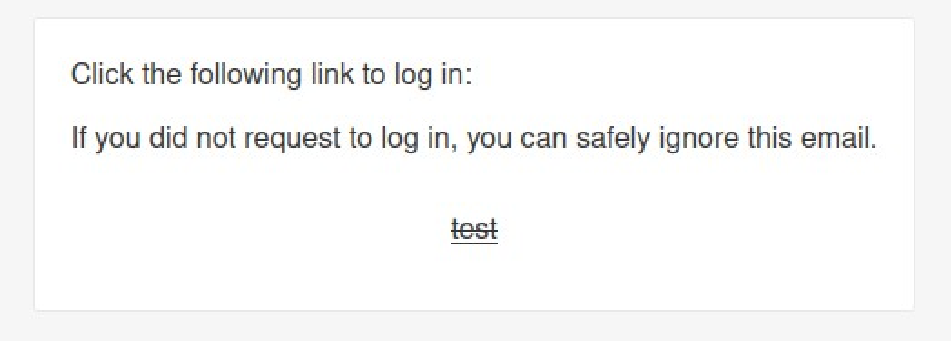
While testing the application for HTML injections in emails, it was discovered that the *ReturnUrl* parameter of the passwordless admin login was not being properly encoded. As a result, HTML code can be injected into login emails. However, since the injection is limited to certain characters, it was not possible to leak the *login* token via an additional *anchor* or *image* tag. Therefore, this issue was rated as *Low* in terms of impact.

The following PoC request can be used to reproduce this issue. After submitting the request, it can be observed that the injected HTML code is rendered in the resulting email.

PoC request:

```
POST /admin/login HTTP/2
Host: bitwarden.example.com
Content-Type: application/x-www-form-urlencoded
[...]
ReturnUrl="><s>test<style>&Email=johannes@cure53.de&__RequestVerificationToken=
[...]
```

[11] https://identityserver4.readthedocs.io/en/latest/topics/refresh_tokens.html#additional-client-settings



Click the following link to log in:

If you did not request to log in, you can safely ignore this email.

~~test~~

Fig.: Rendered email with HTML code injected.

The affected parameter is being processed, as illustrated in the code excerpt below. The code exemplifies that the URL parameters are added to the *queryCollection* which properly URL-encodes the parameters. However, when building the final URL, the *Uri* constructor removes the URL-encoding of certain characters, including HTML metacharacters such as *"<>*. The final URL is then embedded in the email template without further encoding.

Affected file:
src/Core/Utilities/CoreHelpers.cs

Affected code:

```
var queryCollection = HttpUtility.ParseQueryString(queryString);
foreach (var kvp in values ?? new Dictionary<string, string>())
{
       queryCollection[kvp.Key] = kvp.Value;
}

var uriKind = uri.IsAbsoluteUri ? UriKind.Absolute : UriKind.Relative;
if (queryCollection.Count == 0)
{
       return new Uri(baseUri, uriKind);
}
return new Uri(string.Format("{0}?{1}", baseUri, queryCollection), uriKind);
```

It is recommended to ensure that all HTML metacharacters are properly URL-encoded in the resulting URL. One possible solution could be to remove the *Uri* constructor, since it is responsible for unnecessarily removing the encoding process for certain characters.

BWN-08-016 WP1: DoS risk in *attachments* feature via max size (*Info*)

During a source code review of the *server-master* repository, it was found that vault entries in Bitwarden permit file attachments. The maximum file size of an attachment is set to 500MB. The Bitwarden backend writes the entire file content through a stream on the request's body to a file for self-hosted backends in the API service. On downloading such an attachment, the backend responds with a URL from which the client then ultimately downloads the file. The server responsible for providing the attachment on downloading corresponds to a static file server, referred to as attachment service.

Given the large maximum size for attachments, it is reasonable to assume that an attacker could bring the API service down on upload. Same goes for the attachment service which - upon download - could encounter out-of-memory situations caused by running multiple uploads or downloads with files having a size close to the maximum. Dynamic testing in a local environment confirmed that the memory consumption of the Docker engine running the backend increases tremendously in such situations.

Affected file:
server-master/src/Api/Vault/Controllers/CiphersController.cs

Affected code:

```
[HttpPost("{id}/attachment/{attachmentId}")]
[SelfHosted(SelfHostedOnly = true)]
[RequestSizeLimit(Constants.FileSize501mb)]
[DisableFormValueModelBinding]
public async Task PostFileForExistingAttachment(string id, string attachmentId)
{
    if (!Request?.ContentType.Contains("multipart/") ?? true)
    {
        throw new BadRequestException("Invalid content.");
    }
    [...]
    await Request.GetFileAsync(async (stream) =>
    {
        await _cipherService.UploadFileForExistingAttachmentAsync(stream,
cipher, attachmentData);
    });
}
```

To mitigate this issue, Cure53 advises revisiting the maximum file size available for the attachments. A chunk-wise transfer approach should be implemented for both the upload and download processes concerning attachments.

BWN-08-017 WP1: Leakage of encrypted private keys of *organizations* (*Medium*)

A source code review of the *server-master* repository points to a flaw in the controller responsible for serving requests concerning *organizations.* This controller contains an endpoint for querying the keypair of an *organization*. The keypair essentially gets used in the *admin password reset* feature for enterprise *organizations*.

To protect the organization's private key, Bitwarden applies a symmetric cipher using a symmetric *organization* key to the private key of the *organization*. This symmetric key gets encrypted in a user-specific way for each user of the *organization*. The endpoint for reaching the encrypted private key of an *organization* fails to apply proper access control. Moreover, it lets every user of the Bitwarden backend acquire the encrypted private key of an *organization*.

An attacker who has the *organization* ID can download the encrypted private key of that *organization*. In case the attacker was a former member of the *organization*, they would know the symmetric organization key. As such, they could decrypt the private key of the organization, even after rotation. However, it was not possible to use these circumstances to mount a successful attack in the context of this project. Therefore, this issue can be considered as a weakness rather than a vulnerability.

Steps to reproduce:

1. Open an interception proxy like Burp and make sure all traffic between the browser and the Bitwarden backend is recorded by the interception proxy.
2. Create a new account that is not associated with any organization.
3. With that account, log in to the Bitwarden web application.
4. Pick any *GET* request that includes an *authorization* header to the Bitwarden backend from Burp’s history and send it to Burp’s *repeater* functionality.
5. Modify the request as shown below.

   Request:
   ```
   GET /api/organizations/118fcb55-5e0c-4759-bd1a-b058010687ed/keys  HTTP/2
   Host: localhost
   [...]
   Authorization: Bearer eyJhb<REDACTED>GJoy4
   [...]
   ```

6. The organization with the ID *118fcb55-5e0c-4759-bd1a-b058010687ed* was chosen for these reproduction steps. Sending the request to the backend reveals the encrypted private key in its response, as demonstrated by the excerpt shown below.

Response:

```
HTTP/2 200 OK
Server: nginx
[...]

{"publicKey":"MIIB[...]","privateKey":"2.2/
AIjDq[...]","object":"organizationKeys"}
```

Affected file:

server-master/src/Api/Controllers/OrganizationsController.cs

Affected code:

```
[HttpGet("{id}/keys")]
public async Task<OrganizationKeysResponseModel> GetKeys(string id)
{
    var org = await _organizationRepository.GetByIdAsync(new Guid(id));
    if (org == null)
    {
        throw new NotFoundException();
    }

    return new OrganizationKeysResponseModel(org);
}
```

To mitigate this issue, Cure53 recommends deploying a check as to whether the caller of the *{id}/keys* endpoint in the *OrganizationsController* is a member of the *organization* or not. If they cannot be confirmed as members, the request should be denied.