HACKMANIT

Open Penetration Test Report: *KeeWeb*

Version: 1.0.1
06.05.2020

Dr. Marcus Niemietz
Phone: (+49)(0)234 / 45930961 | E-Mail: marcus.niemietz@hackmanit.de

Project Information

Application:	KeeWeb v1.12.3 (9b07bbd5, 2019-11-06) (`https://keeweb.info`)
Environment:	web, native
Developer:	Dimitri Witkowski (Antelle)
Project leader:	Hackmanit GmbH Universitätsstraße 150 44801 Bochum, Germany
Project executive:	Dr. Marcus Niemietz Phone: (+49)(0)234 / 45930961 Fax: (+49)(0)234 / 45930960 E-Mail: marcus.niemietz@hackmanit.de
Project members:	Dr. Christian Mainka (Hackmanit GmbH) Karsten Meyer zu Selhausen (Hackmanit GmbH) Prof. Dr. Juraj Somorovsky (Hackmanit GmbH)
Project period:	2020-03-16 – 2020-04-03
Version of the report:	1.0.1

This report was technically verified by Dr. Christian Mainka.
This report was linguistically verified by David Herring.

Contents

1 Summary

Hackmanit GmbH has tested the security of the web application and native application KeeWeb within 10 working days. KeeWeb was tested in version 1.12.3. The main focus of this security audit was to detect JavaScript based code injection and risks in the authorization process with third-party cloud storage providers.

Weaknesses. During the penetration test, three weaknesses classified as *High* and three further weaknesses classified as *Medium* were identified. The highest ranked weakness targeted the OAuth grant used when a user wants to grant KeeWeb access to their cloud storage. The used OAuth implicit grant exposes the access token to a substantial attack surface including the browser history. A stolen access token allows an attacker to access arbitrary files stored in the user's cloud storage. The other two weaknesses classified as *High* allow the execution of malicious JavaScript code in the user's browser. This allows an attacker to access the opened password database of the victim and steal their credentials. The other weaknesses generally increase the attack surface for the access token and the user's credentials to the utilized cloud storage provider. Additionally, one weakness might allow an attacker to inject their own access token in the user's browser and make the victim use the attacker's cloud storage for additional actions.

We recommended fixing all identified weaknesses to prevent attacks, as well as implementing the additional recommendations to further increase the security of KeeWeb to Mr. Witkowski, who is the main contributor of KeeWeb.

Mr. Witkowski followed our recommendations and released KeeWeb version v1.14.2 on 2020-05-04. We can confirm that v1.14.2 successfully fixes all identified weaknesses and implements our additional recommendations.

Top Weaknesses:

Risk Level	Finding	Reference
H01	Use of the Deprecated OAuth Implicit Grant	Section 6.1, page 8
H02	XSS via Form Fields	Section 6.2, page 10
H03	XSS via a Pseudo-Protocol Definition	Section 6.3, page 12

Structure. The report is structured as follows: In Section 2, the timeline of the penetration test is listed. Section 3 introduces our methodology, and Section 4 explains the general conditions and scope of the penetration test. Section 5 provides an overview of the identified weaknesses and further recommendations. In Section 6, all identified weaknesses are discussed in detail and specific countermeasures are described. Section 7 summarizes our recommendations resulting from observations of the application.

2 Project Timeline

The penetration test was performed remotely from Bochum within the time frame starting at 2020-03-16 until 2020-04-03. In total, Hackmanit invested 10 working days to test the application and write this report.

The weaknesses identified during the penetration test were responsibly disclosed to Mr. Witkowski as the main contributor of the KeeWeb project on 2020-04-16. Mr. Witkowski started to fix the weaknesses immediately and released a first update (v1.14) for KeeWeb on 2020-04-18.

Hackmanit conducted a short retest of the identified weaknesses and informed Mr. Witkowski that all weaknesses except M03 *(Missing mix-up attack protection)* were successfully fixed on 2020-04-30. Mr. Witkowski released a second update (v1.14.2) on 2020-05-04 fixing the last weakness. Mr. Witkowski also implemented both recommendations listed in Section 7.

3 Methodology

Among others, the following tools were used for the penetration test:

Tool	Link
Mozilla Firefox	`https://www.mozilla.org/de/firefox/`
Google Chrome	`https://www.google.com/intl/de_ALL/chrome/`
Internet Explorer	-
Safari	-
Burp Suite Professional	`https://portswigger.net/burp`
Self-developed tools	-

Risk Rating. Each weakness has its own CVSS 3 base score rating (*Common Vulnerability Scoring System Version 3 Calculator*).[1,2] Based on the CVSS 3 base score, the following weaknesses assessment is performed:

0.0 – 3.9:	Low
4.0 – 6.9:	Medium
7.0 – 8.9:	High
9.0 – 10.0:	Critical

[1] `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`
[2] `http://www.first.org/cvss/cvss-guide`

4 General Conditions and Scope

Hackmanit announced in September 2019 that Hackmanit offers a pro bono penetration test program for web and single sign-on (SSO) applications.[3] In a half-year cycle, starting in January 2020, Hackmanit started to offer a free remote penetration test to support non-commercial organizations.

Hackmanit is pleased that Mr. Witkowski, who is the main contributor of the project KeeWeb,[4] sent a request to test the application. KeeWeb is a free cross-platform password manager compatible with KeePass. Furthermore, it is an implementation of a password manager running inside a browser. It does not require any server and aims to work with sensitive data, such as user passwords. Hackmanit has investigated parts of the implementation with a main focus on JavaScript based code injection and risks within the authorization process with third-party cloud storage providers.

Information about the tested application:

- KeeWeb v1.12.3 (9b07bbd5, 2019-11-06)
- Environment: web, native

[3] `https://www.hackmanit.de/de/blog/80-pro-bono-penetration-test`
[4] `https://keeweb.info`

5 Overview of Weaknesses and Recommendations

Risk Level	Finding	Reference
H01	**Use of the Deprecated OAuth Implicit Grant:** The application uses the implicit grant to gain access tokens from different authorization servers. According to the OAuth 2.0 best current practices, the implicit grant should not be used anymore.	Section 6.1, page 8
H02	**XSS via Form Fields:** HTML form fields can be used to inject JavaScript code into the application.	Section 6.2, page 10
H03	**XSS via a Pseudo-Protocol Definition:** The pseudo-protocol `javascript:` allows an attacker to specify JavaScript code, which will be executed after a click event.	Section 6.3, page 12
M01	**Use of an Embedded User-Agent for User Authentication:** The native application uses an embedded user-agent (UA) to prompt the user for authentication when accessing a cloud storage. According to the RFC "OAuth 2.0 for Native Apps", native applications must not use embedded UAs but instead the system's default browser.	Section 6.4, page 14
M02	**Possible Access Token Injection with postMessage() API:** When the application receives messages sent using the `postMessage()` API, it does not validate the sender. This allows an attacker to send arbitrary messages including their own access token to the application.	Section 6.5, page 15
M03	**Violation of the OAuth 2.0 Best Current Practices:** The application makes use of the OAuth 2.0 authorization framework without following the security recommendations from the OAuth working group. The overall security of the application increases if the best current practices are implemented correctly.	Section 6.6, page 17

R01	**Delivering the Content Security Policy:** Some content injection risks can be minimized by delivering the content security policy (CSP) within a `meta` element.	Section 7.1, page 19
R02	**Implementing a Logout Option:** A logout option allows a user to revoke the access to the supported cloud storage providers and reduces the risk of access token leakage.	Section 7.2, page 19

Definitions:

Critical Risk	Weaknesses classified as *Critical* can be exploited with very little effort by an attacker. They have very large negative effects on the tested system, its users and data, or the system environment.
High Risk	Weaknesses classified as *High* can be exploited with little effort by an attacker. They have a major negative impact on the tested system, its users and data, or the system environment.
Medium Risk	Weaknesses classified as *Medium* can be exploited with medium effort by an attacker. They have a medium negative impact on the tested system, its users and data, or the system environment.
Low Risk	Weaknesses classified as *Low* can only be exploited with great effort by an attacker. They have little negative impact on the tested system, its users and data, or the system environment.
Information	Observations classified as *Information* are usually no weaknesses. Examples of these observations are unusual configurations and possibly unwanted behavior of the tested system.
Recommendation	*Recommendation* identifies measures that may increase the security of the tested system. Implementation is recommended, but not necessarily required.

6 Weaknesses

In the following sections, we list the identified weaknesses. Every weakness has an identification name which can be used as a reference in the event of questions, or during the patching phase.

6.1 H01 Use of the Deprecated OAuth Implicit Grant

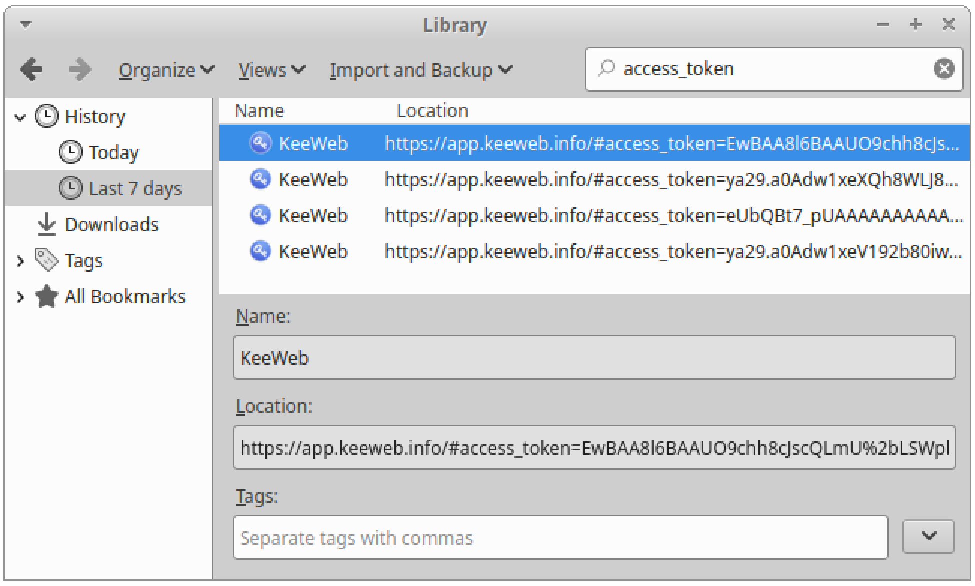
Exploitability Metrics		**Impact Metrics**	
Attack Vector (AV)	**Local**	Confidentiality Impact (C)	**High**
Attack Complexity (AC)	**Low**	Integrity Impact (I)	**High**
Privileges Required (PR)	**None**	Availability Impact (A)	**High**
User Interaction (UI)	**Required**	Scope (S)	**Changed**
Subscore: **1.8**		Subscore: **6.0**	
Overall CVSS Score for H01: 8.6			

General Description. OAuth supports multiple authorization grants for applications to gain access tokens from the authorization server. The "OAuth Authorization Framework" [2] initially defined four grants with specific scenarios for each grant in mind. The implicit grant was specified as a workaround for so-called single-page applications (SPAs) – applications running entirely in the web browser of the user. SPAs were not able to use the default grant – the authorization code grant – because the same-origin policy (SOP) prevented these applications from issuing a request directly to the authorization server.

The implicit grant has multiple security drawbacks in comparison to the authorization code grant. One of the major problems of the implicit grant is the delivery of the access token through the front-channel when the user's browser is redirected from the authorization server to the SPA. The presence of the access token in the URL significantly increases its attack surface and exposes it to the browser history, for example.

Since the initial release of the OAuth authorization framework, browsers have been extended with a feature called cross-origin resource sharing (CORS). CORS allows SPAs to access the authorization server directly like any other OAuth client. This development makes the implicit grant obsolete because SPAs can now use the authorization code grant instead. The avoidance of the implicit grant is strongly recommended. Further details on its weaknesses can be found in the Drafts "OAuth 2.0 Security Best Current Practice" [3, 2.1.2] and "OAuth 2.0 for Browser-Based Applications" [1, 4], as well as the RFC "OAuth 2.0 for Native Apps" [5, 8.2]. These documents instead recommend using the authorization code grant in conjunction with the Proof Key for Code Exchange (PKCE) [4] extension.

Weakness. The application uses the implicit grant (`response_type=token`) when obtaining an access token from any of the supported authorization servers (e.g., Dropbox, Google,



access_token

Name	Location
KeeWeb	https://app.keeweb.info/#access_token=EwBAA8l6BAAUO9chh8cJs...
KeeWeb	https://app.keeweb.info/#access_token=ya29.a0Adw1xeXQh8WLJ8...
KeeWeb	https://app.keeweb.info/#access_token=eUbQBt7_pUAAAAAAAAAA...
KeeWeb	https://app.keeweb.info/#access_token=ya29.a0Adw1xeV192b80iw...

Name: KeeWeb

Location: https://app.keeweb.info/#access_token=EwBAA8l6BAAUO9chh8cJscQLmU%2bLSWpl

Tags: Separate tags with commas

Figure 1: Access tokens stored in the browser history.

or Microsoft). Using the implicit grant exposes the access token to the browser history (see Figure 1). This dramatically increases the attack surface of the access token. If an attacker is able to steal the access token, the attacker can use it to access the cloud storage in the name of the victim and both read and write any file in the storage.[5] This is especially critical in scenarios which involve public or shared computers. A victim might want to use KeeWeb to access his or her password database and open the database from a cloud storage. After the victim finishes using the computer, the victim will close the database but might not delete the browser history. This allows other users to easily access the access token and use it to access the victim's cloud storage. It is important to note that the access token gives the attacker access to *all* files stored in the victim's cloud storage account and not only the password database. The database itself should be secured by a password which the attacker does not obtain using this attack.

Countermeasures. We recommend using the authorization code grant with the PKCE extension instead of the implicit grant, if possible. In order to use the authorization code grant and PKCE, the authorization server must support CORS and PKCE.

Retest. We can confirm that this weakness was successfully fixed in KeeWeb version v1.14 released on 2020-04-18. The application uses the authorization code grant for all cloud storage providers and the PKCE extension if supported.

[5]Depending on the specific cloud storage provider, the access token might only be usable to access a certain folder instead of the entire cloud storage of the victim.

6.2 H02 XSS via Form Fields

Exploitability Metrics		Impact Metrics	
Attack Vector (AV)	**Network**	Confidentiality Impact (C)	**High**
Attack Complexity (AC)	**Low**	Integrity Impact (I)	**High**
Privileges Required (PR)	**Low**	Availability Impact (A)	**Low**
User Interaction (UI)	**Required**	Scope (S)	**Unchanged**
Subscore: **2.1**		Subscore: **5.5**	

Overall CVSS Score for H02: 7.6

General Description. Cross-site scripting (XSS) allows the injection of client-side scripts into web applications that are viewed by other users. Usually, a victim will click on a link which is defined by the attacker. Afterwards, malicious code, like JavaScript, will be executed in the victim's browser due to a code injection vulnerability. This leads to an exposure of sensitive data or a manipulation of the web page.

Weaknesses. We detected different Self-XSS weaknesses in the form field elements. These weaknesses can be used to execute malicious JavaScript code in the user context. Each of these weaknesses can be exploited in two ways: First, the user copies malicious code into the form field and triggers it automatically or with the help of a trusted event (e.g., mouseover). Second, and more importantly, each weakness can be triggered by reading a malicious XML password file, which can be provided by an attacker.

Weakness 1/3: Entry Notes. The form field *notes* can be used to inject JavaScript code and therefore execute XSS attacks. To inject malicious code, as displayed in Figure 2, the user has to create a note within an *entry* that includes a payload such as:
`<script>alert(document.domain)</script>`.

Moreover, an attacker could use such an injection point in combination with a **file upload**. As displayed in Listing 1, the attacker could send the victim a malicious file including JavaScript code which will be directly executed when the file is uploaded by the victim; this allows an attacker to gain access to the complete DOM, and therefore attain the victim's credentials in all opened databases.

Weakness 2/3: Group and Database Name. The form field *name* within the database settings to define the database name (id `grp__field-title`), and also the form field to define the group name (id `grp__field-title`), can be used to inject JavaScript code. To execute JavaScript code, like `<script>alert(document.domain)</script>`, the user has to create a new *entry*. Afterwards, the user has to do a *mouseover* event over the group field, so that the payload will be executed.

The vulnerability is identified in the `data-title` attribute within the *div* element that contains the group name. In contrast to the content of the *div* element, the content of the `data-title` attribute is not encoded with HTML entities.

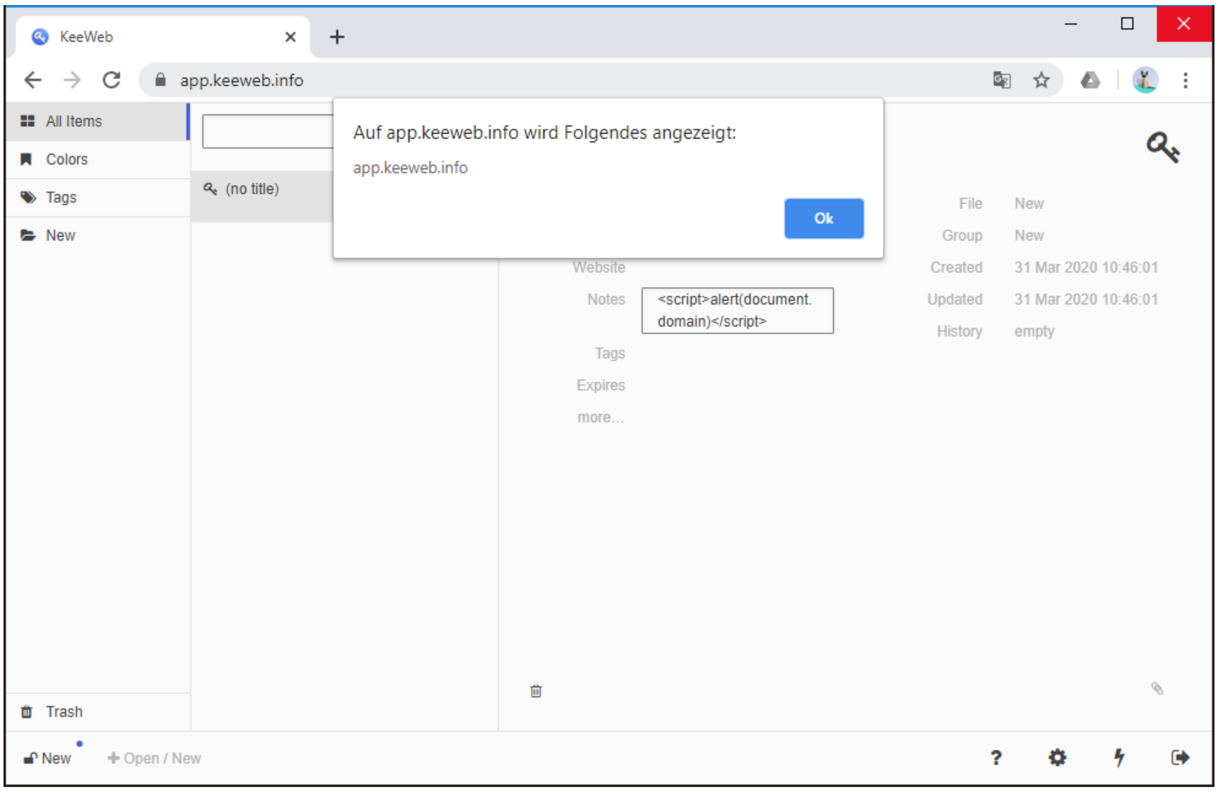


Figure 2: JavaScript code which was injected into the *notes* HTML form field will be directly executed in the application.

To summarize, we detected two injection points for JavaScript code:

1. By clicking on the gear next to a group name, the attacker could define a malicious group name like `<script>alert(document.domain)</script>` within the form field with the *id* `grp__field-title`.

2. In case that a group is not defined, the database name will be used as a placeholder for the group name. By navigating to general settings, the database name can be defined under *Name* within the form field with the *id* `settings__file-name`.

Weakness 3/3: Template Name. KeeWeb allows to specify template elements. However, in contrast to the name of new entries, the name of newly created templates could be used to inject JavaScript code.

To inject JavaScript code, the user has to insert a payload, like `<svg onload=alert(document.domain)>`, into the *title* of the template with the class name `details__header-title`.[6] Afterwards, the user has to explicitly click on the plus symbol to execute the attack vector; the defined JavaScript code will be executed because

[6]Please note that the given payload will be executed in Firefox but – in contrast to the other payloads – not be executed in Google Chrome.

```
...
        <String>
          <Key>Notes</Key>
          <Value>&lt;script&gt;alert(document.domain)&lt;/script&gt;</Value>
        </String>
        <AutoType>
          <Enabled>True</Enabled>
          <DataTransferObfuscation>0</DataTransferObfuscation>
        </AutoType>
        <History/>
      </Entry>
    </Group>
    <DeletedObjects/>
  </Root>
</KeePassFile>
```

Listing 1: JavaScript code within an XML file, which could be used as a storage file for KeeWeb.

newly registered templates will be automatically shown within the drop-down list if new database entries are created.

Countermeasures. We recommend not allowing the reflection of user input, like HTML and especially JavaScript code, for example, with the help of form fields. In addition, special characters like the greater-than sign should be replaced with HTML entities.

Retest. We can confirm that this weakness was successfully fixed in KeeWeb version v1.14 released on 2020-04-18. The application uses appropriate escaping when user input is used in the contexts listed above.

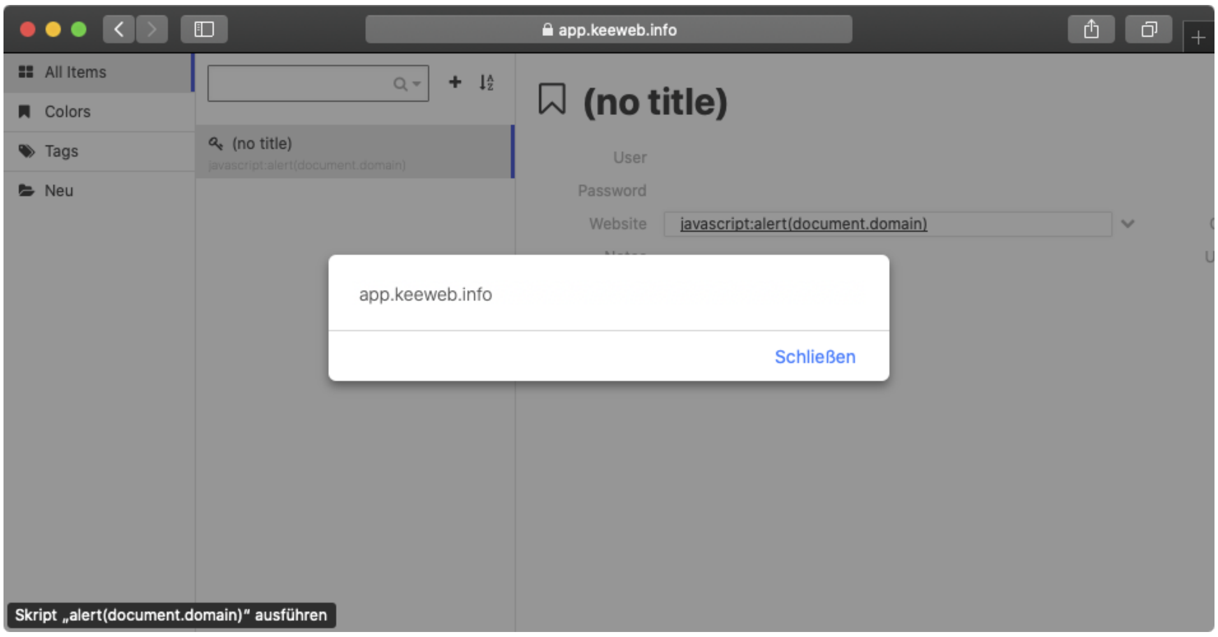
6.3 H03 XSS via a Pseudo-Protocol Definition

Exploitability Metrics		**Impact Metrics**	
Attack Vector (AV)	**Network**	Confidentiality Impact (C)	**High**
Attack Complexity (AC)	**Low**	Integrity Impact (I)	**High**
Privileges Required (PR)	**Low**	Availability Impact (A)	**Low**
User Interaction (UI)	**Required**	Scope (S)	**Unchanged**
Subscore: **2.1**		Subscore: **5.5**	
Overall CVSS Score for H03: 7.6			

This issue is related to the XSS weakness H02. It is mentioned separately due to the reason that it is another variant of XSS attacks.

Weakness. An *entry* within the database can include the URL of a website. To inject JavaScript code, the address form field should consist of a JavaScript pseudo-protocol definition. As an example, a click event on the link created after inserting the pseudo-protocol definition `javascript:alert(document.domain)` will lead to a JavaScript code execution. As described in H02, this payload can also be injected by uploading an XML file containing the payload – therefore increasing the probability of being attacked.

Please note that nowadays only some browsers like Safari allow the execution of JavaScript code by clicking on a `javascript:` pseudo-protocol definition; Google Chrome and Firefox will block such requests (e.g., with a message like `about:blank#blocked`).



(no title)
javascript:alert(document.domain)

Website: javascript:alert(document.domain)

app.keeweb.info

Skript „alert(document.domain)" ausführen

Figure 3: JavaScript code executed in Safari 13 after a click event on the `javascript:` pseudo URI.

Countermeasures. We recommend not allowing the user to use pseudo-protocols like `javascript:` as a user input. A possible solution could also check with a whitelist if a specified URL starts with `http://` or `https://`.

Retest. We can confirm that this weakness was successfully fixed in KeeWeb version v1.14 released on 2020-04-18. The application validates the protocols allowed in URLs using an appropriate whitelist.

6.4 M01 Use of an Embedded User-Agent for User Authentication

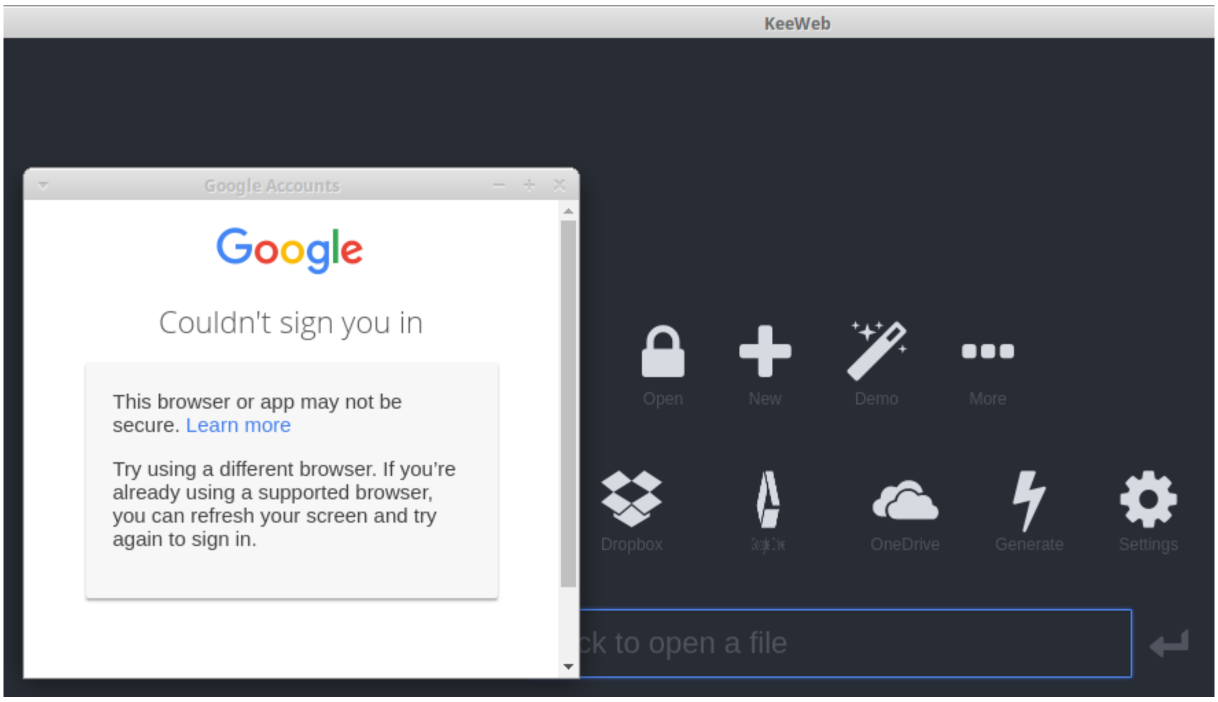
Exploitability Metrics		**Impact Metrics**	
Attack Vector (AV)	**Local**	Confidentiality Impact (C)	**High**
Attack Complexity (AC)	**High**	Integrity Impact (I)	**High**
Privileges Required (PR)	**High**	Availability Impact (A)	**None**
User Interaction (UI)	**Required**	Scope (S)	**Changed**
Subscore: **0.6**		Subscore: **5.8**	
Overall CVSS Score for M01: 6.9			

General Description. When a native application is used to access resources protected by OAuth, the user needs to authenticate to the authorization server and authorize the access. The native application starts the authorization by opening the interface of the authorization server in either the system's default browser or in a user-agent (UA) embedded in the native app itself. Both options were initially defined in the "OAuth Authorization Framework" [2, 9]. However, the later defined RFC "OAuth 2.0 for Native Apps" [5, 8.12] prohibits the usage of an embedded UA and recommends instead to always use the system's default browser. This constraint is reasoned in the security drawbacks that embedded UAs provide. Google already blocks authentication requests from embedded browsers and does not issue access tokens to native applications using embedded browsers (see Figure 4).

Weakness. The native version of KeeWeb currently uses an embedded UA to prompt the user for authentication at the authorization server and authorizing the access to the user's cloud storage. The use of an embedded UA increases the attack surface for both the access tokens and the credentials of the user which are entered in at the authorization server. Embedded UA do not enforce the SOP and are generally considered less trustworthy than the system's default browser.

Countermeasures. We recommend using the system's default browser for the authentication and authorization of the user at the authorization server, and avoiding to use an embedded UA in general.

Retest. We can confirm that this weakness was successfully fixed in KeeWeb version v1.14 released on 2020-04-18. The application uses the system's default browser for authentication and authorization at all cloud storage providers.



Google

Couldn't sign you in

This browser or app may not be secure. Learn more

Try using a different browser. If you're already using a supported browser, you can refresh your screen and try again to sign in.

Figure 4: Google blocks authorization requests from embedded UAs.

6.5 M02 Possible Access Token Injection with postMessage() API

Exploitability Metrics		Impact Metrics	
Attack Vector (AV)	**Network**	Confidentiality Impact (C)	**Low**
Attack Complexity (AC)	**High**	Integrity Impact (I)	**Low**
Privileges Required (PR)	**None**	Availability Impact (A)	**Low**
User Interaction (UI)	**Required**	Scope (S)	**Unchanged**
Subscore: **1.6**		Subscore: **3.4**	

Overall CVSS Score for M02: 5.0

General Description. The `postMessage()` API can be used to bypass the SOP when two windows need cross-origin communication. One example is the communication between one window and a popup or iframe. The receiving window uses an event listener in order to receive messages and is in charge of handling any data received. Generally, any window can send messages to another one if it obtains a reference to the receiving window.

Weakness. The application opens a popup window for the authentication and authorization of the user at the authorization server. After the user finishes the authorization, the popup sends the obtained access token to the application using the `postMessage()` API. To receive the access token, the application listens to messages sent using the `postMessage()`

API. The application extracts the access token from the received message and uses it to access the user's cloud storage. However, the application does validate the sender of received messages. This allows an attacker to inject their own access token in the application by sending a message via the `postMessage()` API. The application will use the access token to access the attacker's cloud storage instead of the user's cloud storage. If the user wants to use the cloud storage to store their locally opened password database, he or she will store their database in the attacker's cloud storage. The attacker can login to their own cloud storage and access the password database of the user. Another attack scenario could be that the attacker makes the user open a database from the attacker's cloud storage. This could be useful for other attacks (e.g., H02).

The technical steps of a possible access token injection attack are as follows:

1. The attacker lures the victim to access a TLS-secured website controlled by them.
2. The attacker's website opens the KeeWeb application in a new window:
   `keeweb = window.open('https://app.keeweb.info')`
3. The victim uses the application as it usually would. When the victim starts the authorization for a cloud storage, the application opens a new popup and listens for messages sent via the `postMessage()` API.
4. The attacker sends a message containing their access token to the application using the `postMessage()` API (see Listing 2).

```
keeweb.postMessage({"access_token":"ya29.a0Ae4lvC2t2ubY8BwwBTJvcsXQOIAvmlWTjlYVGGGmMjG
mCBD_JiyxXZRmIKAP34nyvg8OZA7RklOviUmjDhjuZaF2IRG_Tja1GYOFVkY4EK3ekYYGlbVqvY02B7oudwE2K
I-TT8FSLjbBusCLCH_TaLQ8OtByF689ayif","token_type":"Bearer","expires_in":"3599","scope"
:"https://www.googleapis.com/auth/drive"}, 'https://app.keeweb.info')
```

Listing 2: JavaScript command to send a Google access token to the application using the `postMessage()` API.

Countermeasures. We recommend adding a check if the origin of the message is equivalent to the application's domain. Only messages passing this check should be accepted by the application. This prevents the attacker from sending arbitrary messages containing the attacker's access token to the application using the `postMessage()` API. An example countermeasure is given in Listing 3.

```
...
const windowMessage = e => {
  if (e.origin != 'https://app.keeweb.info') {
    return;
  }
...
```

Listing 3: Possible countermeasure that rejects messages sent from other origins than the application's domain.

Retest. We can confirm that this weakness was successfully fixed in KeeWeb version v1.14 released on 2020-04-18. The application checks the origin of messages received via the `postMessage()` API and does not accept messages from origins other than the application's own domain.

6.6 M03 Violation of the OAuth 2.0 Best Current Practices

Exploitability Metrics		**Impact Metrics**	
Attack Vector (AV)	**Network**	Confidentiality Impact (C)	**Low**
Attack Complexity (AC)	**High**	Integrity Impact (I)	**Low**
Privileges Required (PR)	**None**	Availability Impact (A)	**None**
User Interaction (UI)	**Required**	Scope (S)	**Unchanged**
Subscore: **1.6**		Subscore: **2.5**	

Overall CVSS Score for M03: 4.2

The CVSS score for M03 is defined by the most severe violation not contained in the previous sections.

General Description. The OAuth 2.0 authorization framework is a complex construct of multiple standards which allow various protocol flows and configuration decisions. The "OAuth Working Group"[7] which standardized OAuth as part of the Internet Engineering Task Force (IETF) provides multiple documents defining best current practices for the secure implementation and use of OAuth. In addition to the general "OAuth 2.0 Security Best Current Practice" [3], there are further documents for specific scenarios, such as "OAuth 2.0 for Native Apps" [5] and "OAuth 2.0 for Browser-Based Applications" [1]. Applications using OAuth should follow these best current practices as precisely as possible to strengthen the security of the application and avoid possible weaknesses and attacks.

Weakness. The application uses OAuth to access the user's cloud storage at one of the following providers: Dropbox, Google Drive, and OneDrive.

With each OAuth provider, the application violates multiple measures according to the best current practices. The violations are listed in Table 3.

Countermeasures. We strongly recommend implementing the OAuth best current practices as much as possible in accordance to the supported authorization servers. This ensures that the application uses OAuth in a secure way.

Retest. We can confirm that this weakness was partly fixed in KeeWeb version v1.14 released on 2020-04-18 and completely fixed in KeeWeb version v1.14.2 released on 2020-05-04. The application uses the `state` parameter to protect against cross-site request forgery (CSRF) attacks and distinct and unique `redirect_uris` for each cloud storage provider to protect against mix-up attacks.

[7] `https://datatracker.ietf.org/wg/oauth/about/`

Violation	Best Current Practice	Reference
Use of the OAuth implicit grant	Use the authorization code grant in conjunction with the PKCE extension. See H01 for details.	[3, 2.1.2], [5, 8.2], [1, 4]
Missing CSRF protection	Protect the application against cross-site request forgery (CSRF) attacks. For example, use the `state` parameter.	[3, 2.1], [5, 8.9], [1, 9.4]
Use of an embedded UA	Use the system's default browser for authorization grants. See M01 for details.	[5, 8.12]
Missing mix-up attack protection	The application supports the usage of multiple authorization servers but does not use protection against authorization server mix-up attacks. The application should use an unique `redirect_uri` for each supported authorization server to distinguish which authorization server issued the token response.	[3, 2.1], [5, 8.10], [1, 9.5]
Missing CSP	Applications using refresh tokens should restrict mailicious JavaScript execution by defining a strong content security policy (CSP). See R01 for details.	[1, 9.7]

Table 3: Violations of the OAuth best current practices.

7 Recommendations

In the following sections, we provide our recommendations to improve the security of the tested system.

7.1 R01 Delivering the Content Security Policy

We recommend using the CSP as a countermeasure to minimize the risk of content injection vulnerabilities, such as XSS. According to CSP Version 3,

> "A Document may deliver a policy via one or more HTML `meta` elements whose `http-equiv` attributes are an ASCII case-insensitive match for the string *Content-Security-Policy*."[8]

As an example, `<meta http-equiv="Content-Security-Policy" content="script-src 'self'">` could be used to only allow the inclusion of scripts from the same origin.

Please note that it is usually a better approach to use an HTTP header instead of a `meta` element. By looking on the current implementation of the application, some content injection risks can be mitigated if the `meta` element is placed as early as possible in the document.

Retest. We can confirm that this recommendation was successfully implemented in KeeWeb version v1.14 released on 2020-04-18. The application delivers an appropriate CSP when accessed by a browser.

7.2 R02 Implementing a Logout Option

Currently, the application does not offer any option to remove access tokens after a user has authorized it to access their cloud storage. The user has to manually delete the browser's storage if he or she wishes to remove the access tokens. This could be necessary if he or she used a public or shared computer to access their cloud storage or wishes to log in at a cloud storage provider using a different account.

We recommend implementing an option which allows the user to logout at the supported cloud storage providers. The logout option should revoke the access token at the authorization server and remove the revoked access token from the browser's storage. Removing the access token from the browser's storage reduces the risk of the access token being leaked and revoking it mitigates the risk even if the access token was previously leaked.

Retest. We can confirm that this recommendation was successfully implemented in KeeWeb version v1.14 released on 2020-04-18. The application offers an option to logout at the cloud storage providers. When the option is invoked the application removes the

[8] `https://www.w3.org/TR/CSP3/`

access token from the browser's storage and revokes it if supported by the authorization server.

References

[1] D. Waite A. Parecki. *OAuth 2.0 for Browser-Based Apps - Draft 05*. Internet Engineering Task Force, Feb. 2020. URL: https://tools.ietf.org/html/draft-ietf-oauth-browser-based-apps-05.

[2] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). Internet Engineering Task Force, Oct. 2012. URL: https://tools.ietf.org/html/rfc6749.

[3] T. Lodderstedt et al. *OAuth 2.0 Security Best Current Practice - Draft 14*. Internet Engineering Task Force, Feb. 2020. URL: https://tools.ietf.org/html/draft-ietf-oauth-security-topics-14.

[4] N. Sakimura, J. Bradley, and N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636 (Proposed Standard). Internet Engineering Task Force, Sept. 2015. URL: https://tools.ietf.org/html/rfc7636.

[5] J. Bradley W. Denniss. *OAuth 2.0 for Native Apps*. RFC 8252 (Proposed Standard). Internet Engineering Task Force, Oct. 2017. URL: https://tools.ietf.org/html/rfc8252.