Pentest-Report Obsidian Clients & UI 11.2023

Cure53, Dr.-Ing. M. Heiderich, M. Pedhapati, Dr. D. Bleichenbacher, H. Li, R. Maini, H. Jaiswal, C. Luders

Index

Introduction

> *"Obsidian is the private and flexible writing app that adapts to the way you think. Obsidian stores notes on your device, so you can access them quickly, even offline. No one else can read them, not even us. Obsidian uses open, non-proprietary files, so you're never locked in, and can preserve your data for the long term."*
>
> From https://obsidian.md/

This document pertains to the first iteration of a multifaceted penetration test and source code audit against various Obsidian software client aspects performed by Cure53 in Q4 2023.

To give some context regarding the assignment's origination and composition, the client contacted Cure53 in September 2023 with the proposal to perform a full-scale security assessment of the focus traits. The test execution was scheduled for CW46 November 2023, whereby eighteen work days were invested to materialize a precise appraisal of the framework's susceptibility to attack and compromises. Six senior consultants were selected to fulfill the analysis and requirements based on their abundant experience and technical know-how handling features of this nature.

Only a single Work Package (WP) was required for this document, which reads as follows:

- **WP1**: Crystal-box pentests & code audits against Obsidian clients & UI

The methodology conformed to a crystal-box strategy, whereby assistive materials such as sources, documentation, and other assorted entities were provided to facilitate the undertakings. This phase was preceded by the preliminary stage, within which a selection of essential preparation actions were completed in CW45 in order to fully equip the testers for the endeavors ahead.

A private and shared Discord channel was established for communications between the two organizations. Generally, the discourse was seamless and highly conducive to a productive pentest. The exhaustive scope setup meant that cross-team queries regarding the goals or underlying infrastructure were altogether minimal, while the procedures were not delayed or blocked outright at any point. The testers also relayed a multitude of progress and status updates during the live reporting process, which served to raise awareness of certain high-profile findings at the point of detection.

Following a reasonable depth of coverage across the targets, the assessors witnessed and documented four security-relevant discoveries for the Obsidian clients and UI. This yield is undeniably small; however, the worrisome impact levels of the findings contribute to the negative overall impression.

The two *Critical* severity vulnerabilities garnered elevated concern, entailing the exposure of arbitrary file read scenarios as outlined in tickets DYL-01-006 and DYL-01-007. The Obsidian project management team should strive to remediate these severe faults as soon as possible.

To summarize, the Obsidian clients and UI are unfortunately blighted by substantially damaging circumstances if successfully exploited by an attacking party. However, despite the unfavorable final verdict, Cure53 is certain that a satisfactory security standard is attainable through the immediate mitigation of all shortcomings described herein. Furthermore, the insights garnered for this exercise's second part should be perceived simultaneously in order to extract a holistic overview of the wider Obsidian client security posture.

Moving on, the report will now provide bullet-pointed information related to the scope, test setup, and leveraged materials. Following that, all findings are presented in chronological order of identification with two subsections: *Identified Vulnerabilities* and *Miscellaneous Issues*. Each finding will include a technical outline, a Proof-of-Concept (PoC), and ideal methods to resolve the limitation in question. Finally, the report will conclude by elaborating on the general impressions gained during this test and discussing the perceived security posture of the Obsidian clients and UIs.

Scope

- **Penetration tests & source code audits against Obsidian software & UI**
    - **WP1**: Crystal-box pentests & code audits against Obsidian clients & UI
        - **Documentation:**
            - Detailed documentation was shared with Cure53
            - *Local_dev_environment_guide.md*
        - **Obsidian:**
            - **Repo:**
                - /obsidian
        - **Branch:**
            - obsidian-master/
        - **Commit ID:**
            - da7a11f80520c0535afff08f09ee9e062230cb02
        - **Obsidian Static:**
            - **Repo:**
                - /obsidian-static
            - **Branch:**
                - obsidian-static-master/
            - **Commit ID:**
                - d3cd2d346872e96c60fce60e73d1524af2e20c69
    - **Test-supporting material was shared with Cure53**
    - **All relevant sources were shared with Cure53**

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, all tickets are given a unique identifier (e.g., *DYL-01-001*) to facilitate any future follow-up correspondence.

DYL-01-001 WP1: CORS bypass via flawed URL validation (*High*)

***Note**: This issue was fixed by the Obsidian development team and the fix was verified by Cure53. The issue as described no longer exists.*

Testing confirmed the presence of a vulnerability affecting the Obsidian Electron app regarding the ability to bypass the Cross-Origin Resource Sharing (CORS) checks for any URL, which permits any embedded page to access data from arbitrary web pages.

Browsers implement a critical security check known as the Same-Origin Policy (SOP), which prevents web pages from accessing data via alternative origins. The CORS protocol circumvents the SOP by leveraging specific response headers, enabling specific origin web pages to access their responses.

Cure53 noted that modifications to the response header for the Obsidian Electron app have been specifically implemented for third-party plugins. *Access-Control-Allow-Origin* was altered to an asterisk (*) to enable web pages to access the response if the URL matched a certain pattern. The URL checks seemed secure at first glance, owing to the fact that the path in the URL always ends with a forward slash (/) and hence prevents the use of query strings or hash fragments to bypass the checks. However, this can be circumvented by utilizing a username in the URL, such as *https://github@google.com*.

As a result, requests from the embedded page could bypass CORS checks and access responses from any website, including localhost and alternative sites that are only accessible within the intranet, thus incurring substantial risk.

Affected file:
obsidian-master/src/main.ts

Affected code:
```
if (!hasOrigin && (/^https?:\/\/[^\/]*(github)/.test(url))) {
    responseHeaders['Access-Control-Allow-Origin'] = ['*'];
}
```

Steps to reproduce:

1. Copy the following code snippet and paste it to an Obsidian note:

   Example code snippet:
   ```
   <iframe width="100%" height="700px" 
   src="https://randomstuffhuli.s3.amazonaws.com/c53/obs-f832e40a/cors-
   bypass-exp/index.html"></iframe>
   ```

2. Switch to preview mode.
3. Observe that the *https://google.com* content is displayed.

To mitigate this issue, Cure53 advises utilizing an URL parser API rather than regular expressions when conducting URL checks, due to the latter's propensity for security flaws and lack of edge-case coverage. Consequently, the safest approach would typically involve parsing the URL, extracting the hostname, and determining whether it corresponds to a legitimate domain.

In terms of the intended functionality, the developer team could validate whether the parsed hostname from the URL represents *github.com*, which would prevent circumvention using usernames or other methods. In addition, one could prepare an allow list to compare against legitimate hostnames rather than solely checking that the hostname culminates in *.github.com*, which is another sound and common measure to combat this weakness. However, this strategy effectively permits all subdomains and hence poses additional security implications.

To summarize, employing an allow list for stringent comparison would be the safest resolutionary implementation, effectively preventing URL bypass methods that leverage usernames, query strings, hash fragments, and similar.

DYL-01-006 WP1: Arbitrary file read via local file embedding (*Critical*)

***Note**: This issue was fixed by the Obsidian development team and the fix was verified by Cure53. The issue as described no longer exists.*

Cure53 acknowledged that the vulnerability identified as CVE-2023-2110 has not been comprehensively mitigated. Accordingly, an attacker is granted the opportunity to read arbitrary local files and bypass the implemented remediation measures under certain circumstances.

CVE-2023-2110 exploits the *app://local* protocol registered by Obsidian to access local files, circumventing checks related to referrer. It has been noted within the source code that *app://local* has been deprecated and replaced by generating a random ID upon startup, such as *app://dd0d859619595c5cd0ad427acbb436600f97*. In the event that the web page remains unaware of the random ID, local file access will be blocked.

However, another pertinent function entitled *fixFileLinks* has been designed to replace elements in Markdown files, whereby *src* contains *file:///* with the aforementioned random ID representation.

In essence, if an iframe's source represents *file:///tmp/test.html*, it will be replaced with *<iframe src="app://{Platform.resourcePathPrefix}/tmp/test.html">* during preview. Consequently, *test.html* can utilize *window.origin* to obtain the random ID and subsequently access other local files.

If a malicious actor is able to write HTML on the computer and possesses knowledge of the file's location, they would be able to bypass the fix applied for CVE-2023-2110 and access arbitrary local files.

In practice, this exploit may entail social engineering or another similar tactic that prompts a victim to download and save a file to a specified location. Additionally, browsers often leverage default download locations such as */Users/username/Downloads*, which may be brute-forced or targeted. Certain operating systems may even facilitate username leakage[1], which increases the likelihood of a successful compromise.

Affected file:
obsidian-master/src/main.ts

Affected code:
```
protocol.registerFileProtocol('app', (req, callback) => {
    let url = req.url;
    let noframe = false;
    // ...
    else if (url.indexOf(FILE_ROOT) === 0) {
      url = decodeURIComponent(url.substr(FILE_ROOT.length));
      if (!isWin) {
        url = '/' + url;
      }
      url = path.resolve(url);
      // Disallow framing if the path is a UNC path
      if (isUncPath(url)) {
        noframe = true;
      }
    }
    else {
      url = '';
    }

    let headers: Record<string, string> = {};
    if (noframe) {
      headers['X-Frame-Options'] = 'DENY';
```

[1] https://fingerprint.com/blog/apple-macos-mdns-brute-force/

```
    }
    callback({ path: url, headers });
  });
```

Steps to reproduce:

1. Run the following command to write an HTML file locally:

   Command:
   ```
   echo
   "<script>fetch(origin+'/etc/hosts').then(res=>res.text()).then(alert)
   </script>" > /tmp/test.html
   ```

2. Copy the following Markdown content and paste it to Obsidian:

   Markdown content:
   ```
   iframe: <iframe src="file:///tmp/test.html"></iframe>
   ```

3. Switch to reading view.
4. Observe that an alert is displayed with the */etc/hosts* content.

To mitigate this issue, Cure53 advises preventing the embedded web page from initiating any requests to the app protocol. Considering the plausible attack method, the insertion of a random ID into the app protocol would not suffice, and other vulnerabilities may lead to the leakage of the random ID, which would enable adversaries to retrieve local files via the leaked ID.

Therefore, the root cause can be attributed to the fact that unauthorized pages (such as the embedded page) are able to send requests to the app protocol and read the response. The Obsidian team should henceforth prioritize fixing this particular aspect, which is achievable by blocking all requests when an embedded web page attempts to access resources from the app protocol. This action will ultimately prevent the embedded web page from accessing local files.

DYL-01-007 WP1: Arbitrary file write via path traversal in Sync plugin (*Critical*)

***Note**: This issue was fixed by the Obsidian development team and the fix was verified by Cure53. The issue as described no longer exists.*

While reviewing the Sync plugin's source code, Cure53 witnessed that the path is not sanitized, which may enable path traversal exploitation opportunities for the purpose of writing files to arbitrary locations.

When a new file is received from the Sync server, an *allowSyncFile* check is enforced on the client side to examine the file's extension and type. However, this process fails to inspect whether the filename contains special characters. Moreover, the file synchronization process also neglects to verify whether the final write location is within the vault. This ultimately evokes a path traversal vulnerability that enables file placement in arbitrary locations using a filename, such as *any_folder_name/../../../../tmp/test.md*.

Since remote vaults are shareable with other users, adversaries can construct a remote vault with a malicious payload and invite a victim to collaborate. If the victim connects to the vault and initiates synchronization, the malicious file will be downloaded to the victim's computer at the specified file location.

The Sync feature permits a number of default syncable file types, including *md*, canvas, and images, for which SVG is recognized as a legitimate image. However, since SVG also functions as XML when constituting an image, it can be embedded as a web page. By leveraging the flaw outlined in ticket DYL-01-006, adversaries can write SVG to a specified location and use iframes to access arbitrary files on the computer.

A user may plausibly modify their settings to permit syncing of additional file types, i.e., via the selection of *Sync all other types*. This will enable adversaries to write executable files or bash scripts to the computer, elevating the likelihood of Remote Code Execution (RCE) and other associated risks.

Affected file:
obsidian-master/src/app/plugins/sync/sync.ts

Affected code:

```
async syncFileDown(server: ServerConnection, syncFile: SyncFile) {
        let path = syncFile.path;
        // ...
        let folder = getDirName(path);
        if (folder !== '/' && folder !== '' && !await adapter.exists(folder))
{
                await adapter.mkdir(folder);
        }
        await adapter.writeBinary(path, data, {
                ctime: syncFile.ctime,
                mtime: syncFile.mtime,
        });
}
```

Steps to reproduce:

1. Create a new local vault.
2. Enable the Sync plugin.
3. Open DevTools and switch to the network tab.
4. Create a new and passwordless remote vault.
5. Inspect the */vault/create* request to retrieve the *token*, *id*, *password*, and *salt*.
6. Download the PoC file from https://cure53.de/exchange/457632098572348975/socket.js
7. Edit the PoC file and update the *token*, *vaultId*, *vaultPassword*, and *vaultSaltl*.
8. Run the following command:

   Command:
   ```
   npm init -y
   npm install websocket
   node socket.js
   ```

9. Connect to the remote vault created at Step 3.
10. Click *Start syncing* on the UI.
11. Close the settings tab.
12. Click on the *arb_write* file in the vault.
13. Verify that the */etc/hosts* content is displayed on the UI and that */tmp/arb_write.svg* has been created.

To mitigate this issue, Cure53 advises checking the legitimacy of paths when synchronizing files from the remote server to the local system. This requires normalizing the path and verifying whether it exists within the vault folder; if this is not the case, the synchronization request should be rejected.

By ensuring path legality in this context, all synchronized files will be written exclusively within the vault folder, henceforth neutralizing the capability to write to other file directories and thus blocking path traversal attempts.

DYL-01-009 WP1: App protocol origin leak via CSS snippets *(High)*

***Note**: This issue was fixed by the Obsidian development team and the fix was verified by Cure53. The issue as described no longer exists.*

Cure53 noted that many features within Obsidian load third-party CSS files through the use of two functions: theme and CSS snippets. Both functionalities allow users to define additional CSS and alter the UI style. However, aside from style modifications, CSS can also be employed to extract data from the page, such as text or attribute values.

For instance, using the CSS selector *img[src^="a"]* enables one to choose elements whereby the *src* attribute initiates with an *a*. This can load another background image with a URL such as *http://example.com?q=a*. When a page contains an image with the *src* attribute beginning with *a*, the browser sends a request to *http://example.com?q=a*. By repeating this action, adversaries can progressively leak characters, ultimately revealing the entire content of the *img src* on the server side.

As mentioned in ticket DYL-01-006, the *fixFileLinks* function corrects all *file:///* prefixed sources to the app protocol. However, adversaries can exploit similar methods, leveraging CSS to disclose complete URLs. Once the full URL is exposed, adversaries can determine the correct app protocol ID.

If a user enables the Sync function and allows Sync appearance settings alongside CSS snippets or themes, malicious actors will be granted the ability to load prepared CSS files using CSS to leak the app protocol. Subsequently, they can leverage the Sync feature to automatically load files containing iframes, thus enabling the embedded page to access local files via the leaked URL.

Steps to reproduce:

1. Download the CSS leak server file:
   https://cure53.de/exchange/457632098572348975/server.js
2. Run *node server.js*.
3. Open the settings modal in Obsidian.
4. Click *Appearance*.
5. Scroll to the CSS snippets section and open the snippets folder.
6. Download the following file to the folder:
   https://cure53.de/exchange/457632098572348975/exp.css
7. Click *Reload snippets*.
8. Enable *exp.css*.
9. Close the settings modal.
10. Create a new note.
11. Input *<img id=leak src=file:///>*.
12. Switch to the reading view.

13. Observe that the leaked origin is printed on the server console. If this is not the case, attempt the steps again after restarting the server and Electron app.

To mitigate this issue, Cure53 suggests imposing strict CSP limitations on the sources from which styles and other resources are loadable, which would prevent adversaries from exploiting CSS files to leak information. Despite the fact that CSS snippets and themes load CSS files locally, common CSS exploits often require the use of *@import*[2] to reload CSS from a remote server. As such, one can mitigate the impact of this particular shortcoming by restricting the *style-src* in CSP.

Given that users can load style files of their own accord, comprehensive remediation of the CSS data exfiltration fault would prove highly challenging. However, the Obsidian team can limit its impact where feasible. For instance, resolving the limitation outlined in ticket DYL-01-006 would mean that adversaries will be blocked from accessing files even if they are able to obtain the app protocol origin, hence lowering the efficacy of this attack strategy. One must also note that all elements presented on a web page (such as text) will remain susceptible to CSS data exfiltration, which renders complete mitigation significantly difficult.

Alternatively, the developer team could prevent shared vaults from syncing themes and CSS snippets, therefore enforcing that only remote vaults created under one's own account would be permitted to synchronize the two setting types in question. This approach would restrict adversaries from sharing shared vaults that contain malicious styles and consequently constrain the attack surface.

[2] https://research.securitum.com/css-data-exfiltration-in-firefox-via-single-injection-point/

Conclusions

In this section, Cure53 aims to provide in-depth commentary regarding all viewpoints and bug classes observed during the winter 2023 crystal-box examination of the Obsidian clients and UI. With this, one can hope that the Obsidian maintainers can glean the security efficacy of the scope based on the evidence collected.

The testers primarily concentrated on reviewing the Obsidian Sync feature and compiling an accurate risk assessment with concern for the plausibility of a system compromise via offensive or breach schemes.

The communication between the client and server relies on HTTPS and WebSockets. HTTPS is utilized to create remote vaults, while communication with the sync server occurs via WebSockets for data transfer.

During vault synchronization, checks are performed on file names to ensure that only permitted file extensions are syncable. Other files require syncing additional settings, which is indicative of the dev team’s security considerations at the design level.

However, the download function during the file synchronization process was verified to be afflicted by a path traversal vulnerability. Specifically, the client fails to normalize the path, which enables adversaries to write files to arbitrary locations on the computer. Supporting guidance on this erroneous situation is provided in ticket DYL-01-007.

In addition to syncing regular *noteSlacks*, Obsidian also furnished the option to sync other configuration files. While this behavior is convenient when combined with shared vault functionality, the enhanced practicality ultimately expands the attack surface. For instance, shared vaults can enable malicious actors to load crafted CSS files for data theft purposes, as extrapolated in ticket DYL-01-009. The Obsidian developers can minimize the attack surface and increase the framework’s resilience to associated threats by restricting shared vault permissions, e.g., limiting syncing to note-related files.

Elsewhere, embedded iframe-related functionality has been pinpointed as an area of weakness that continues to affect the Obsidian compound. In light of this, Cure53’s source code evaluation determined the presence of a CORS bypass vulnerability, as highlighted in ticket DYL-01-001.

The app protocol was also considered another pertinent and prevalent attack vector. Despite the patches applied for previous vulnerabilities, an adversary can still bypass checks and access local files under certain circumstances, as evidenced in ticket DYL-01-006.

To finalize this report, Cure53's inspections of the respective source code substantiate the argument that astute security paradigms and vulnerability remediation protocols have been considered by the Obsidian team. However, some noteworthy and fundamental flaws remain unresolved, accentuating the potential for patch bypasses in order to reproduce certain detrimental behaviors.

Furthermore, the Cure53 technicians were only able to achieve moderate coverage over the Obsidian Electron source code specifically, which necessitates the requirement for supplemental auditing assignments in order to fully comprehend all security faults that are persisted.

Cure53 would like to thank Erica Xu, Steph Ango, Shida Li, and Tony Grosinger from the Obsidian team for their excellent project coordination, support, and assistance, both before and during this assignment.