Project - Browser Extensions security

Zaina Ramadan, Lukas Gamard

June 7, 2024

1 Background

Users are offered a lot of options when it comes down to browsing the web.They have many browsers to choose from, depending on what OS they use and what properties they value most. A user can choose a browser that focuses more on privacy but maybe compromises the user experience. To further personalize their browsing experience, users can also pick between a myriad of browser extensions to further personalize their environment. Among those, one can find password managers, tab managers and all sorts of cosmetic extensions. Browser extensions open however new attack vectors, some of which we will describe in our report. Finally, we will present a prototype for static analysis of browser extensions, looking to prevent extensions with common vulnerabilities to be released.

2 Goal

In order to not spread ourselves too thin, we will focus our study on Chrome browser extensions. Google Chrome is one of the most popular browsers nowadays, and provides extensive documentation for browser extension development. In this context, we will first go through the basics of browser extensions, present their security model and some vulnerabilities. Our goal is to provide the reader with a framework that can be used to statically analyse an extension, in order to identify some possible vulnerabilities.

3 Chrome browser extensions

3.1 General structure of browser extensions

From a user's perspective, an extension is accessible through a button located close to the URL input field in the browser, which gives access to all sorts of passive or active functionalities. We will however focus on the developer's view, and look with more details into what happens in the background, how Chrome browser extensions are structured, and how they operate. We will not spend time on UI elements, but instead discuss the extension's overall structure and security aspects in more depth.

From a developer's perspective, a browser extension is a program running on top of the browser. It can access the DOM, has a local memory, and can even communicate with remote servers [8]. All of this is regulated by a security framework, which we intent to describe. The main components of an extension are [1]:

- **the extension manifest**, contained in a single file named `manifest.json`, located in the extension's root. The manifest is a mandatory file, describing the extension's behavior, as well as declaring a list of the set of permissions it requires to operate.
- **service workers**, core component of the extensions runtime environment. Those are a new development, part of the new version of the manifest, currently being migrated to. Previously, the background of an extension consisted of an event script that was persisting through the life-cycle of the extension. This part has been redesigned to instead have more task-oriented service workers, executing specific tasks.
- **content scripts**, allowed to interact with and modify the DOM.
- **UI components**, consisting of toolbar action and side panel
- **declarativeNetRequest**, allowing the extension to block, intercept or modify network requests

The different scripts are written using Javascript, the UI elements with HTML, and the manifest in JSON. As we will describe in the next section, the different scripts operate in different environments, and can only interact with each other through API calls.

3.2 Security model of browser extensions

Before we delve into which extension vulnerabilities are common and how they are exploited, we need to have an understanding of the security model of browser extensions and what functionality is allowed or disallowed. For the scope of this project, we will discuss the Chrome extension security model, which was designed to protect users from vulnerabilities in benign-but-buggy extensions (meaning extensions that contain flaws or errors that can cause unintended behavior or security risks). The Chrome security architecture builds on the principles of least privilege, privilege separation and strong isolation. Below we present each of the three principles and how they are used in the Chrome security architecture [4] [6].

3.2.1 Least privilege

The principle of least privilege states that any entity, such as a user or a program, is granted only the necessary permissions and resources to perform its intended task [10]. This is enforced in Chrome extensions by requiring developers to specify the privileges needed in the manifest file. Extensions are only granted access to the privileges declared in the manifest. Listing 1 [5] shows an example of an extension's manifest file. We see in the permissions that the extension can access "ajax.googleapis.com" and the API for activeTab. Would the extension in listing 1 become compromised, the attacker would not have the privilege to access "bank.com", for instance.

Listing 1: simple extension example

```
{
  "manifest_version": 2,

  "name": "Getting started example",
  "description": "This extension shows a Google Image
                   search result for the current page",
  "version": "1.0",

  "browser_action": {
    "default_icon": "icon.png",
    "default_popup": "popup.html",
    "default_title": "Click here!"
  },
  "permissions": [
    "activeTab",
    "https://ajax.googleapis.com/"
  ]
}
```

Although this restricts what an extension can do, developers can (and are likely to) request more privileges than needed. They may even request the privilege to execute arbitrary code by listing a native binary in the manifest. This is of course the most dangerous form of extensions, where arbitrary, and possibly malicious code, can be executed. To mitigate this, Google doesn't allow such extensions to be published on its gallery unless the developer of the extension signs a contract with Google.

3.2.2 Privilege separation

Extensions are divided into multiple components: content scripts, the extension core, and a native binary. Each component has its own set of privileges and permissions where access between the different components is restricted. This separation helps contain the impact of security breaches by limiting the level of access an attacker can obtain. Below, we present each component and its privileges:

- **Content scripts** interact directly with web pages via the DOM. Consequently, they have low privileges and are at the highest risk of attack.
- **Extension core** has direct access to the extension API:s declared in the manifest, unlike the content scripts, and has therefore high privileges. The extension core does not directly interact with untrusted websites. It can instead either interact via a content script or by issuing an `XMLHttpRequest`.
- **Native Binary** are used to run arbitrary code in the extension. To list a native binary, the developer must supply a native Netscape Plug-in API (NPAPI) binary. The native binary only interacts with the extension core, but its behavior is unpredictable, since it can run arbitrary code. Native binaries are not by default covered by the security measures for web extensions. They must instead undergo a manual security evaluation before they can be published.

By having these different levels of separation, content scripts and the core extension never directly communicate with each other. Instead, they run in separate processes and communicate by sending structured clones over an authenticated channel. This makes it much harder for attackers to exploit vulnerabilities and gain unauthorized access, since these different components cannot be directly linked.

3.2.3 Strong isolation

There are several mechanisms used to isolate extension components from each other and from web content. Firstly, the origin of an extension is specified by including a public key in the extension URL, instead of deriving the origin using information such as the scheme, host, and port of the URL. When the URL is loaded, the browser verifies the extension using the public key. The extension scripts are hence not loaded from the network but from the user's file system. We can therefore utilize the same-origin policy to isolate the extension script from browser internals and other web content.

Another level of isolation is achieved by running the content scripts and the extension core in separate processes. This, for instance, prevents exploits in the web browser from being used to access the extension's API through the extension core.

Lastly, when accessing the DOM, content scripts use their own JavaScript objects instead of the ones used by the webpage they are trying to access. This is known as having isolated worlds where the content scripts and external web pages never exchange JavaScript pointers, making an attack much more difficult. At a high level, the implementation consists of creating a hash table (where the keys are DOM implementation objects) for each isolated world. When a world is executed, the corresponding hash table is used.

3.3 Browser extension vulnerabilities

Despite Google's attempt at enforcing security by having strong isolation and privilege separation, many exploits are still possible. Malicious extensions can abuse the permissions that are requested in the manifest, especially since there is no restriction on which permissions you can request. For instance, consider this simple host permissions `http://*/*` which allows an extension to read and modify all data on a website that a user visits. A relatively easy attack that an extension with this permission could perform is credential theft. Once a user logs in to a website that requires authentication, the malicious extension becomes active and can steal the entered data by invoking `document.getElementById()`. Of course, other (and less obvious) permissions can be abused as well to perform more intricate attacks.

A more dangerous vulnerability is introduced by using `eval()` within the source code, specifically `eval()` coupled with untrusted user data. If the data were to contain malicious code, that code would then get executed within the context of the content script and the strong isolation mechanisms would fail to prevent an attack. Luckily, since `eval()` is well-known for causing security vulnerabilities, few developers use it in their extension code [7].

Click injections are another vulnerability that can be exploited in browser extensions. This type of attack occurs in extensions that register event handlers for DOM elements, for instance, a handler for a button's `onClick()` event. The attacks is performed by having a malicious website invoke the extension's event handler, thereby tricking the extension into performing an action that was not explicitly requested by the user. This is known as a click injection attack [7].

Furthermore, the communication between the content script and the extension core in itself can sometimes cause security risks. The following study [7] found, upon evaluating a dataset of chrome extensions, that two extensions with vulnerable content scripts were able to access some core extension privileges. This is enabled by having the content script send a message to the core extension, requesting the core extension to perform some action with its privileges. In the found extensions, XHR requests were made on behalf of the content script and the results where returned. This essentially means that the content script can make arbitrary XHR requests even though content scripts should not have access to cross-origin XMLHttpRequest objects.

Last but not least, the same study reported that direct network attacks are the largest class of core extension vulnerabilities where 88% of the found core extension vulnerabilities are a result of a direct network attack. This is something to be expected is there is no seperation layer between core extensions and code in HTTP scripts or data in HTTP XMLHttpRequests.

4 Our work

Static analysis can be a powerful tool for mitigating vulnerabilities such as the ones in section 3.3. For our study, we decided to focus on a few elements and present a framework of how a thorough analysis could be conducted. Our threat model is an attacker using a browser extension to execute remote code, inspired by the tool published by Somé [9], we implemented some code to automatically detect possible vulnerabilities in a browser extension. Following the automatic analysis, the extensions should then be reviewed manually to decide if it could indeed be harmful.

4.1 Static analysis of browser extensions

Our work was split into two parts. The first part consists of an examination of the extensions Content Security Policy (CSP) and permissions. We extract the policies specified in the CSP field of the manifest and report `unsafe eval` and `unsafe inline` as potentially unsafe policies. Similarly, for the permission analysis we utilize a file containing a list of permissions that may be exploited for malicious behaviour. If we get a match, the result is reported to the user with some information on how the permission might be misused. We are of course aware that an extension may need some higher permission levels for some functionality, for instance rearranging and managing tabs. However, this could still be potentially dangerous which is why the user should be made aware. Indeed, the `tabs` permission for example gives the extension the possibility to query the tab for its URL, which could be used for phishing or forgery. In the same way, a loose CSP allows for the execution of undesired scripts, which we want to detect.

The second part of our static analysis is an inspection of all the scripts (content scripts and background scripts). Just as previously, some functions or API calls are needed in certain cases, but they still provide a gadget that could be used as an entrypoint for an attack. It is therefore important to properly identify them and review their usage before releasing an extension. We therefore scan the JavaScript files for a set of selected functions and API calls, gathered from different online resources [2, 9]. This set includes invocations of dangerous functions such as `eval`, invocations to the AJAX API which allows bypassing the Same Origin Policy (SOP) and making cross-origin requests as well as storage API calls since they may store or retrieve data for web applications. As before, we report any matches to the user with some text explaining the vulnerability.

In order to evaluate the tool, we needed a data set of Chrome extensions. Since a ready data set was not publicly available, we implemented a web scraper using the Selenium [3] library for automated web browsing. We scraped the ids of several Chrome extensions from the Chrome Web Store and then used the requests library in Python to download the source code for each extension in CRX format.

5 Results

We ran our code on a sample of 234 web extensions, that we gathered using our own crawler. 210 of them were tagged by our program as potentially unsafe and would require further manual investigation. We should note however that our analysis was more of a proof of concept, and should be refined to better filter out extensions that are not harmful. One could, for example, look at the context in which these calls are being invoked to determine if they are actually dangerous. Many of the function calls we flag are for instance dangerous when coupled with user data, this is something we could use to further refine the tool. Another direction for improving the static analysis would be to build an Abstract Syntax Tree (AST) for the scripts and analyse what the possible inputs and outputs to every function call are. This would make the analysis of how the sensitive API calls are used a lot more accurate.

It should also be noted that Google is currently transitioning to the next version for the manifest file (V3), which implies much more restrictions for browser extensions. We still decided to base our tool on the manifest V2, since there are still more applications running on that version.

References

[1] Official documentation for Chrome Extensions. https://developer.chrome.com/docs/extensions/get-started. [Online; accessed May-2024].

[2] bowserjr. https://github.com/notbella/bowserjr/tree/master, 2019.

[3] Selenium. https://www.selenium.dev/, 2024.

[4] Barth, A., Felt, A., Saxena, P., and Boodman, A. Protecting browsers from extension vulnerabilities.

[5] Bondy, B. R. Chrome extension sample on github. https://github.com/bbondy/chrome-extension-sample/tree/master, 2015. [Online; accessed 12-May-2024].

[6] Carlini, N., Felt, A., and Wagner, D. An evaluation of the google chrome extension security architecture.

[7] Carlini, N., Felt, A. P., and Wagner, D. An evaluation of the google chrome extension security architecture. In *21st USENIX Security Symposium (USENIX Security 12)* (2012), pp. 97–111.

[8] Mehta, P. *Creating google chrome extensions.* Springer, 2016.

[9] Somé, D. F. Empoweb: empowering web applications with browser extensions. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 227–245.

[10] Wikipedia. Principle of least privilege. https://en.wikipedia.org/wiki/Principle_of_least_privilege, 2024. [Online; accessed 12-May-2024].

Appendix

Individual contribution

We started planning early together. We agreed on the structure of our report, what our tool would do and it's general structure. Lukas wrote the introduction and general structure parts. Zaina took care of the security model section, and started implementing the tool, including the crawler. Lukas then finished the part analysing content scripts.

Our program

Our tool for atatic analysis can be found at https://gits-15.sys.kth.se/gamard/DD2525.

Analyser output

Here is an excerpt from the result of our static analyser. It helps the manual analysis by locating the potentially harmful scripts in the extension, and naming what API calls are to investigate. Note that the random string corresponds to a unique identifier used for an extension.

```
{
    "afcbjmdkfcdfbamemeadbpmabohjehcl": {
        "unsafe_permissions": [
            {
                "Permission": "webNavigation",
                "Warning_text": "Access your browsing activity",
                "Notes": "The <code>tabs</code> permission is required
                by the <code>chrome.tabs</code> and <code>chrome.windows</code>
                modules. <br />The <code>webNavigation
                </code> permission is required by the <code>chrome.webNavigation
                </code> module."
            }
        ],
        "web_entry_vulnerability": {
            "afcbjmdkfcdfbamemeadbpmabohjehcl/settings.js": [
                {
                    "web_entry_points": "chrome.runtime.onMessage Usage",
                    "Description": "<code>chrome.runtime.onMessage</code> is fired
                    when a message is sent from either an extension process
                    (by <code>runtime.sendMessage</code>) or a content script
                    (by <code>tabs.sendMessage</code>)."
                }
            ],
            "afcbjmdkfcdfbamemeadbpmabohjehcl/contentScript.js": [
                {
                    "web_entry_points": "chrome.runtime.onMessage Usage",
                    "Description": "<code>chrome.runtime.onMessage</code> is fired
                    when a message is sent from either an extension process
                    (by <code>runtime.sendMessage</code>) or a content script
                    (by <code>tabs.sendMessage</code>)."
                }
            ],
            ...
        },
        "dangerous_function_call": {
            "afcbjmdkfcdfbamemeadbpmabohjehcl/settings.js": [
                {
                    "Dangerous_function_call": "jQuery .append()",
                    "Description": "<code>.append()</code> when combined with user
                    input can lead to Cross-site Scripting (XSS) vulnerabilities.
                    When the CSP directive <code>'unsafe-eval'</code> is used unsafely
                    in conjunction with this function,
                    XSS is possible."
                }
            ],
            ...
        }
    }
}
```