Securing Chrome extensions

Henrik Kultala, Andreas Kärrby

June 2021

Contents

1 Background

Extensions are programs written by third-party software developers. These programs run in the browser, and work to dynamically alter the browsing experience for the user [1]. Among other things, they offer cosmetic things like custom cursors; utility-oriented things like integrated TODO-lists and reminders to drink water; and even full-blown minigames like Minesweeper or Chess.

From a security perspective, extensions pose an interesting challenge. They run with elevated privileges which allows them to do things that web pages are not allowed to (exactly what these things are will become clearer later in this report). Because of this, a lot of work has gone into securing them. One such security measure is that extensions run in separate contexts from websites, which means they cannot directly access each other's data. Furthermore, the extensions themselves are split up into different components, and they too execute in separate contexts.

The components we will be concerning ourselves with are:

- **Background scripts**, which have full access to all privileges that the extension has been granted
- **Content scripts**, which is the only type of component that has direct access to read and modify the DOM of the webpages that the extension is modifying
- **UI pages**, which constitute the interface for the user to interact with the extension (through actions like right-clicking to get a context menu on a page, or clicking the extension icon in the browser toolbar to get a popup window)

However, while isolating all these extension components from each other (and also from websites) into different execution contexts serves as a security mechanism, they must also be allowed to communicate in order to carry out meaningful work. To do this, extensions use clearly defined message-passing API:s. These API:s differ in syntax somewhat depending on the types of the communicating components, but in general they work very similarly to each other. We will only consider "messages passed between components" as a primitive.[1]

When it comes to the privileges of extensions, they are granted on a "need-to basis". Extension developers declare a manifest file (*manifest.json*) which includes things like the name and version of the extension, along with the privileges it needs to carry out its tasks. Listing 1 shows an example manifest file. Most fields are self-explanatory. We can see that ***background.js*** is a background script (one of the aforementioned types of components), and ***content.js*** is a content script (another of the types of components). The "content_scripts" field also contains the "matches" parameter, which designates the URL:s at

[1] The interested reader can read more about these API:s at https://developer.chrome.com/docs/extensions/mv3/messaging/

which the content scripts will execute. Extensions that want to run on every webpage a user visits will contain a match pattern with wildcards.

For the permissions, the "storage" permission allows the extension to store data locally in the browser of the user, the "contextMenus" permission is needed to modify context menus (when right-clicking a page), "cookies" is needed to read and modify user cookies, and "downloads" allows the extension to download data to the user's machine. This list of permissions is by no means complete, but rather serves to show the kinds of elevated privileges that extensions can be granted.[2]

Listing 1: Example *manifest.json* file

```
{
  "name": "The name of the extension",
  "description": "An short description of the extension.",
  "version": "0.0.1",
  "manifest_version": 3,
  "background": {
    "service_worker": "background.js"
  },
  "permissions": [
    "storage",
    "contextMenus",
    "cookies",
    "downloads"
  ],
  "action": {
    "default_popup": "popup.html"
  },
  "icons": {
    "16": "/images/logos/logo16.png",
    "32": "/images/logos/logo32.png",
    "48": "/images/logos/logo48.png",
    "128": "/images/logos/logo128.png"
  },
  "content_scripts": [
    {
      "matches": ["*://*/*"],
      "run_at": "document_idle",
      "js": ["content.js"]
    }
  ]
}
```

[2] A comprehensive list of available permissions can be found at https://developer.chrome.com/docs/extensions/mv3/declare_permissions/

Now, as one might imagine, their elevated privileges make extensions prime targets for malicious actors. There are several possible attacker models. One is where the attacker is the author of a malicious browser extension. Under the guise of "functionality", an attacker may request permissions which may seem benign for the supposed purpose of the extension, while in reality they can be used for things such as stealing user data or posting spam to social networks [2].

Another model is the network attacker. For example, if an extension loads any script from a web server through HTTP instead of HTTPS, a network attacker can intercept and replace this script with arbitrary code of his or her choosing [3].

A third model, and the one we will be focusing on, is the web attacker. In this model, extensions are "benign-but-vulnerable", and can be exploited by malicious web applications to carry out privileged tasks on the web application's behalf (a sort of *confused deputy*-attack).

In the following subsections, we provide some specific technical details which are related to the vulnerabilities presented in this report.

1.1 HTTP Referer header

In the article "Extended Tracking Powers: Measuring the Privacy Diffusion Enabled by Browser Extensions", Starov and Nikiforakis discuss accidental information leakage in Google Chrome extensions through the "HTTP Referer" header [4]. When extensions request resources, such as images or scripts, from third-party websites, they make HTTP-requests. Among other things, these requests contain the name of the "referer" site which the requests go through. This is the site that the user of the extension is currently visiting.

This means that if an extension loads on every page the user visits, and if it requests resources from a third-party website every time, all websites that the user visits will be indirectly announced to the third-party website because of the "referer" header. If this third-party site is malicious, it might make use of the other information of the HTTP-requests to link a specific user to the visited websites and thus construct the user's browsing history. This could then for example be used for targeted advertisements.

1.2 Same Origin Policy (SOP)

Websites are subject to something called the *Same Origin Policy* (SOP). This serves as a means of protection, preventing websites from reading the responses of arbitrary HTTP-requests sent to each other. More concretely, if a user is logged in to for instance their bank account in one tab, and then opens a malicious website in another tab, then the SOP prevents the malicious site from reading data from the bank account. The reason why this attack would work to begin with (if SOP didn't exist) is because when a request is sent (from the malicious site or other), all session cookies are automatically included, and thus the request will seem to be coming from the logged in user [5].

This, however, does not apply to extensions: they are not subject to the SOP [6]. The only thing preventing an extension from reading data from a particular website is its permissions. But while an extension might not state that it wants to read data from all possible URL:s, it might state in the "matches" parameter (described previously in section 1) that it wants its content script to be able to run on all websites. Based on the experience of the authors of this report, this results in the extension getting access to all websites as if it had specified such a permission. Thus, even without explicitly stating a "permission" to read data from any website, an extension might get this privilege simply by asking to run its content script on all websites.

To prevent malicious extensions from running rampant, there is a vetting process for all extensions that are submitted to the Chrome Web Store [7]. However, even with assuming that one manages to stay clear of malicious extensions, being exempt from the SOP means one might encounter benign but vulnerable extensions. These might be able to get through the scrutiny meant to catch malicious extensions because they are in fact not malicious. All it takes is that an extension allows a website to make a request to an arbitrary URL and that the extension then forwards the response back to the original website.

2 Goal

The goal of our project is to explain the basics of extension vulnerabilities to a target audience that is comfortable with programming, but that do not have much security training. The purpose is to raise "security awareness" for developers who may be interested in creating browser extensions. We chose the web attacker model with "benign-but-vulnerable" extensions specifically because of this goal.

To limit the scope, we also chose to only look at Chrome extensions, but the API:s used for developing browser extensions are rather similar for other vendors as well (such as Safari, Edge, and Opera), so the vulnerabilities discussed herein are most likely also relevant for these other browser platforms.

The "deliverables" are i) a GitHub repo with the vulnerable extensions we produced, along with README:s of how to run them, and ii) this report with more technical details, theoretical background, and finally possible remedies for how to patch the vulnerable extensions.

3 Description of work

Based on the literature study performed (with the findings reported in section 1), we brainstormed scenarios where one might accidentally create a vulnerable extension. We made an effort to try to only consider cases that one might actually encounter in reality. That is, we did not consider clearly malicious extensions as something that might "accidentally" be vulnerable.

Overall we were presented with two challenges when implementing vulnerable extensions. First of all we had to confirm that the vulnerabilities encountered during our literature study were still relevant and that we could replicate them. Some of the articles we read were a couple of years old meaning there might be new security measures in place preventing them. Furthermore, the articles generally didn't provide concrete examples of vulnerable code but rather pointed to real vulnerable extensions that might have been patched since the article was published.

Secondly, when we managed to implement a vulnerability we had to remember to create the surrounding extension in such a way that it was somewhat realistic to fulfill our goal outlined in section 2. While one could argue that our target audience of "non-security experts working in the computer science field" might not be aware of the most obvious vulnerabilities, we preferred creating examples that could be useful for developers that might have heard of some basic security concepts. For instance, an extension that does nothing else than allowing bypass of the SOP for websites it is active on might realistically be created by a security–unaware developer. But more interesting in our opinion is an extension where the developer might be aware of the SOP but might not realize that their extension allows a bypass of it because of the developer's lack of experience or sloppiness.

Following is a brief description of our work, divided into subsections for each of the vulnerable extensions we implemented. All code can be found at https://github.com/andreaskth/securing-chrome-extensions/.

3.1 Tracking by content provider

In exploring the HTTP Referer header leakage described in subsection 1.1, we decided to make an extension that loads images into every page a user visits. This extension loads all the images it displays from a third-party web server that we set up. We made this web server print all the requests it received. We could then confirm that the HTTP Referer header was present and contained the name of the website that the extension was injecting images into. While only the host name, the domain name and the scheme was present, this still is enough to violate the privacy of the user.

Inspecting the request more closely, we also found that it sent the IP address of the user with the request (in the *X-Forwarded-For* header). Figure 1 shows an example excerpt of what the image provider server might receive. Putting these together, the web server could construct the browsing history (the domains visited) of a particular IP address without the consent of the IP address owner; as long as they had the extension activated.

The code for this vulnerability can be found at: https://github.com/andreaskth/securing-chrome-extensions/tree/main/tracking-vuln.

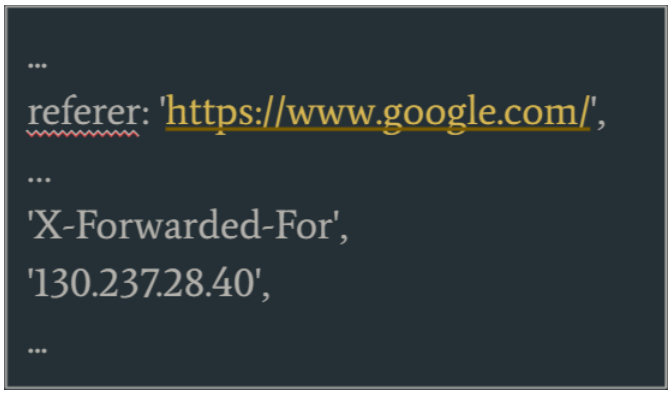

Figure 1: Excerpt of an HTTP-request sent to an image provider. The output shows that a user with IP address 130.237.28.40 has visited `https://www.google.com`

3.1.1 Possible remedies

The first thing to consider is to not load the images from a third-party provider. This would eliminate the issue altogether, and it is easy to accomplish: extensions allow images to be bundled with them. If it is necessary to fetch the images from a third-party provider, it is notably trickier to solve the leakage.

First of all one has to make the HTTP-requests in a more explicit manner to increase the control over them. This means that you no longer set the *src* attribute of an `<img>` tag; instead you could use for example the fetch API. When sending the request, you want to set the $referrerPolicy$ to "no-referrer" to prevent the "referer" header to be set.

However, this is not sufficient. Since we changed our means of getting the image, we introduced another problem: now the "origin" header is set in our request. This header is similar to the "referer" in that it will give away what website you visit. To fix this, we have to send our HTTP-request through a proxy. This could perhaps be done through the background page of the extension, but this proved difficult to get working, so in our fix we instead set up a simple proxy server.

We sent the image URL as an argument to this server and had it redirect our request to the supplied URL. By doing this we managed to fetch the image with the image provider getting neither a valid "referer" nor "origin" value. The image provider could still see our IP address since we hosted the proxy server locally, but without the name of our visited websites it could not learn our browsing history anymore.

3.2 Fetching from unsafe content provider

In this vulnerability, the extension in question will fetch some content from a third-party content provider that it trusts, and will inject the received content

without sanitizing it. This is a problem because the content provider may be malicious, or may itself have been compromised to return malicious data.

For our example, the extension will fetch HTML content from a third-party provider, and inject it into a div right above any HTML element with an *id* of "content". Then, we have a web server emulating this content provider. In its "benign mode", it will simply return innocuous content (Figure 2), but in its "hacker mode" (representing a malicious – or at least compromised – content provider), it will return an image with a malformed *src* attribute, and an *onerror* attribute which will send the user's cookies to the hacker's URL (Figure 3).

The code for this vulnerability can be found at: https://github.com/andreaskth/securing-chrome-extensions/tree/main/unsafe-fetch-vuln.

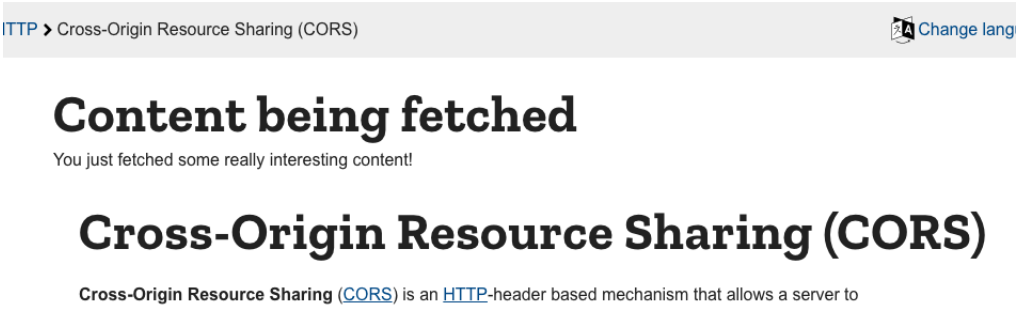

Figure 2: Content returned from a benign content provider is injected before a div with $id = content$.

```
Andreass-MBP:unsafe-fetch-vuln andreaskarrby$ node web-server/app.js
Example app listening at http://localhost:3001
New request to /fetch
New request to /hacker: preferredlocale=es; lux_uid=162220423964161907
```

Figure 3: Cookies are leaked to a malicious server due to a malicious or compromised content provider.

3.2.1 Possible remedies

First of all, one should generally try to minimize the amount of code that is fetched from other sources, since every piece of such code is a potential place to mount an attack. For the times where this is required, however, one should always take great care to sanitize what is received. Oftentimes the programmer will have an idea of what (s)he expects to fetch, so the code could be written to reflect that. For example, if they expect to just receive content with `<div>`, `<main>`, and `<article>` tags, it could make sense to remove all other tags from the response.

Finally, even when fetching content from other sources, it would be best to

fetch it as text, and take care of the HTML-part of the content locally in the extension code, which is under the control of the programmer.

3.3 Bypassing SOP

The Same Origin Policy is violated when a website can make an HTTP-request to an arbitrary URL and read the response. An extension that allows a website to make such requests and then also returns the response would thus be vulnerable to SOP bypasses, because of its privileges described in section 1.

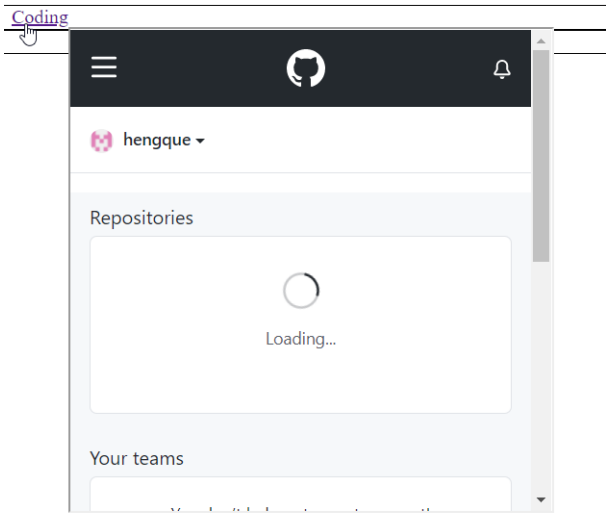

Figure 4: Extension showing a preview when mousing over a link to `https://www.github.com`

We implemented an extension that shows previews of websites when you mouse over hyperlinks (like on Wikipedia); see Figure 4 for an example of what this might look like.[3] The trickiest part with this implementation was trying to come up with something that wouldn't be too obviously vulnerable. In the end we "justified" our vulnerable extension by having it allow the websites that used it to make decisions regarding the content returned to them. In other words, the websites made requests that our extension handled and optionally they could ask the extension to let them look at the responses and act according to their contents.

We set up a malicious website that made requests to sites (containing sensitive information) that the user might be logged in to, and then asked the

[3]The extension is not fully functional, with the previews being quite broken. Since we had difficulties we focused our efforts elsewhere since well functioning extensions wasn't the main goal of this project.

extension to let it inspect the responses. In this way, our malicious site could abuse the extension to steal data from the sites the user was logged into.

The code for this vulnerability can be found at: https://github.com/andreaskth/securing-chrome-extensions/tree/main/SOP-vuln.

3.3.1 Possible remedies

The fix here is quite simple: just disallow the websites to read the content of the previews. This means that we would remove the functionality of having the websites make decisions based on the content that would appear in the previews, but this is necessary for preserving the SOP. One could perhaps try to enforce that a website only gets to read the content of a link pointing to a page under the *same origin*, but this doesn't sound very useful since you probably already know what those pages contain. Besides, you can still retain the functionality of allowing a website to state links it doesn't want a preview of; the only thing removed is allowing the website to make this decision after reading the content that would be shown in the preview.

4 Reflections

As the above sections have shown, writing extensions in a secure manner is not always easy. The SOP bypass is perhaps somewhat more forced than the rest, but we still hold that all vulnerabilities could very plausibly end up in "real" extensions in one shape or another. Some of it stems from developer unawareness (such as not sanitizing third-party content), whereas some risks could reasonably be mitigated by the platform itself (i.e. Google).

For example, it would be good if fetching content from third-party content providers did not automatically disclose the user's browsing habits to the content provider. Circumventing this from the developer-side requires: i) knowing about the problem, and ii) finding other ways to fetch the content that do not disclose the user's privacy information. Since there were quite some hoops to jump through to get ii) to work, it would be good if this could work already "out of the box" when programming browser extensions.

In general, since extensions are built with standard web technologies, it is no surprise that many vulnerabilities associated with the web may also appear in extensions. However, it is important to understand that the extension security model is slightly different than that of web applications, and therefore you can never assume that security measures that exist in one model will automatically exist in the other.

To summarize, extensions come with their own risks and pitfalls, and their elevated privileges make them prime targets for attackers. Extension developers can reduce, as much as possible, the amount of privileges they request (in the manifest.json file), but even extensions operating with "bare minimum" permissions can cause a lot of problems if care is not taken to secure them.

Finally, it is worth remembering that both malicious and vulnerable extensions can slip through the vetting process that extensions undergo before being published to the Chrome store (as several of the papers referenced in section 1 have shown), so for the end user, it can unfortunately be quite difficult to know which extensions are safe, and which ones are not.

References

[1] Google. [n. d.] Extensions. Chrome extensions documentation for developers. (). https://developer.chrome.com/docs/extensions/.

[2] Alexandros K. et al. 2014. Hulk: eliciting malicious behavior in browser extensions. In *Proceedings of the 23rd USENIX Security Symposium*. usenix, 641–654.

[3] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An evaluation of the google chrome extension security architecture. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*, 97–111.

[4] Oleksii Starov and Nick Nikiforakis. 2017. Extended tracking powers: measuring the privacy diffusion enabled by browser extensions. In *Proceedings of the 26th International Conference on World Wide Web*, 1481–1490.

[5] Wikipedia. [n. d.] Same-origin policy. (). https://en.wikipedia.org/wiki/Same-origin_policy#Read_access_to_sensitive_cross-origin_responses_via_reusable_authentication.

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

[7] Google. [n. d.] Publish in the chrome web store. (). https://developer.chrome.com/docs/webstore/publish/.

A Statement of contributions

Initially we both read the article "EmPoWeb: Empowering Web Applications with Browser Extensions" which we used as a basis for our project. Henrik then followed up on some of the references in that article while Andreas focused on documentation about extensions.

Putting our garnered knowledge together we worked together on coming up with possible vulnerable extensions. Andreas developed the extension for the HTTP Referer vulnerability (the "Easter Eggstension") while Henrik created the malicious website that abused it. Henrik also wrote the README for this vulnerability and implemented a fix.

For the unsanitized fetch vulnerability, Andreas wrote the extension and the malicious website, while Henrik helped debugging it and making it work. Andreas also wrote the README for this vulnerability.

For the SOP bypass vulnerability, Henrik wrote the extension and the website while Andreas helped during the conceptual stage to make it somewhat realistic. Henrik also wrote the README for this vulnerability.

Andreas set up and structured the code repo and wrote the majority of the main README. He also structured the report by creating a skeleton.

Regarding the report, Andreas wrote:

- the "Background" intro part (i.e. section 1 but not subsection 1.1 or subsection 1.2)
- section 2 about our goals
- subsection 3.2 about implementing the vulnerability of fetching resources from a malicious content provider
- section 4 with our reflections

while Henrik wrote:

- subsection 1.1 about the HTTP Referer header
- subsection 1.2 about the Same Origin Policy
- the "Description of work" intro part (section 3)
- subsection 3.1 about implementing the vulnerability of a content provider tracking users
- subsection 3.3 about implementing the vulnerability of an extension allowing bypass of the SOP

We also both proof-read the entire text and provided suggestions for each other's parts.