6.857: Computer and Network Security Due: May 16, 2018

Security Analysis of Firefox WebExtensions

Srilaya Bhavaraju, Tara Smith, Benny Zhang
srilayab, tsmith12, felicity

Abstract

With the deprecation of Legacy addons, Mozilla recently introduced the WebExtensions API for the development of Firefox browser extensions. WebExtensions was designed for cross-browser compatibility and in response to several issues in the legacy addon model. We performed a security analysis of the new WebExtensions model. The goal of this paper is to analyze how well WebExtensions responds to threats in the previous legacy model as well as identify any potential vulnerabilities in the new model.

1 Introduction

Firefox release 57, otherwise known as Firefox Quantum, brings a large overhaul to the open-source web browser. Major changes with this release include the deprecation of its initial XUL/XPCOM/XBL extensions API to shift to its own WebExtensions API. This WebExtensions API is currently in use by both Google Chrome and Opera, but Firefox distinguishes itself with further restrictions and additional functionalities. Mozilla's goals with the new extension API is to support cross-browser extension development, as well as offer greater security than the XPCOM API.

Our goal in this paper is to analyze how well the WebExtensions model responds to the vulnerabilities present in legacy addons and discuss any potential vulnerabilities in the new model. We present the old security model of Firefox extensions and examine the new model by looking at the structure, permissions model, and extension review process. We then identify various threats and attacks that may occur or have occurred before moving onto recommendations.

2 Background

Because Firefox Quantum's latest addon model was released recently in November 2017, very little research has been conducted on the model. However, there are a number of papers that display the weaknesses in Firefox's old versions, and even Mozilla has commented that their old system had several major insecurities. With Firefox Quantum and its WebExtensions API, Mozilla has attempted to fix a majority of the issues found in previous versions of Firefox.

WebExtensions was first announced in 2015, when only validated and signed extensions could be published. The first version of WebExtensions was released in 2016. In 2017, support for legacy addons was terminated, and currently only WebExtensions are supported. With its cross-browser compatibility, WebExtensions supports both the "chrome" and "browser" namespaces, as well as both callbacks and promises.

2.1 Terminology

- Addons/Extensions - Code that can be downloaded in the form of a small application to enhance a user's browsing experience.
- Legacy Addons - Addons that use the old XPCOM API.
- Manifest Files - These describe the package and map its location on disk to a chrome URL. They are examined when a Mozilla application starts up to see what packages to install.[1]

2.2 Firefox Legacy Addons

Firefox Legacy Addons had access to the full set of privileged XPCOM APIs and Javascript code modules, as well as the browser's internal UI implementation.[2] They could directly manipulate the XUL API to customize the user interface. While this permitted for a great deal of customization regarding Firefox, it also led to many insecure practices and opened the door for malicious code injections.

Legacy Extensions required two manifest files: `install.rtf` and `chrome.manifest`. Because Legacy Extensions had higher permissions than necessary, Firefox implemented an isolation mechanism in attempt to keep developers from accessing privileged information. They implemented a "sandbox" to isolate some of the JavaScript files included, but this sandbox could be disabled by developers with ease. Firefox also did not require extensions to be signed until 2015, which meant that it was difficult to know which extension could be trusted, as not all trustworthy extensions were signed.

Another issue with Legacy Addons was the tight coupling. Because extensions had access to the internal implementation, developers would use specific patterns to find and insert or replace code snippets. This often led to broken extensions when Firefox updated its backend and tended to delay Firefox development.[2]

2.3 Firefox WebExtensions Addons

With the WebExtensions API, only one manifest file (`manifest.json`) is required. The WebExtensions API no longer has direct access to the Firefox user interface. Instead, the API includes methods, such as browserAction, to help an extension access information about the interface. Furthermore, extensions gain limited access to the JavaScript API through background scripts and see a "clean" version of the DOM objects in a normal web page through the window global functionality.[18] This will be elaborated in Section 3.

The use of Firefox can be considered in three main components: Firefox, web pages, and addons. We assume Firefox to be trustworthy; in other words, we believe it contains no malicious code. Web pages have content that is loaded by Firefox so that users can see the web page. We assume web pages can be insecure and potentially have malicious code in their scripts. We also consider extensions to potentially be untrustworthy.

Firefox has `addon.mozilla.org` (AMO) as their official addon site. Official addons can be uploaded onto the AMO site after being verified and reviewed by Mozilla. To be considered an official addon and be published on the AMO website, Mozilla runs the extension through a series of automated computer tests before human examination of the code for any signs of malicious behavior. The human reviewers are required to sign their extension when they publish an extension.

The sandbox isolation that was present with Legacy Addons has been maintained and strengthened with the creation of WebExtensions to limit the access between different sections of Firefox.

3 Firefox Security Model

As previously noted, former versions of Firefox contained very little security surrounding Addons.[3] Addons did not have to be signed, malicious code could be easily uploaded, and the vague attempt at security using JavaScript sandboxes could be disabled by developers of the addon. In short, there was little to no protection against malicious or vulnerable extensions. In 2015, Mozilla stated that it would no longer allow unsigned extensions to run, thus addressing one of the problems.[4]

3.1 Security Sandboxing: Isolating Extensions

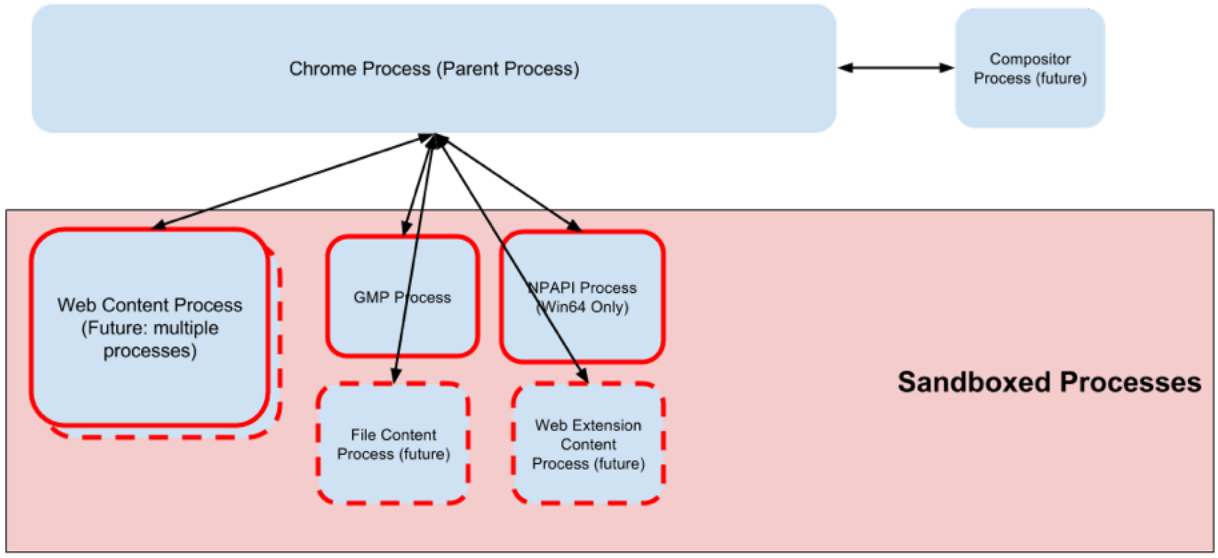

Figure 1: Firefox sandbox model. The Chrome (parent) process is isolated from its child processes.

Firefox itself uses a method known as Security Sandboxing to keep itself secure. While the exact details differ between platform/operating systems, the general idea is the same: there exists a parent process that has access to the underlying operating system. The parent process has child processes that are placed in sandboxes. These sandboxes each have a distinct set of permissions that are meant to restrict the child process's access to APIs, the browser, and other processes.[5] Figure 1 is a visual representation of this model.

Presently, Firefox has the following child processes: Web Content, Gecko Media Plugin (GMP), and NPAPI. Web Extensions did not have their own distinct restrictions until Firefox v55, but they were still restricted to some level because they were considered a child process and not completely trustworthy.[6][7] It is noted several times on Mozilla's website that they will eventually load in their own separate processes to ensure their isolation.

3.2 The Anatomy of Extensions

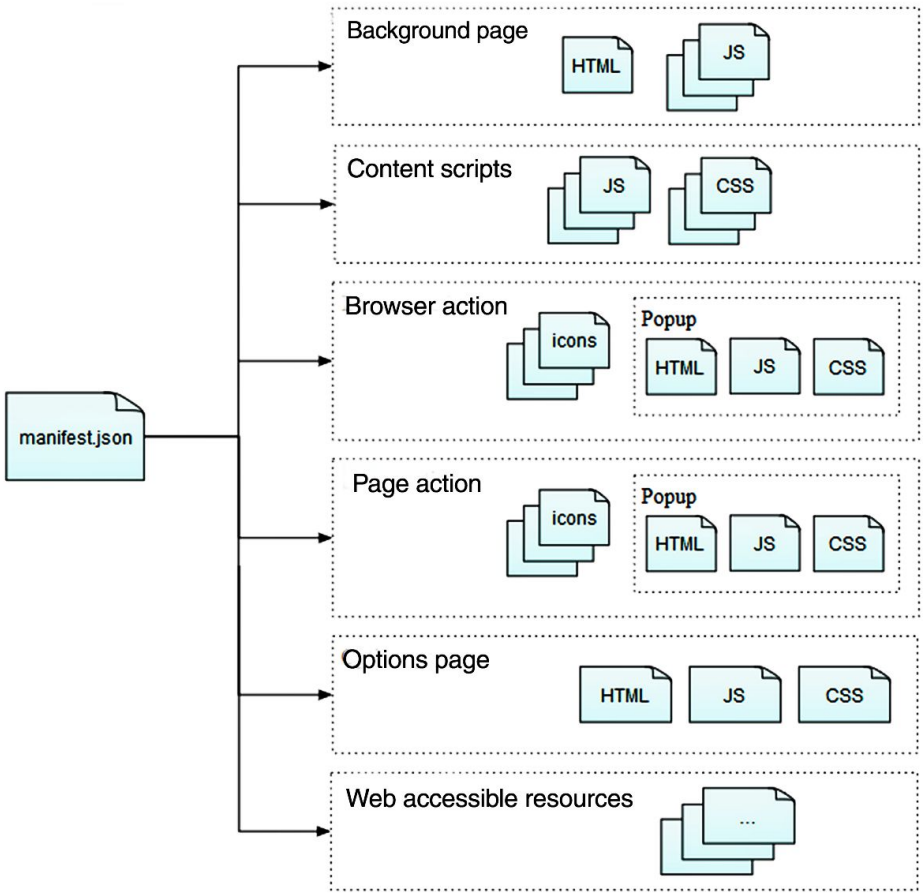

Figure 2: Anatomy of extensions. The manifest.json points to the many files that the addon will use/need to run.

Firefox extensions may contain background scripts and content scripts. Background scripts exist for long-term storage, are created upon installation, and persist until the extension is disabled or uninstalled.[8] They have full access to the WebExtensions API given the corresponding permissions and can communicate with the content scripts. Content scripts run in the context of the page and can access and manipulate web pages to an extent. They can view the DOM, as well as make cross-domain XHR requests, and have limited access to the WebExtension APIs. They can also communicate with background scripts loaded by the extension through a messaging API. While content scripts cannot directly access page scripts loaded by the web page, they can exchange messages with them.[18] Extensions can also include sidebars, popups, and option pages, which are HTML documents that provide content for user interface components.

3.3 Permissions Models

In previous versions of Firefox, addons by default had the same level of permissions as the browser. It was discovered that a majority of Firefox addons were over-permitted; many of the extensions did not require the full library of code that they could access. In a study of 25 addons, 3 required permission to run arbitrary code/access files on a user's computer, but all 25 of them could do so.[9]

Firefox's WebExtensions' new permission system is very similar to that of Google Chrome. An addon developer must explicitly state which permissions are required for their addon to run on Firefox, and these permissions are then revealed to the user during installation. If these permissions are not given in the manifest.json file, they cannot be used.

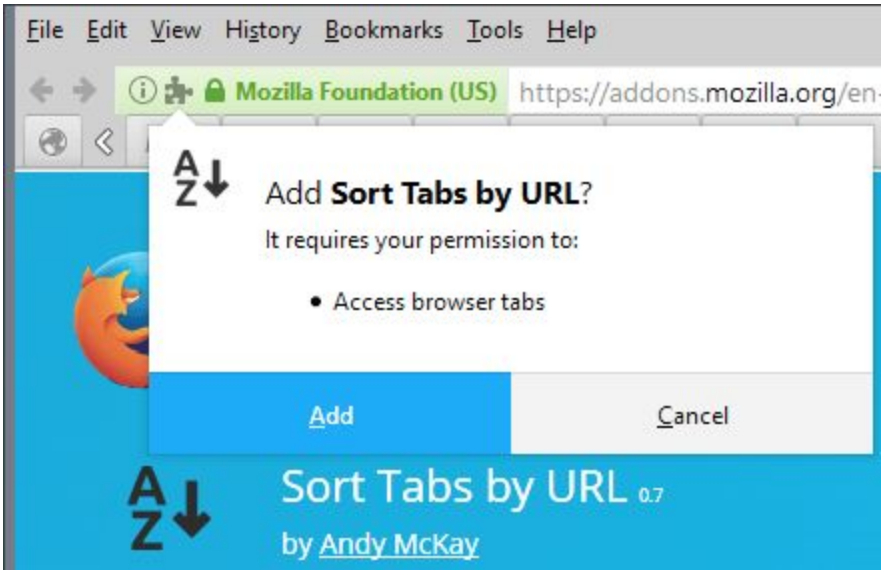

Figure 3: An extension requesting permissions[10]. This is a newer feature for Firefox and is supposed to be even more detailed so that the user can be informed/warned when deciding to download an addon.

According to Firefox's site: "If you request permissions using this [the permissions] key, then the browser may inform the user at install time that the extension is requesting certain privileges, and ask them to confirm that they are happy to grant these privileges. The browser may also allow the user to inspect an extension's privileges after installation."[11] Permissions that must be stated are any access to the WebExtensions API (API permissions), access to the active tab (activeTab permissions), and access to certain hosts/websites (Host permissions). The three different permissions are described in further detail below.[11]

- Host Permissions: Host permissions are specified through patterns that match to a group of URLs. The WebExtension is given extra privileges if the pattern matches the URL. An example of a host permission is "`*://developer.mozilla.org/*`" The extra permissions allow for XMLHttpRequest and Fetch access, the ability to inject scripts via tabs.executeScript, the ability to receive events from the webRequest API, the ability to access cookies for the specified host (so long as the "cookies" API permissions is also included), and the ability to bypass tracking protection (so long as there are no wildcards in the pattern).

- API Permission: These permissions allow the WebExtension to access certain libraries of the WebExtensions API. Some examples of the libraries include bookmarks, sessions, and tabs. It is noted in the Firefox website that there are exceptions, but including these permissions simply allow access to the API.
- activeTab Permissions: This makes it so that the extension is granted extra privileges only for the active tab when the user interacts with the extension. A user must click on the extension's action, select its context menu item, or activate a keyboard shortcut defined by the extension in order for the extension to have these permissions. Extra permissions included with the activeTab permissions include the ability to inject JavaScript or CSS into the tab and the ability to access the privileged parts of the tabs API only for the current tab.

There is also an "Optional Permissions" key in the json file. These permissions are not required upon installation, but they can be requested at some point while the extension is running on Firefox. Optional Permissions can consist of API Permissions and Host permissions only.[12]

3.4 Firefox Addon Review System

Firefox addons can be developed and distributed by anyone. The official Firefox library of extensions, however, is located at addon.mozilla.org (AMO). The AMO website's review process for Legacy Extensions was extremely lengthy, as consisted only of manual review of code within an extension for any malicious indicators.[13]

With Firefox WebExtensions, the AMO has modified their review process. There is an automated review that forces the addon to pass a number of tests in a sandboxed environment for general security. Following this is a manual review of the app. Developers who submit their extensions for review are required to send in human-readable code for the review. The addon is reviewed against the policy in place for addons.[14]

In between the automated check and the human-review, the addon is free to exist on the AMO site and available for download if it passes the automated review.[13] This potentially presents an opportunity for malicious addons to exist on the site if the automated review does not catch them. There have already been several reported issues regarding this.[13] However, Firefox addons can be removed from the AMO site if the human-reviewer finds anything wrong with the addon. They can even be blocklisted if it severely violates the AMO policy on extensions.

4 Threats & Vulnerabilities

Mozilla's deprecated XUL/XPCOM based extension API was vulnerable to several different threats. WebExtensions has responded to many of these threats through tightened permissions and a sandboxed model approach. Even so, WebExtensions still has potential for certain vulnerabilities. Some of these come from the extension signing model, while others exploit the similarity of WebExtensions to Chrome's extension framework. This section will provide an overview of some of the threats in the legacy model that WebExtensions responds to as well as some threats that it may still be vulnerable to.

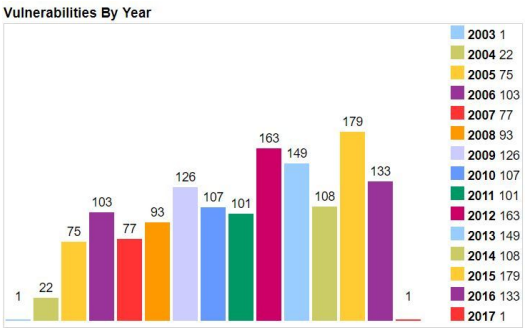

Figure 4: Vulnerabilities Discovered in Firefox Per Year

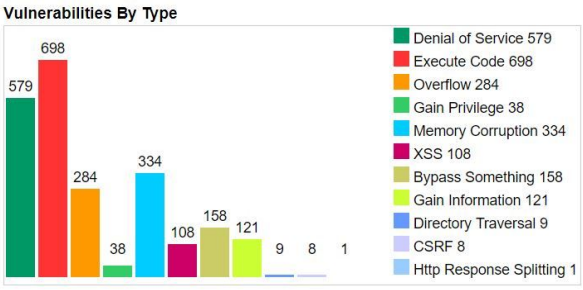

Figure 5: Vulnerabilities Discovered in Firefox By Type

With the release of WebExtensions and Firefox Quantum, the number of vulnerabilities exploited in Firefox has drastically decreased, as seen in Figure 4. In the previous Legacy model, there was less isolation between the addons and Firefox's internal code. This probably explains the large number of arbitrary executions of code errors. With the sandboxed model in the new WebExtensions API, it is more difficult for extensions to gain access to Firefox's internal code. The model's enforcement of permissions ensures this. These changes may have contributed to the drastic decreases, for example, in detected "Execute code" vulnerabilities.

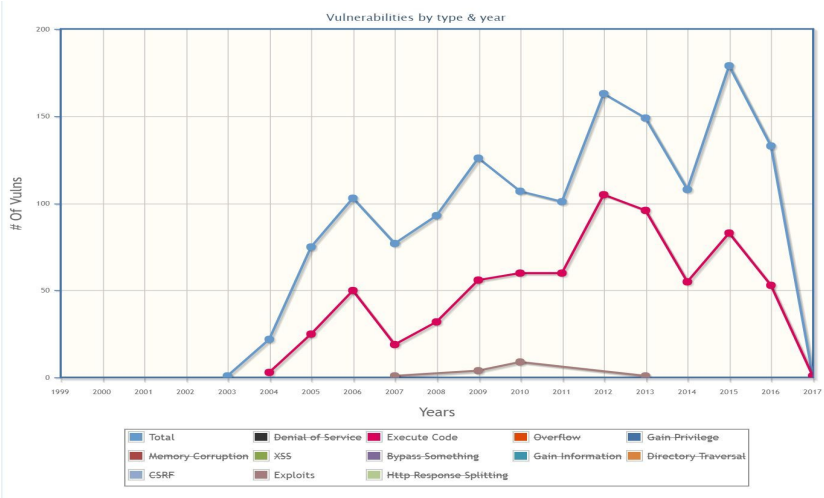

Figure 6: Vulnerabilities Discovered in Firefox by Type and Year

4.1 File System Access

The sandbox and new restrictions to the File and Directories Entries API is meant to prevent accessing local files directly. In order to be sure of this, Mozilla aims to prevent privilege escalation through strong trust models and IPC implementation. Firefox's current testing involves review of the components of Firefox to make sure their security model is strong enough. They also plan to audit and fuzz test IPC mechanisms to communicate between content and parent processes.[5]

In Mozilla's previous XPCOM extension model, extensions were allowed access to the file system and other contexts outside the browser through the API. However, with Firefox WebExtensions this capability has

been removed. Files can still be read from an extension, but the specific file directory must be specified in manifest.json.

There is, however, a non-standard API based off of Chrome's implementation: the File and Directories Entries API. With this, it is possible to develop applications that read and write directories in a virtual, sandboxed file system. Firefox's implementation does not support creating files. Only files that are selected by the user by input and files that are "dragged and dropped" into the web app can be accessed.

Firefox warns against using the File and Directories Entries API in "production sites facing the web". This implies that Firefox recognizes the threat of compromised content processes by malicious webpages. However, it does not provide any defense except for warning developer against using it in certain extensions. If the AMO review process is robust enough, extensions that use the API for "web-facing applications" should be rejected by the AMO review process. However, it is unclear how the process would detect whether an application could be "web-facing" unless indicated by the developer.[41]

4.2 Rogue Extensions

Recently, rogue extensions have been discovered in both Firefox and Chrome. These extensions are able to hijack browsers and block their own removal. In Firefox, a fake message claiming a required manual update for Firefox tricks users into installing the extension. The extension blocks "about:addons" in its background.js by searching for the string in the URL and closing the tab if the string is found. "About:addons" is where the user can manage all of their extensions, so the rogue extension prevents manual removal of itself. The only way to counter the extension, once it is installed, is to run Firefox in "safe mode". "Safe mode" runs firefox with extensions disabled. The extension can be removed in safe mode, a functionality not available in Chrome.

```
if(null!==b&&b!==void 0)return b},{urls:['<all_urls>']},['blocking'])}
catch(a){}}(),function(){try{function a(){browser.tabs.query({}).then(
function(a){var b=a.find((a)=>a.url===['ab','ou','t:','ad','do','ns'].join(''));
```

Figure 7: Script to block removal of extension

Firefox's AMO review process may not catch such threats, particularly if they come in the form of side-loaded extensions or from third parties. Furthermore, once the "tabs" permission is requested and granted by the user, the extension is given privileged access to the tabs API. This enables it to access the URL information of open tabs. However, closing tabs is not a privileged permission, so an extension can do this without requesting the "tabs" permission. We think that WebExtensions should not by default grant access to certain tabs methods and should not automatically grant privileged access with the "tabs" permission. Furthermore, it should check for any type of blocking when the close tabs method is used.[17]

4.3 Bypassing the Review Process

As previously mentioned, Mozilla's AMO Review Process is a combination of automated review and manual review. However, it has been shown that the automated scanning of extensions can be easily bypassed.

Mozilla's extension signing plan uses automated review to determine whether the extension can be signed. The automated review mainly aims to combat malware that comes in the form of side-loaded extensions. AMO extensions are signed automatically after every review. Non-AMO extensions need to be signed for every update. In the past, only extensions that did not pass the review would be sent for manual inspection. However, a simple extension was created to bypass the automatic AMO validator. This extension monitors HTTP(S) requests for Basic Auth credentials and POSTs them to an arbitrary HTTP server. Then, it runs an arbitrary local process when a given URL is loaded. Finally, it downloads JavaScript code from a remote server and runs it with full privileges.[13][40]

4.4 TabHiding

The ability to hide tabs through the tabs.hide() API raises some security concerns. It is currently still an experimental API and is disabled by default. However, pinned tabs and currently active tabs cannot be hidden. Tabs that are in the process of closing cannot be hidden. Furthermore, once an entire window is closed, the hidden tab is also closed.

Nonetheless, hidden tabs that run without the user's knowledge can run intensive tasks in the background, track the user across different networks, spawn windows and ads, or make inappropriate network requests. They make use of Audio and WebRTC.

The goal of WebExtensions is to be more secure than legacy addons, but measures against hiding tabs are relatively sparse. While the `tabs.hide()` API is still experimental and requires separate permissions from "tabs", Mozilla leaves it up to "good faith" of the addon to provide information on what its hiding tabs do if it uses them. It is possible to display whether hidden tabs make use of audio, but it is difficult to tell whether the tab makes use of WebRTC. Mozilla claims that it is an extremely time-intensive process to check whether each addon bypasses the built-in security measures for hiding tabs. Currently with WebExtensions, the user is shown a list of hidden tabs, information normally provided by visible tabs (url and title), and a way to remove the hidden tab. They also claim to show indicators when a tab displays invasive behavior, such as WebRTC or audio.

A permission request for hiding tabs "tabsHide" is separate from a permission request for "tabs". Unlike other privilege requests, permission for "tabs" grants access to privileged parts of the tabs API. Much information about tabs can be accessed without the "tabs" permission. For example, an extension can access a list of opened tabs and open, update, move, reload, and remove tabs without explicitly asking for the "tabs" permission. An extension can even insert JavaScript and CSS into the tabs without the "tabs" permission if it has host permission for the tab. With the additional privileges, an extension can access the URL, title, and favicon of the tab.

4.5 Malicious Web Pages

Content scripts act as an intermediary between page scripts and background scripts. As previously mentioned, they run in the context of the particular web page. This is in contrast to background scripts, which are part of the extension, and page scripts, which are part of the web site itself (loaded through `<script>`). Background scripts cannot directly access the content of the web pages. If a background script needs to access content of web pages, the extension must include a content script to do so.

Content scripts can only access a small subset of the WebExtensions API. However, because they can still communicate with the background scripts through a messaging system, they can indirectly access other parts of the WebExtensions API that the associated background script has permission to access.[18]

Figure 8: Methods callable from content script

By default, content scripts do not have access to objects created by page scripts. However, they can still access these objects with `wrappedJSObject` as shown here. It is not possible however, to access objects created in the web page and passed to the page script this way. Mozilla also warns that once objects are accessed this way, the object's properties and functions are no longer reliable as untrusted code could have redefined the page objects. They recommend rewrapping objects after they are accessed.

Content scripts can still communicate with page scripts using the standard DOM API methods window.postMessage and `window.addEventListener` APIs. The limited communication of content scripts and page scripts is meant to protect from the case of malicious web pages so that the web pages do not obtain access to the WebExtensions API.

However, we found that it is still possible for a malicious web page to indirectly gain access to the WebExtensions API methods. This is especially possible if the content script uses `eval()`. When a content script listens for and receives a message from the page script, there is no restriction on what the message can be. Therefore, if a response action of the content script is to to evaluate the message (using `eval()`), then the page script (which normally has no access to the WebExtensions API) can indirectly run any code with the elevated privileges of the content script. A simple example of this is here. This problem can be escalated further, since the content script can indirectly access other API methods through the background script. Therefore, in theory a malicious web page could potentially have the ultimate privileges of a background script.

As `eval()` is generally considered a risky method to use, developers and reviewers need to make sure that the only way input into `eval()` can be received is they way that it is intended to be received (e.g. through user input). Developers should also note that specifying regex also does not necessarily help, since JavaScript could be written to match the regex.

To make implications of the `eval()` method clearer, the WebExtensions documentation states that `eval()`, which runs in the context of the content script, is separate from `window.eval()` which runs in the context of the page. `Window.eval()` should be used in place of `eval()` wherever possible. To protect against abuse of `eval()`, extensions developed with WebExtension APIs have a Content Security Policy (CSP) applied to them by default. Under this policy, the extension is not allowed to evaluate strings as JavaScript. Websites specify a CSP using an HTTP header from the server. This aims to specify legitimate sources of content, such as scripts and embedded plugins.[42] However, it has been shown that the CSP header can be bypassed.[39]

However, `eval()` is still a useful method that content scripts may want to use. Therefore, Mozilla may benefit from a tool (such as VEX) that catches and analyzes flow from a website through an `eval()`, whenever a loaded extension uses the method.[38] Furthermore, they can check if replacing the `eval()` with `window.eval()` changes the function of the extension.

4.6 Cross-Site Request Forgery (CSRF) Attacks

Cross-site request forgery attacks, which allow "man-in-the-middle" adversaries to perform unauthorized activities on a website as authenticated users, are constantly a concern for web browsers. CSRF attacks exploit the trust that a site has in a user's browser, by taking advantage of the fact that every request to a website includes cookies and these cookies are often used for authentication purposes.[23]

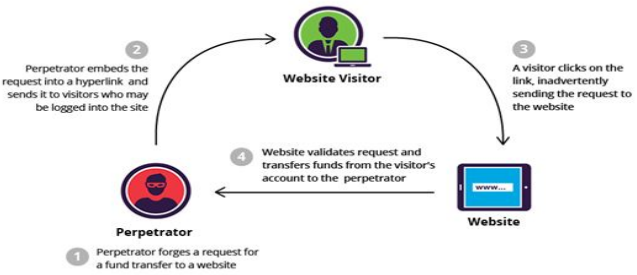

Figure 9: Cross-site request forgery flow

Mozilla states that it currently does not allow websites to determine whether a request is from the actual user or a third-party adversary. It claims that Firefox 60 will attempt to protect users against CSRF attacks through the use of same-site attributes. These have one of two values: strict or lax. In strict mode, cookies will not be sent when a user clicks on a link from an external site, so users are treated as unauthenticated, regardless of whether they have active session. In lax mode, cookies will be sent when users intentionally click on a link to an external website. However, exceptions include links that are cross-domain sub-requests, such as the ones made for images or frames, in which cases the cookies will not be sent. The lax mode is designed for applications that may be incompatible with strict mode.[31]

4.7 Bypassing Web Page and Background Communications

The WebExtensions API does not support the `externally_connectable` manifest functionality. In the Chrome extensions API, this allows websites to connect to extensions directly. In other words, it allows runtime.sendMessage API to be passed directly from the web page to the background script. Despite lack of support for this in WebExtensions, it is still possible for a website to communicate with the background script.

Since the content script cannot by default modify the site's initial DOM, it is not possible to define a function in the content script that communicates with the background script and inject it into the site. However, a workaround for this is to use two text areas, one of which is for responses and is located in the web page. Now, a message sent from the webpage will populate the text area. The content script can be defined to

identify when the text area is populated and then send the message in the text area to the background script. After this, the text area is cleared. Once the background script receives a message, it sends a response back to the content script. The content script writes the message into the textarea, where the webpage listens for it. The web page can now call a callback function with the message and clear the textarea once the message is parsed. An attempt to demonstrate this is here.[24]

5 Analysis

It is obvious that Mozilla took several measures to protect against the vulnerabilities found with legacy addons. The main measure taken is the change in permissions and the sandboxed approach to keep extensions separate from Firefox's internal code. However, other protections from known vulnerabilities mostly come in the form of warnings to developers about the use of certain methods. Furthermore, many APIs are still experimental, so it is not clear how strict Mozilla will be in allowing the use of these in extensions. Much of the safety of published extensions depends on the AMO review process and how strict the process is. However, given that the process is partially manual and partially automated, it is highly impossible to find all potentially vulnerable or potentially malicious extensions. This review process must particularly be wary of tab hiding, rogue extensions that prevent their removal, cross-script forgery attacks, and scripts that bypass the review process. Several tools have been designed to help track the flow of such potential attacks, so using these may be of benefit.

6 Security Best Practices & Recommendations

When it comes to extensions vetting, AMO should advance their automation and manual review process as attacks become more sophisticated. Tools, such as VEX, may help with tracking specific flows that lead to potential vulnerabilities (such as with `eval()`).[38] They should also make sure that addons are not posted on AMO without human review. Malicious webpages could modify certain functions within the extension and run its own code with elevated privileges. Therefore, the review process should be sure to restrict cross-site requests and injection of remote scripts from potentially untrustworthy code.

Sharing objects between page and content scripts in JavaScript should be done with care. Using Firefox's `wrappedJSObject` provides a way of sharing, but developers must remember to rewrap the object after access as well and be wary of the fact that sharing this way is potentially untrustworthy. If remote content is inserted, it should be inserted safely. In other words, arbitrary access to the DOM should be restricted and use of native DOM manipulation methods should be encouraged.

Where possible, extensions should use `window.eval()` instead of `eval()` in content scripts. Furthermore, extensions should not add buttons and other GUI elements directly to web pages. They should create their own UI instead. In addition, it is recommended that moz-extension paths are not directly injected, because such information could be used to fingerprint the user. AMO claims to detect changes to third party libraries and disable any extensions in which it finds such changes, so third party libraries should be used with caution.

In terms of the user, they should be wary of the permissions that they grant extensions as too many permissions increase the likelihood of malicious code execution. The should also keep software up to date to have the updated protections from vulnerabilities, connect to sites via HTTPS (which have a layer of encryption for security purposes), disable WebRTC functionalities, clear browser history and cookies frequently (so

sensitive information is not stored for too long), and use proxy extensions (as this prevents access to the user's specific IP Address).

7 Conclusions and Further Work

While public details about the AMO review process are relatively limited, it would be interesting to obtain more information about the specifics of the automated and manual review processes. Knowing this information would help to better determine whether the review process is robust against specific types of attacks. Future work should focus on developing extensions with particular vulnerabilities or malicious capabilities and checking if they pass the AMO review process. Additionally, Firefox 60 claims to protect against cross-site request forgery attacks, so this should be tested in the new release as well.

8 Appendix

All relevant code used to demonstrate and test some of the concepts can be found here.

References

[1] “Manifest Files.” *MDN Web Docs*, 11 Oct. 2014, developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Tutorial/Manifest_Files.

[2] Needham, Kev. “The Future of Developing Firefox Add-Ons.” *The Mozilla Blog*, 21 Aug. 2015, blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/.

[3] Verdurmen, Julian. “Firefox Extension Security.” *Radboud University*, 2008. www.cs.ru.nl/bachelorscripties/2008/Julian_Verdurmen___0413380___Firefox_extension_security.pdf

[4] Millman, Rene. “Mozilla Changes Security Model to Bolster Extension Protection.” *SC Media UK*, 25 Aug. 2015, www.scmagazineuk.com/mozilla-changes-security-model-to-bolster-extension-protection/article/534981/.

[5] “Security/Sandbox/Hardening.” *Mozilla Wiki*, 11 Apr. 2017, wiki.mozilla.org/Security/Sandbox/Hardening.

[6] “Security/Sandbox/Process Model.” *Mozilla Wiki*, wiki.mozilla.org/Security/Sandbox/Process_model.

[7] “Electrolysis.” *Mozilla Wiki*, wiki.mozilla.org/Electrolysis.

[8] “Anatomy of an Extension.” *MDN Web Docs*, Oct 19, 2017, 3:39:21 PM, developer.mozilla.org/en-US/Add-ons/WebExtensions/Anatomy_of_a_WebExtension#Background_scripts.

[9] Barth, Adam, et al. *Protecting Browsers from Extension Vulnerabilities*. 2009, *Protecting Browsers from Extension Vulnerabilities*, www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-185.pdf.

[10] Brinkmann, Martin. “Firefox's New WebExtensions Permission System - GHacks Tech News.” *GHacks Technology News*, Publisher Ghacks Technology NewsLogo, 6 Mar. 2017, www.ghacks.net/2017/03/06/firefoxs-new-webextensions-permission-system/.

[11] “Permissions.” *MDN Web Docs*, Mar 23, 2018, 3:33:52 PM, developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/permissions.

[12] “Optional Permissions.” *MDN Web Docs*, Apr 5, 2018, 5:20:16 PM, developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/optional_permissions.

[13] Brinkmann, Martin. “Mozilla Changes Review Process for Firefox WebExtensions.” *GHacks Technology News*, Publisher Ghacks Technology NewsLogo, 3 Oct. 2017, www.ghacks.net/2017/10/03/mozilla-changes-review-process-for-firefox-webextensions/.

[14] “Add-on Policies.” *MDN Web Docs*, developer.mozilla.org/en-US/Add-ons/AMO/Policy/Reviews.

[15] Arntz, Pieter. “New Chrome and Firefox Extensions Block Their Removal to Hijack Browsers.” Malwarebytes Labs, Malwarebytes, 18 Jan. 2018, blog.malwarebytes.com/threat-analysis/2018/01/new-chrome-and-firefox-extensions-block-their-removal-to-hijack-browsers/.

[16] Chipman, Ryan, et al. Security Analysis of Chrome Extensions. 2016, Security Analysis of Chrome Extensions, courses.csail.mit.edu/6.857/2016/files/24.pdf.

[17] “Tabs.” MDN Web Docs, Feb 13, 2018, 2:36:05 PM, developer.mozilla.org/en-US/Add-ons/WebExtensions/API/tabs.

[18] “Content Scripts.” MDN Web Docs, May 11, 2018, 6:34:50 AM, developer.mozilla.org/en-US/Add-ons/WebExtensions/Content_scripts.

[19] “Example Extensions.” MDN Web Docs, Feb 24, 2018, 11:39:28 PM, developer.mozilla.org/en-US/Add-ons/WebExtensions/Examples.
[20] “Modify a Web Page.” MDN Web Docs, Oct 25, 2017, 2:20:38 PM, developer.mozilla.org/en-US/Add-ons/WebExtensions/Modify_a_web_page.
[21] “Safely Inserting External Content Into a Page.” MDN Web Docs, Mar 29, 2018, 1:12:21 AM, developer.mozilla.org/en-US/Add-ons/WebExtensions/Safely_inserting_external_content_into_a_page.
[22] “Security Best Practices.” MDN Web Docs, Feb 14, 2018, 2:14:09 PM, developer.mozilla.org/en-US/Add-ons/WebExtensions/Security_best_practices.
[23] “Cross-Site Request Forgery.” Wikipedia, Wikimedia Foundation, 12 May 2018, en.wikipedia.org/wiki/Cross-site_request_forgery.
[24] Gecko. “Webextensions / Firefox Communication Between Website and Extension Background Script.” Romanian Security Team, 3 Nov. 2017, rstforums.com/forum/topic/106920-webextensions-firefox-communication-between-website-and-extension-background-script/.
[25] Krustev, Vencislav. “News Tab Adware – What Is It and How To Remove It from Firefox.” How to, Technology and PC Security Forum | SensorsTechForum.com, 16 Apr. 2018, sensorstechforum.com/news-tab-adware-remove-firefox/.
[26] “Security/Contextual Identity Project/Containers.” MozillaWiki, 13 Dec. 2017, wiki.mozilla.org/Security/Contextual_Identity_Project/Containers.
[27] “TabHiding - Security Concerns.” MozillaWiki, 22 Mar. 2018, wiki.mozilla.org/WebExtensions/TabHiding#Security_concerns.
[28] “Multiple Vulnerabilities in Mozilla Products Could Allow for Arbitrary Code Execution.” Center for Internet Security, 27 Mar. 2018, www.cisecurity.org/advisory/multiple-vulnerabilities-in-mozilla-products-could-allow-for-arbitrary-code-execution_2018-032/.
[29] Vijayan, Jai. “Rogue Chrome, Firefox Extensions Hijack Browsers; Prevent Easy Removal.” Dark Reading, 18 Jan. 2018, www.darkreading.com/attacks-breaches/rogue-chrome-firefox-extensions-hijack-browsers-prevent-easy-removal/d/d-id/1330854?piddl_msgid=330511.
[30] Dimooz. “Firefox Extensions a Hacker Should Have (and Use).” HackThis!!, 12 Nov. 2017, www.hackthis.co.uk/articles/firefox-extensions-a-hacker-should-have-and-use.
[31] “Cross Site Request Forgery (CSRF) Attack.” Incapsula, www.incapsula.com/web-application-security/csrf-cross-site-request-forgery.html.
[32] Kovacs, Eduard. “Mozilla Adding New CSRF Protection to Firefox.” Information Security News, IT Security News and Cybersecurity Insights: SecurityWeek, 27 Apr. 2018, www.securityweek.com/mozilla-adding-new-csrf-protection-firefox.
[33] Arghire, Ionut. “Mozilla Isolates Facebook with New Firefox Extension.” Information Security News, IT Security News and Cybersecurity Insights: SecurityWeek, 27 Mar. 2018, www.securityweek.com/mozilla-isolates-facebook-new-firefox-extension.
[34] Sanchez-Rola, Iskander, et al. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. Usenix, Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies, www.usenix.org/system/files/conference/usenixsecurity17/sec17-sanchez-rola.pdf.

[35] Carlini, Nicholas, et al. An Evaluation of the Google Chrome Extension Security Architecture. Usenix, An Evaluation of the Google Chrome Extension Security Architecture, www.usenix.org/system/files/conference/usenixsecurity12/sec12-final177_0.pdf.

[36] Buyukkayhan, Ahmet, et al. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities, wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/crossfire-analysis-firefox-extension-reuse-vulnerabilities.pdf.

[37] Guha, Arjun, et al. " 2011 IEEE Symposium on Security and Privacy." Verified Security for Browser Extensions, www.ieee-security.org/TC/SP2011/PAPERS/2011/paper008.pdf.

[38] Bandhakavi, Sruthi, et al. VEX: Vetting Browser Extensions For Security Vulnerabilities. VEX: Vetting Browser Extensions For Security Vulnerabilities, madhu.cs.illinois.edu/usenix10.pdf.

[39] "Content-Security-Policy: Misconfigurations and Bypasses." Compass Security Blog - Offensive Defense, blog.compass-security.com/2016/06/content-security-policy-misconfigurations-and-bypasses/.

[40]"Automated Scanning of Firefox Extensions Is Security Theater (And Here's Code to Prove It)." Automated Scanning of Firefox Extensions Is Security Theater (And Here's Code to Prove It) – Dan Stillman, danstillman.com/2015/11/23/firefox-extension-scanning-is-security-theater.

[41]"File and Directory Entries API." MDN Web Docs, developer.mozilla.org/en-US/docs/Web/API/File_and_Directory_Entries_API.

[42]"Content Security Policy." MDN Web Docs, developer.mozilla.org/en-US/Add-ons/WebExtensions/Content_Security_Policy.