Privacy Threat Analysis Of Browser Extensions

Analyse der Privatsphäre von Browser-Erweiterungen
Bachelor-Thesis von Arno Manfred Krause
Tag der Einreichung:

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science
cased

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

Vorgelegte Bachelor-Thesis von Arno Manfred Krause

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

Tag der Einreichung:

Abstract

Browser extensions are widely used today to enhance the way end users interact with the Internet. But, the broad range of functionality that the browser provides to its extensions can be misused to harm the user and violate his privacy. Indeed, many malicious browser extensions are known to security experts and researchers have already developed ways to detect them. In our work, we contribute a in-depth threat analysis of browser extensions and proof its applicability with a system to integrate explicit attacks into existing extensions. Then, we show that we can integrate our malicious implementations into real-world extensions without the need to change the extensions' structure or privileges. Thereby, the user will not notice the added behavior because the extension requests the same privileges than before. Furthermore, our work is applicable to the majority of modern browsers because we focus an extension architecture supported by Chrome, Firefox, Edge, and Opera.

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den August 22, 2016

(Arno Krause)

Contents

1 Introduction

Web browsers are a common tool nowadays to interact with the world wide web. They allow a user to view web content, interact with web applications, and communicate with others around the world. To enhance this interaction, browsers support extensions that can change the look and behavior of web pages and add further functionality to the browser. For that purpose, the browser provides additional interfaces to its extensions that provide access to browser-internal features such as the user's bookmarks, stored cookies, or additional user interfaces.

The wide range of functionality that a browser provides to its extension is unfortunately used by criminals to attack the extension's user. They implement and distribute malicious extensions such as *Febipos* or *Kilim* which target the user's social media accounts and spread spamming content without the user's consent [10, 11]. Both are known to security vendors and modern anti-virus software is configured to detect and remove them.
But not only anonymous attacker misuse browser extensions. The popular extension *Hola* which currently has about 9 million users received negative publicity after its botnet-like behavior was uncovered. The extension's base functionality allows to bypass regional restrictions on web content by routing the user's requests to other users of their network for who the restrictions do not apply. This creates a VPN-like network that masks the origin of the user's requests but with the difference that each defaulting customer acts as an endpoint for the network. Therefore, the user is regularly exposed to the open Internet by traffic from other users that took on his IP address. Furthermore, the company behind the extension did not inform their users about a second service that allows paying customers to route targeted traffic through the network until the operator of a web platform claimed that the network was used to execute DDoS attacks against his servers [1, 20]. This incident triggered several security analysis of the network and the used software which confirmed the botnet-like behavior [31].

With our work, we want to show the threats that arise from browser extensions and that existing extensions already have the privileges to implement malicious behavior. To increase the scope of our work, we focus an extension architecture that is supported by most modern browsers: Google's *Chrome* browser, Mozilla's *Firefox* browser, Microsoft's *Edge* browser, and the *Opera* browser.

Table 1.1 shows browser usage statistics from different companies [2, 9, 3]. These provide frameworks that other websites include and are therefore able to collect user information from many different web pages. A general and exact statistic about browser usage is currently not possible, because the browsers' companies do not publish the numbers of active users. The presented statistics show that our work is currently applicable to extensions from around 70% of browser users.

1.1 Contribution

In our work, we show that a browser extension posse a threat to the user's privacy and can be use to intentionally harm him. For that purpose, we conducted a threat analysis of extensions and their capabilities. As a proof-of-concept, we designed a system to integrate malicious behavior into existing extensions. Our design consists of interchangeable components that we can integrate in another extension if the extension's declared and the component's needed privileges

Browser	w3counter	StatCounter	NetMarketShare
Chrome	59.5%	62.38%	50.95%
Internet Explorer	8.6%	10.73%	29.60%
Firefox	10.1%	15.43%	8.12%
Safari	13.1%	4.59%	4.51%
Opera	2.6%	-	1.40%
Edge	1.6%	3.04%	5.09%

Table 1.1: Different statistics about the global browser share of July 2016 [2, 9, 3]

match each other. This allows us to integrate malicious behavior into a presumably benign extension without the need for a change in the extension's privileges which will be displayed on its installation and update.
Finally, we show that our design is indeed applicable to existing extensions. We fetched several extensions with a high number of users and analyzed their privileges. We discovered that many extensions are very complex and provide a bulk of features which results in many needed permissions and thus facilitates the integration of our components.

1.2 Outline

The remainder of this paper is structured as follows. In Chapter 2 we review related works to our topic which include the extension architecture that we focus in our work (Section 2.1), threat analysis of malicious browser extensions (Section 2.2), and the detection of malicious extensions (Section 2.3).
We provide additional background information in Chapter 3 including probably less known terminology that we use in our paper in Section 3.1 and an overview of the extension architecture that we focus in Section 3.2.
In Chapter 4, we present the results of our theoretical threat analysis. Section 4.1 contains threats that arise from the extension's full access to a displayed web page and Section 4.2 contains threats that arise from the functionality provided by the browser to its extensions. We present our design that acts as a proof-of-concept for our theoretical analysis in Chapter 5. It consists of three steps with interchangeable components. We present common methods to identify the current user of a web page and our implemented components for that task in Section 5.1, our components to transfer collected user information to a remote server and fetch the source code for our attack components in Section 5.2, and finally some attack scenarios using browser extensions and our implemented attack components in Section 5.3.
We tested the applicability of our implementations against real-world extensions and present the results of this analysis in Chapter 6. Then, we present existing and from other researchers proposed countermeasures against malicious extensions in Chapter 7. And finally, we discuss further work in Chapter 8 and give a brief conclusion to our work in Chapter 9.

2 Related Works

In this chapter, we review related works that we discovered through our research and that left their mark on our design and concept. In the first section, we discuss the researches that directly affected the extension architecture that we analyze in this paper namely the work of Barth et al. which lay the foundation for the extension architecture and the work of Carlini et al. in which they evaluated the new architecture and conducted additional security features [16, 21]. In the second section, we discuss the effort of others that already conducted a threat analysis of malicious browser extensions which we also contribute in our work. As third, we present efficient approaches to detect malicious extensions. Although we do not contribute in this section, the presented works provide a deeper understanding of possible malicious behavior.

2.1 Extension Architecture

The architecture of Firefox's Add-ons was the target of multiple scientific analysis which revealed several vulnerabilities such as an attacker being able to compromise the user's machine or to hide an installed, malicious extension completely from the user [13, 43]. The researchers proposed different approaches to improve the security for the user such as to find malicious code pattern [13], monitor the extension at runtime and detect the theft of sensitive information [23, 39, 43], or add an integrity check of an extension's code at runtime [43]. A very successful approach was proposed by Barth et al. who designed a new extension architecture that was later on adapted by multiple browsers [16].

Barth et al. analyzed Firefox's Add-on model and found several vulnerabilities which may be used by an attacker to gain access to the user's computer. In their research, they focused on unintentional exploits in extensions which occur because extension developers 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 get control over the user's machine.

To decrease the attack surface in the case that an extension is compromised, Barth et al. proposed a new, more secured model for extensions. The new model consists of three main principles that should hamper an attacker who has successfully compromised an extension:

- **Privilege Separation** Instead of running with the user's full privileges like Firefox Add-ons, the extension's content is divided into three components with different and unique privileges. Each *content script* is executed in the scope of a single web page and has direct access to the web page's DOM. Therefore, it is exposed to potential attacks from malicious web pages and has no further privileges than to exchange messages with the extension's core. The *core* contains the extension's main logic and for that purpose has access to the browser's provided API. It has neither direct access to web pages, nor to the user's machine. To get access to web content, the core can execute content scripts in the scope of a web page or execute XMLHttpRequests to communicate with a web server. To interact with the host machine, the core can send messages to optionally included *native binaries* which are the only components of an extension that have access to the host machine with the user's full privileges. The researches state that these three layers which only communicate through a message channel, are more difficult for an attacker to break through because he has to find an exploit in each layer to finally get access to the user's system.

- **Least Privilege** The access to the browser's API is restricted. By default an extension has no access to any function provided by the browser. The developer has to explicit declare a set of functions to which his extension has access and the extension is not able to increase this set functions at runtime.

- **Strong Isolation** Each component of an extension runs in its own process on the operating system. This isolates them completely from each other because they do not share a common memory section and are therefore not able to invoke methods or access variable of each other. A second isolation mechanism called *isolated world* separates content scripts and the underlying web page. Again, they run in their own, separated process on the operating

system which effectively isolates them from each other, but the content script still has full access to the web page's DOM. To mitigate potential *cross-origin JavaScript capability leaks* [17], the content scripts and the web page host their own instance of the `document` object which mirrors the natively stored DOM of the web page. Any modification to the DOM is transmitted to all instances, but any other modification of the `document` object is not.

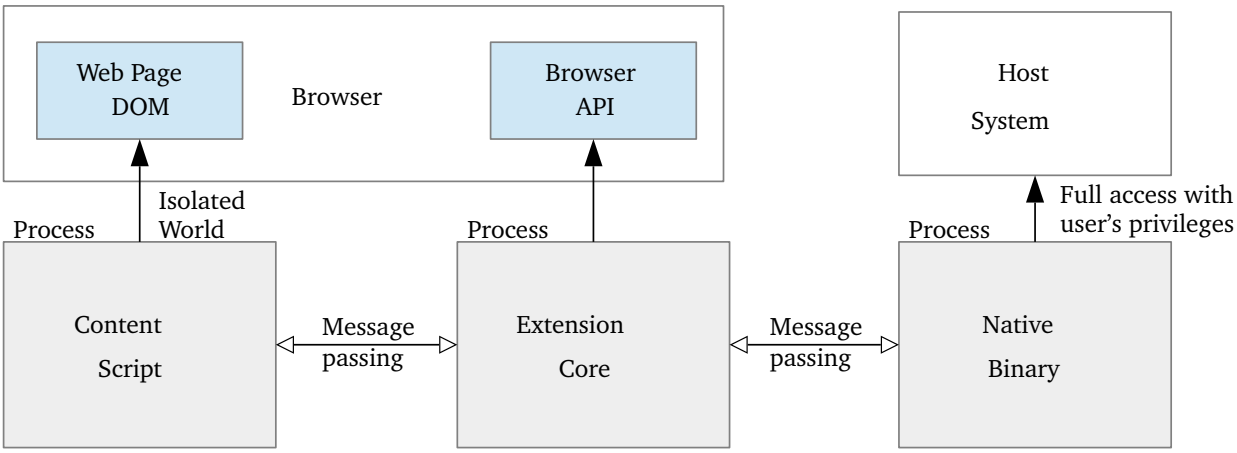

Figure 2.1: Overview of the proposed extension architecture.

The extension architecture developed by Barth et al. was later on adapted by Google's developers for their Chrome browser. Carlini et al. evaluated Google's implementation and focused on the three security principles: *isolated world*, *privilege separation* and *permissions* [21]. For that purpose, 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 property `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. Hence, they stated that 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 Carlini et al. discovered that it is rarely needed because content scripts are already effectively protected by the isolated world mechanism. 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 a 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. Google's developers were responsive to the exposed vulnerabilities and added a default Content Security Policy to the extension's background [15]. Furthermore, they disabled the fetching of remote scripts with HTTP using script elements inside the extension's background script.

In our work, we contribute an analysis of the extension architecture with the focus on intentionally implemented attacks to harm the extension's current user and violate his privacy. Our research shows that although the extension architecture holds many security features, it does not shield the user from intentionally implemented malicious behavior.

2.2 Threat Analysis of Malicious Extensions

Other researchers have already investigated the threats that arise from a malicious extension. For that purpose, they followed a workflow that we also use in our research. They examined the extension's architecture, pointed explicit threats, and implemented an extension that uses found threats as a proof-of-concept.

Liu et al. evaluated the security of Chrome's extension architecture against intentional malicious extensions [33]. Their implementation of a malicious extension is able to execute password sniffing, email spamming, DDoS, and phishing attacks. The extension needs minimal permissions to execute the attacks such as access to the tab system and access to all web pages with the `http://*/*` and `https://*/*` permissions. 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 threat 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 [18]. Bauer et al. implemented several attacks such as stealing sensitive information, executing forged web request, and tracking the user. All their implemented 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.

Ter Louw et al. evaluated Firefox's Add-on model with the main goal to ensure the integrity of an extension's code [43]. 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 the found exploits to hide their extension completely by removing it from the list of installed extensions 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 when 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.

By contrast with presented researches, we do not limit our self to the extension architecture of a single browser but instead focus on a cross-browser architecture. For that purpose, we ensured that our proof-of-concept implementations work as expected in all applicable browsers. Furthermore, our extension analysis shows that even complex attack scenarios can be executed with existing extensions.

2.3 Detection Of Malicious Extensions

Hulk is an dynamic analysis and classification tool for chrome extensions [30]. 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 as *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. The developers stated that his task can not be accomplished by their analysis tool. Lastly an extension without any trace of suspicious behavior is labeled as *benign*. 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.
Before the actual analysis, Hulk performs static preparations such as to collect URLs that may trigger the extension's behavior. The extension's source code and especially the manifest file with its host permissions and URL pattern for content scripts serve as sources. Additionally, the developer fed Hulk with popular websites such as Facebook, Amazon, 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 actual, dynamic analysis consists of the monitoring and evaluating 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 closing Chrome's extension page as soon as the user opens it and therefore denying the user access. 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*. These consist of overridden DOM query functions that create elements on the fly. If a script queries for a DOM element the honey page creates it and monitors any interaction.

WebEval is an analysis tool to identify malicious Chrome extensions [28]. 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 a machine learning algorithm. Ideally the system would run without human interaction. The research of Jagpal et al. shows that the false positive and false negative rates of their system decreases over time but new threats result in a sharp increase. They arrived at the conclusion that human experts must always be a part of their system. In three years of usage WebEval analyzed 99,818 extensions in total and identified 9,523 (9.4%) malicious extensions. Automatic detection identified 93.3% of malicious extensions which were already known and 73,7% of extensions flagged as malicious were confirmed by human experts.
In addition to their analysis pipeline they stored every revision of an extension that was distributed to the Google Chrome web store in the time of their research. A weakly rescan targets extensions that fetch remote resources that may become malicious. New extensions are compared to stored extensions to identifying near duplicated extensions and known malicious code pattern. WebEval also targets the identification of developer who distribute malicious extensions and fake accounts inside the Google Chrome web store. Therefore reputation scans of the developer, the account's email address and login position are included in the analysis process.
The extension's behavior is dynamically analyzed with generic and manual created behavioral suits. Behavioral suits replay recorded interactions with a web page to trigger the extension's logic. Generic behavioral suits include techniques developed by Kapravelos et al. for Hulk [30] such as *honeypages*. Manual behavioral suits test an extension's logic explicit against known threats 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 threats surface WebEval can be expanded to quickly respond. New behavioral suits and detection rules for the self learning algorithm can target explicit threats.

VEX is a static analysis tool for Firefox Add-ons [13]. Bandhakavi et al. analyzed the work flow of Mozilla's developers who manually analyze new Firefox Add-ons by searching for possible harmful code pattern. They implemented VEX to extend and automatize the developer's search and minimize the amount of false-positive results. VEX statically analyses the flow of information in the source code and creates a graph system that represents all possible information flows. They created pattern for the graph system that detect possible cross-site scripting attacks with *eval* or the DOM function *innerHtml* and Firefox specific attacks that exploit the improper use of *evalInSandbox* or wrapped JavaScript objects. More vulnerabilities can be covered by VEX by adding new flow pattern. VEX targets buggy Add-ons without harmful intent or code obfuscation.

3 Background

In this chapter, we provide background information for our paper to give the reader a better understanding of our work. We explain technical elements in Section 3.1 that are probably not commonly known and give an overview of the extension architecture that we focus in Section 3.2.

3.1 Terminology

Browser

A browser is a desktop application to display web pages. It executes HTTP requests to fetch web pages from remote hosts and allows the user to interact with them. Furthermore, the browser executes JavaScript embedded into a web page which allows a dynamic interaction and changes the web pages look according to Cascading Style Sheets (CSS).

Browser Extension

A browser extension is an additionally piece of software that enhances a browser's functionality or user interface. Unlike browser plugins, it is considered as a part of the browser itself. Therefore, it has access to some features of the browser such as cookies or bookmarks and can modify displayed web pages.

Browser Plugin

A browser plugin is an external application that runs on the hosting system. It commonly provides an API or user interface that is embedded into a web page on demand. We do **not** cover plugins in this paper.

Web Application

A web application is a server-client software application hosted on a remote server and accessible for the user through a web page. This allows a company to provide an application without the need for the user to install it locally.

Document Object Model (DOM)

The *Document Object Model* is a standardized API for representing HTML or XML documents as a tree [26]. This allows scripting languages such as JavaScript to easily manipulate the document's content. It is used by browsers to internally store web pages and can be access by JavaScript through the `document` object.

XMLHttpRequest (XHR)

The XMLHttpRequest API allows JavaScript to programmatically execute HTTP and HTTPS requests. It is often used to load dynamic content or to transfer information to a server.

Same Origin Policy (SOP)

The *Same Origin Policy* is a browser security policy that isolates web pages with different *origins* from each other. The origin of a web page is defined by the scheme, host, and port of its URL [14]. The browser permits scripts to access the

content of another web page only if both have the same origin. Whereas, the origin of a script is defined by the origin of the web page that it is embedded into and not the origin from where the script was fetched. This policy prevents malicious scripts to access sensitive information from another web page and to execute requests to cross-origins.

A simple scenario shows why this policy is an important and efficient tool to secure the user's privacy. If a user authenticates to a web application such as an online banking platform, a cookie that contains a unique identifier is often stored inside the user's browser. The cookie is send along any request to the web application and authenticates the user on each request. Without the SOP, a malicious script within another browser tab could send a valid request to the application because the authentication cookie is send along the request. The script could fetch a secured web page from the application and read out the user's sensitive information or perform a request to execute an action on behalf of the user such as to execute a banking transfer.

Content Security Policy (CSP)

The *Content Security Policy* is another browser security policy that restricts the sources from which the web page is allowed to fetch its resources [46]. Unlike the SOP which restricts all web pages, the CSP has to be declared by the web page's author. It is intended to reduce the attack surface in the case that an attacker has successfully compromised the web page. By explicit declaring origins from which the web page is allows to fetch resources, the attacker is hampered because he is not able fetch additional malicious content from his server. Furthermore, the CSP disables the use of *eval* and related functions such as `setTimeout(string, number)`, `setInterval(string, number)`, and `new Function(string)` and it disables inline JavaScript (`<script>...</script>`) and inline event handler (`<button onclick="...">`).

A CSP contains of several directives and corresponding values delimited by a semicolon. A directive declares the restriction for a particular resource type such as scripts, styles, or images. The corresponding values declare origins from which the web page is allowed to load the resources. They may either be a URL with wildcards to match several origins at once, `'none'`, or `'self'`. The *none* key disables the loading of the resource from any origin and the *self* key restricts it to the web page's origin. To remove the additional restriction on eval, the web page's developer may add the key `'unsafe-eval'` to the script directive and to remove the additional restriction on inline scripts, the developer may add the key `'unsafe-inline'` to generally allow inline scripts. A more fine-grained tuning can be achieved by a nonce or hash. A nonce is randomly generated value that is declared in the script directive and enables any script element that has a *nonce* attribute containing the same random value. Similar, adding the computed hash value of a script to the script directive enables the execution of this script. Both approaches do not work for inline event handler.

The following example shows a valid CSP:

```
default-src 'self'; script-src 'self' https://trusted.server.com/* 'unsafe-eval';
```

First, we set the default directive and therewith restrict all other directives to the origin of the underlying web page. Then, we lighten the restriction to load scripts from a trusted server and allow the use of eval in loaded scripts.

3.2 Extension Architecture

In the past, each browser had its own architecture for extensions. This lead to an additional workload for a developer, if he wanted his extension to be compatible with multiple browsers. He had to implement an extension for each browser although each provides equal functionality. Nowadays, the browsers' developers seem to address this unhandy situation and started to use a cross-browser extension architecture. In our paper, we focus this architecture which is supported by Google's *Chrome* browser, Mozilla's *Firefox* browser, Microsoft's *Edge* browser, and the *Opera* browser. We have analyzed the extension's structure in-depth and provide our results in this section.

The cross-browser extension architecture is based on a research from 2010 [16]. The researchers examined the model of Firefox's Add-ons and revealed many vulnerabilities in connection to Add-ons running with the user's full privileges. This enables an attacker, in the case that he has compromised the Add-on, to access arbitrary files and launch new processes. To counter the found exploits, the researchers proposed a new model that should protect the user from unintentionally implemented vulnerabilities.

The developers of Google's Chrome browser where the first to adapt the proposed model in 2010. Then in 2013, the developers of the Opera browser switched the browser's underlying framework to *Chromium* which is also the framework for Chrome [19]. With this change, they adopted the same extension architecture that Chrome uses. In 2015, the developers of Mozilla's Firefox browser announced that they will support the extension model, too. They published a first version of their implementation in version 42 of Firefox. Currently, their implementation is still in development and therefore not all browser APIs are supported [7]. Similar, Microsoft started in 2016 to implement the same architecture for their new Edge browser and the support for many browser APIs is currently still in development [6].

3.2.1 General Structure

Extensions implemented in the cross-browser architecture use only web technologies such as JavaScript, HTML, and CSS. All included files are declared in a mandatory manifest which also holds the extension's meta information such as its name, version, and author. Figure 3.1 shows an example of a manifest file.

```
1  {
2    "manifest_version": 2,
3    "name": "Extension Name",
4    "version": "1.1.2",
5    "background": {
6      "scripts": [ "background.js" ]
7    },
8    "content_scripts": [ {
9      {   "js": [ "content.js" ],
10       "matches": [ "http://*/*", "https://*/*" ] }
11   ],
12   "permissions": [ "cookies", "tabs", "http://*/*", "https://*/*" ],
13   "content_security_policy":
14     "script-src 'self' https://*.example.com/ 'unsafe-eval'; object-src 'self'"
15 }
```

Figure 3.1: Example `manifest.json` file.

The extension's general structure is divided into a background page that holds the extension's main logic and content scripts that are used to interact with web pages.

Background Page

The *background page* is a for the user invisible HTML document which includes JavaScript code that controls the extension's behavior. The extension that uses the manifest shown in **??** declares a single JavaScript file that is added to its

background page on line 4. From within the background page, the extension has access to additional functionality provided by a browser API such as access to the browser's tab-system, the possibility to observe and intercept web requests, or the access to the user's bookmarks. The full list of provided modules is available at the developer platform[1].

To use some of the browser API modules, the extension has to declare a corresponding permission in its manifest. Similar, to access a remote server using a web request or a tab that contains a web page, the extension has to declare a host permission that matches the target's URL. A host permission is a URL pattern that may contain wildecards for the scheme, domain, or path to match several web pages at once. For example, the URL pattern `http://*.example.com/*` matches `http://api.example.com/` and `http://www.example.com/foo` but not `https://www.example.com/` and `http://www.example.org/`. The example manifest file shown in **??** declares the `cookies` and `tabs` permissions and host permissions for all web pages using the two URL pattern `http://*/*` and `https://*/*`.

Additionally, a developer can declare the access to API modules that are not required for his extension's basic functionality as optional permissions. If the extension requests access to an optional API module at runtime, the browser prompts the user to confirm the request. A confirmed optional permission stays active even after a browser restart until the extension itself removes it. Furthermore, host permission may also be declared as optional which gives the user a fine-grained tuning of websites to which an extension has access to.

An extension may include additional user interface elements such as pop-up or option pages. JavaScript inside a user interface has direct access to the background page which allows it to invoke methods and access variables of it and vice versa.

Content Scripts

A extension can not directly access a web page or its DOM from within its background. For that purpose, it uses content scripts which have full access to the DOM of a single web page in whose context they are active. This allows the extension to extract values from the web page or to modify its content. Content scripts are very limited in their access to the browser's API. They can only use a small subset of modules such as the internationalization or the storage module. Furthermore, the background and a content script can not directly interact with each other. They can only use a JSON-based communication channel to transfer commands or data between each other.

Besides the programmatically injection of a content script into a tab from within the extension's background, the developer can also register the content script in the extension's manifest. Using a URL pattern like for host permissions in combination allows to determine the web pages in that the content script will be active. In contrast to the programmatically injection which needs a valid host permission to access the tab, the statically registered content scripts do not need additional permissions. In the manifest shown in **??**, the extension registers a single JavaScript file on line 7. It will be active in any web page whose URL matches `http://*/*` and `https://*/*`.

A content script underlies almost the same restriction as a web page. The Same Origin Policy that restricts a script to access another origin than its own applies partly to a content script. If the web page contains an iframe element from a cross-origin, the content script that runs in the main web page can not access the iframe's content. The iframe is handled as separate web page which means that another content script may be active in its context. To allow the execution of a content script in iframes, the `all_frames` option has to be declared either in the manifest or for a programmatically injection. The SOP also restricts web requests to cross-origins. But here exists a exception for content scripts because they are allowed to execute programmatically executed web requests using a XMLHttpRequest. The request is only restricted by the extension's host permissions but not by the SOP.

3.2.2 Security Features

The researchers that designed the cross-browser extension architecture focused to improve the security of the extension's user. They developed the architecture under the assumption that many extension developers are merely hobby developers and not security experts and therefore may unintentionally implement vulnerabilities [16]. The extension architecture underlies several security features which we present in this section.

[1] `https://developer.chrome.com/extensions/api_index`

Privilege Separation

While analyzing Firefox Add-ons, the researcher found many exploits that allow an attacker to access the user's system. This threat arises from the fact that Firefox Add-ons run with the user's full privileges which allows them to access arbitrary files and launch new processes. Therefore, the researcher's first step was to divide the extension in components where each has a unique set of privileges. Nowadays, no part of the extension is able to access the user's machine directly. They can only exchange messages with natively running applications.

Component Separation

To hamper an attacker that has compromised one component of an extension, the extension's components are strictly separated from each other. The extension's background and each content script runs in its own process on the host's operating system. This creates an efficient boundary between the components because they do not share a common memory section and are therefore not able to invoke methods or access variables of each other. If an attacker has compromised one component of the extension, he can only access the other through the JSON-based message channel. If the extension's developer did not implement an attack vector at the other side of the communication channel, the attacker is not able to compromise the rest of the extension.

Isolated World

Because content scripts are exposed to potential attacks from malicious web pages, they underly an additional security feature called *Isolated World*. Content scripts and the JavaScript inside the web page run in their own process on the operating system. Consequently, they are not able to access methods or variables of each other. Furthermore, content scripts have their own instance of the `document` object mirroring the web page's DOM that is natively stored inside the browser. If a script modifies the DOM, each instance is updated accordingly. But if a script overrides a DOM method or adds a non-standard property to its `document` object, the changes will not be transfered to the internally stored DOM and consequently not to other instances of the `document` object, too.

This mechanism effectively shields content script from *cross-origin JavaScript capability leak* attacks that try to manipulate the behavior of JavaScript methods used by the content script [21, 17]. Furthermore, if a content script inserts untrusted HTML code into a web page's DOM for instance setting the `innerHTML` property of a DOM element, any code inside the content will be executed inside the web page's separated process instead of the content script's process. Thus, the strict separation prevents potential XSS attacks against the extension that the developer may has added unintentionally.

Permissions

The permission system is not only intended to get a lead of the extension's capabilities and purpose, but also acts as a security feature to reduce the attack surface in the case that an attacker has compromised the extension. It works with the principle of least privileges. An extension has by default access to no API modules and origins. Only if it declares a permission it is granted access to the corresponding privilege. An attacker is also restricted by this constraints. He is not able to access other privileges than the declared ones.

Content Security Policy

To reduce the threat of potential cross-site scripting attacks, the background page underlies a Content Security Policy. The default CSP consists of the directives `script-src 'self'` and `object-src 'self'`. This limits the loading of scripts and other resources to files from within the extension's bundle. Additionally, it disables the use of `eval` and inline scripts.

A developer may relax or tighten the policy for his extension by declaring a custom CSP in the extension's manifest which has to contain at least the *script* and *object* directives. Adding the URL of a remote origin to the CSP allows to fetch resources from the declared host. Again, an URL pattern may be used to match several hosts at once, but wildcards

are only allowed for the subdomain. To ensure that remotely loaded resources have not been replaced or modified by a network attacker, only origins that use a secured connection such as HTTPS are allowed. If the developer wishes to use `eval` or related functions in his extension's background page, he has to add the value `'unsafe-eval'` to the *script* directive. If he wishes to use inline scripts, he has to add the hash value of the script to the script directive. The `'unsafe-inline'` key is not allowed and therefore the execution of inline event handler can not be enabled.

The manifest shown in **??**, declares a custom CSP on line 12. It enables the loading of remote scripts from any URL that matches `https://*.example.com/` and enables the user of `eval` in the background page.

Privacy Preserving Measures

The before described security features aim to protect the user from an attacker who tries to compromise the extension. They do not protect the user from intentionally implemented malicious behavior. But there exists some features that if used correctly preserve the user's privacy. These features reduce the privileges an extension has by default and give the user the possibility to explicit grant the extension further privileges.

The `activeTab` permission gives an extension temporary access to the currently active web page if the user explicitly invokes the extension. This occurs if the user clicks an interface element that belongs to the extension such as a button in the browser's toolbar or an entry in the web page's context menu. The extension will lose the access to the web page as soon as the tab loads another web page or is closed. This feature is intended to reduce the amount of needed permissions for extensions that only interact with the current web page and only on the users demand. Otherwise, those extensions would need full and persistent access to any web page to be able to act if the user invokes them.

Similar to the `activeTab` permission, the extension's developer can declare permissions as optional. This states that the permissions are not necessary for the extension's base functionalities but provide additional features on the user's demand. If the extension requests an optional permission, the browser prompts the user to confirm. Even host permissions may be declared as optional, allowing the user to fine-grained control the extension's access to web pages. However, the extension has to remove the requested permission itself. If not, the permission stays active until the extension is uninstalled. The user can not remove an accepted permission by himself if the extension does not provide a proper interface.

3.2.3 Comparison

In this section, we present the architectures of Firefox Add-ons and Safari extensions. We outline their general structure and in case of Firefox Add-ons present their security mechanisms. Then we compare them with the cross-browser extensions.

Firefox Add-ons

Firefox's support for the multi-browser extension model is currently still in development [7]. Therefore, extensions implemented in the old Add-on model are still in use.

Mozilla distributes Add-ons through their own web store[2] but also allows the installation from other sources. Add-ons published through the web store are the target of a review by Mozilla's security experts [4]. After passing the review, the Add-on is signed by Mozilla what is shown on installation to the user. If an Add-on is published private and was not the target of a successful review, it is labeled as untrusted on installation.

To develop an Add-on, an additional JavaScript framework distributed by Mozilla is necessary. It contains several core modules that encapsulate the functionality that the browser provides to the Add-ons similar to the browser's API of cross-browser extensions. Additionally, the `chrome` module even provides full access to browser-internal functionality including the access to the underlying host system. To use a module, the Add-on has to explicit include it in its source code using the global function `require` that takes the module's name as parameter.

[2] `https://addons.mozilla.org/en-US/firefox/`

Firefox uses a security mechanism that is comparable with the cross-browser extensions' permission system. It serves the same purpose to reduce an attackers scope of action if he managed to compromise the extension. On compiling the Add-on, a scanner retrieves all calls of `require` from the Add-on's source code and lists all requested modules. The runtime loader will actually prevent the loading of modules that are not listed. Furthermore, the list of requested module is primarily used by Mozilla's reviewers to get a hint of the extension's purpose and capabilities. If an Add-on requires for example the `chrome` module that gives access to the browser's internal functionality which includes, among others, full access to the host system, the Add-on is subject to an extra in-depth security review. But in contrast to cross-browser extensions, the user does not get notified if the Add-on requests a particular module.

In their framework, Firefox's developers have adapted the separation between the extension's core and content scripts like the cross-browser extensions use it. The `page-mod` module provides the functionality to execute scripts in the scope of a web page. However, this separation is not complete because the `window/utils` module still provides a function to directly access the web page's `window` object.
Furthermore, the Add-ons components are not as strictly separated as the components of a cross-browser extension. Where each content script and the extension's background run in its own process on the operating system, Firefox uses a programmatically solution to separate the Add-ons content. They wrap untrusted content such as web pages and provide only a filtered view to the Add-on. An attacker is more likely to expose a vulnerability in the wrapping implementation than he is able to bypass the process boundary between the contents of a cross-browser extension because he has no access to the hosting system.

In conclusion, the Add-on architecture has less build in security features and provides more functionality than the cross-browser extension architecture. This increases the danger of compromised Add-ons because an attacker is less restricted in his actions and is able to harm the user more than with a compromised cross-browser extension. The fact that each Add-on published over the web store is targeted by a security review decreases the danger from malicious and vulnerable implemented Add-ons, but user are still able to install not-reviewed Add-ons from other sources.

Safari Extensions

Compared to other extension models, Safari extensions are very limited in their capabilities. The browser provides almost no functionality and therefore extension's are restricted to the interaction with web pages. This increases the user's privacy at least partly because an extension can not access information stored inside the browser such as bookmarks or the browsing history. But still, most sensitive information are stored inside web pages.

To publish an extension, the developer needs a certificate provided by Apple. This ensures that only registered developer are able to publish extensions and hampers the distribution of malicious extensions. If a certificate is invalid Safari will disable the developer's extensions.

The browser facilitates the extension development by providing a build-in user interface. It manages the extension's content such as meta information or source code files. An additional feature allows to define content that will be blocked in web pages. The extension's developer can add content-blocking rules to his extension that are compiled into a byte-code format and processed directly at runtime. This renders the programmatically examination of a web page's content and determination of blocking unnecessary and therefore provides a better performance [8].

In conclusion, Safari extensions compared to the other extensions can do the least harm to the user because of their limited capabilities. Furthermore, the developers of malicious extensions are hampered enormous by the registration.

4 Threat Analysis

An extension can use a wide range of different features to enhance the user's interaction with a web page. We want to show that these features may be used by a developer of a malicious extension to harm the user. For that purpose, we have analyzed the extension's capabilities and found potential threats. We have found several permissions and modules that an attacker may use to harm the user's privacy, use his device to launch attacks against others, or remove privacy preserving measures and therefore support attacks from malicious web pages.

4.1 Content Scripts

A big threat to the user's privacy that an extension posses is its full access to a web page. If the extension uses a content script with a URL pattern that matches any web page, it has access to any user data that the page contains. There exists no further restriction such as additional permissions to access password fields or other container of sensitive data. Furthermore, an extension is able to execute cross-origin requests to any arbitrary web page. For that purpose, it can use the mechanics of a DOM element that fetches a resource from a remote server such as an iframe, image, script, or link. We have listed some potential attack scenarios:

- **Steal User Data From Forms** Any information the user transmits over a form in a web page is accessible for an extension. To steal this information, the extension adds an event listener which is dispatched when the user submits the form. At this point in time, the extension can read out all information that the user has entered in the form. This approach gives the attacker access to the user's personal information such as his address, email, phone number, or credit card number but also to identification data such as social security number, identity number, or credentials. Especially username and password for a website's login are typically transmitted with a form.

- **Steal Displayed User Data** Any information about the user that a web page contains is accessible for an extension. To steal this information, the attacker has to explicit know where it is stored in the web page. This is mostly a trivial task, because most web pages are public and the attacker is therefore able to analyze the targeted web page's structure. With this attack, an attacker is able to obtain a broad range of different information such as the user's financial status from his banking portal, his emails and contacts, his friends and messages from social media, or bought items and shopping preferences.

- **Modify Forms** An extension can add new input elements to a form. This tricks the user into filling out additional information that are not necessary for the website but targeted by the attacker. For that purpose, the extension adds the additional input fields to the form when the web page loads, steals the information when the user submits the form, and removes the additional fields afterwards. The last step is necessary because the web application would return an error the form's structure was modified. This attack will succeed if the user does not know the form's structure beforehand. To decrease the probability that the user knows the form already, the extension can determinate whether or not the user has visited the web page to an earlier date before executing the attack.

- **Modify Links** An extension can modify the URL of a link element to redirect the user to a another web page. This page may be malicious and infect the user's device with malware or it may be a duplicate of the web page to which the link originally led and steal the user's data. But the attacker may also enrich himself through the extension's users. Some companies pay money for every time someone loads a specific web page. The attacker can redirect the user to this web page and gain more profit.

- **Denial Of Service** If the source attribute of a DOM element such as an iframe or image changes, the browser sends a HTTP request to fetch the desired resource from the targeted server. An extension can add an unlimited number of elements to the web page's DOM and thereby flood a targeted server with requests. The attack is even more potent if the malicious extension is installed on many browsers and each executes the attack simultaneously.

4.2 Browser API

In this section, we present found threats that arise from API modules and are unlocked by a declared permission. Most of the permissions give access to an API module whose features may be misused in some form.

A developer may declare permissions as optional if they are not necessary for his extension's basic functionality. At runtime, the user is prompted to grant the extension the access to a requested permission. However, the user can not deny a once granted permission by himself. Therefore, a optional permission is as dangerous as standard permission in terms of potential threats because once it was accepted by the user it stays active and the extension may misuse the provided functionality. Furthermore, empirical studies showed that users tend to ignore security warnings which refers also to security relevant prompts [12, 42].

The following paragraphs show the threats which we found associated with a permission. Each paragraph has the permissions's name as heading which is equal to a probably associated module. Additionally, if the permission results in a warning on the extension's installation, we added it to the paragraph.

activeTab

The active tab permission grants an extension temporary access to the currently active web page if the user invokes the extension.

An extension may execute any malicious behavior that does not target a specific web page without the need to declare a content script or host permissions. This allows to decrease suspicious permissions such as the access to all web pages because the `activeTab` permission does not result in a warning on installation.

background

This permission is not related to an API module. Instead, if one or more extensions with the background permission are installed and active, the browser starts its execution with the user's login into the operating system without being invoked and without opening a visible window. The browser will not terminate when the user closes its last visible window but keeps staying active in the background. This behavior is only implemented in Chrome and can be disabled generally in Chrome's settings.

A malicious extension with this permission can still execute attacks even when no browser window is open.

bookmarks

This module gives access to the browser's bookmark system. The extension can create new bookmarks, edit existing ones, or remove them. It can also search for particular bookmarks based on parts of the bookmark's title, or URL and retrieve the recently added bookmarks.

The user's bookmarks give information about his preferences and used web pages. This may be used to identify the currently active user or to determinate potential web page targets for further attacks.

On installation, an extension with this permission shows the user the following warning:

Read and modify your bookmarks

contentSettings

The browser provides a set of *content settings* that control whether web pages can include and use features such as cookies, JavaScript, or plugins. This module allows an extension to overwrite these settings on a per-site basis instead of globally.

A malicious extension can disable settings which the user has explicitly set. This will probably decrease the user's security while browsing the web and support malicious web pages.

On installation, an extension with this permission shows the user the following warning:

Manipulate settings that specify whether websites can use features such as cookies, JavaScript, plugins, geolocation, microphone, camera etc.

cookies

This module give an extension read and write access to all currently stored cookies, even to *httpOnly* cookies that are normally not accessible by client-side JavaScript.

An attacker may use an extension to steal session and authentication data which are commonly stored in cookies. This allows him to act with the user's privileges on affected websites. Furthermore, an malicious extension may restore deleted tracking cookies and thereby support user tracking attempts from websites.

downloads

This module allows an extension to initiate and monitor downloads. Some of the module's functions are further restricted by additional permissions. To open a downloaded file, the extension needs the `downloads.open` permission and to enabled or disable the browser's download shelf, the extension needs the permission `downloads.shelf`.

With the additional permission `downloads.open`, a malicious extension can download a harmful file and execute it. Another malicious approach is to exchange a benign downloaded file with a harmful one without the user noticing.

geolocation

The HTML5 geolocation API provides information about the user's geographical location to JavaScript. With the default browser settings, the user is prompted to confirm if a web page want's to access his location. If an extension uses the geolocation permission, it can use the API without prompting the user to confirm.

On installation, an extension with this permission shows the user the following warning:

Detect your physical location

management

This module provides information about currently installed extensions. Additionally, it allows to disable and uninstall extensions. To prevent abuse, the user is prompted to confirm if an extension wants to uninstall another extension.

An attacker may use the feature to disable another extension to silently disable security relevant extension such as *Adblock*[1], *Avira Browser Safety*[2], or *Avast Online Security*[3].

On installation, an extension with this permission shows the user the following warning:

Manage your apps, extensions, and themes

[1] `https://chrome.google.com/webstore/detail/gighmmpiobklfepjocnamgkkbiglidom`
[2] `https://chrome.google.com/webstore/detail/flliilndjeohchalpbbcdekjklbdgfkk`
[3] `https://chrome.google.com/webstore/detail/gomekmidlodglbbmalcneegieacbdmki`

proxy

Allows an extension to add and remove proxy server to the browser's settings. If a proxy is set, all requests are transmitted over the proxy server.

This feature may be used by an attacker to send all web requests over a malicious server. For example, a server that logs all requests and therefore steal any use information that is transmitted unsecured.

On installation, a extension with this permission shows the user the following warning:

Read and modify all your data on all websites you visit

system

The `system.cpu`, `system.memory`, and `system.storage` permissions provide technical information about the user's machine.

These information may be used to create a profile of the current user's machine and identify him on later occasions.

tabs

An extension can access the browser's tab system with the tabs module. This enables the extension to create, update, or close tabs. Furthermore, it provides the functionality to programmatically inject content scripts into web pages and to interact with a content script which is active in a particular tab. To inject a content script, the extension needs a proper host permission that matches the tab's current web page. The tabs permission does not restrict the access to the tabs module but only the access to the URL and title of a tab.

A malicious extension may prevent the user from uninstalling it by closing the browser's extensions tab as soon as the user opens it. The programmatically injection takes a content script either as a file in the extension's bundle or as a string of code. Therefore, a malicious extension may inject remotely loaded code into a web page as a content script that executes further attacks.

On installation, an extension with this permission shows the user the following warning:

Access your browsing activity

webRequest

This module enables an extension to modify outgoing web requests and their responses. For that purpose, it provides several events which are fired at different stages of a web request's life cycle. To get access to the web requests, the extension needs the `webRequests` permission and proper host permissions that match the request's URL. Additionally, the `webRequestBlocking` permission is needed in order that the extension can block the web request's processing and manipulate it. Blocking can be used to cancel or redirect the request and to modify the request's and the response's headers.

A malicious extension can use this module to remove security relevant headers such as a CSP , intercept outgoing requests, or redirect requests from benign to malicious web pages.

This permission itself does not result in a warning when an extension that requires it is installed. But, to get access to the data of a web request the extension needs proper host permissions and these result in a warning. The often used host permissions `http://*/*`, `https://*/*`, and `<all_urls>` result in the following warning:

Read and modify your data on all websites you visit

5 Design

We have analyzed potential threats in the browser API and showed our results in the previous chapter. In this chapter, we present our design and implementation to proof that the results of our theoretical analysis are applicable in practical scenarios.

We designed our implementation as a set of components with different functionalities and permissions. This allows us to integrate some of our components into an existing, benign extension if the extension has declared the privileges needed to execute the component. Because we do not change the extension's declared content and permissions, there will be no new warnings if the extension is updated. Especially Chrome disables an extension if the warnings shown on installation change with an update and the user has to explicit accept the changes and subsequently re-enable the extension.

Figure 5.1 shows a graphical overview of our design. The *identification* and *communication components* represent the core of our design and have to be included in the benign extension. Their duty is to collect as many as possible pieces of information about the extension's current user and send them to a remote server. If the server has successfully identified the user, we load the source code of our *attack components* from the remote server and execute them. The implementation of the server and the identification is not in the scope of our work.

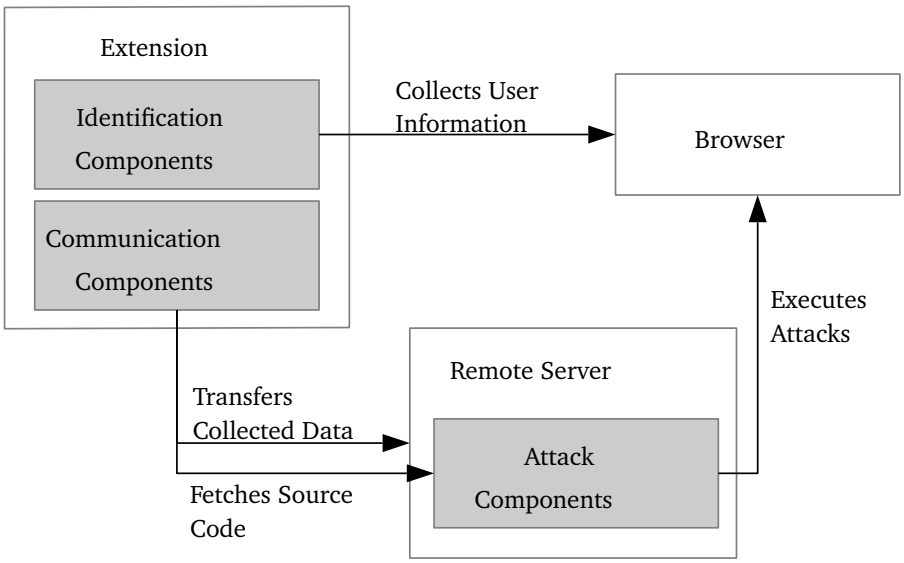

Figure 5.1: Overview of our design.

This design brings several advantages. Because the bulk of our malicious implementations namely the attack components are not present at installation, a static analysis tool which uses content matching to find known malicious code pattern is less likely to detect our malicious intentions. Furthermore, the identification of the current user allows us not only to attack a worthwhile target but also to probably bypass a dynamic analysis. If we are able to detect that our implementation is currently the target of a dynamic analysis, we can fetch a benign script instead of our attack script.

For our implementations, we use the popular web library *jQuery*[1] to simplify the interaction with a web page's DOM.

[1] jQuery Homepage: `http://jquery.com/`

5.1 Identification

Identifying the current user of our extension allows us to target our attack only at specific users. In beforehand, we can evaluate whether or not an attack is worthwhile if we collect as much as possible pieces of information about the user such his financial status or his position in a company we want to target. Furthermore, we are able to detect if our extension is target of an dynamic analysis such as *Hulk* or *WebEval* and evade detection [30, 28].

To find approaches which we can use for our implementation, we first analyzed existing techniques for user tracking and fingerprinting. We present our results and our own implementations to support these techniques in the following two sections. Furthermore, we present selected parts of our implementations to collect the user's personal information in the third section.

We transfer collected information to our remote server which handles the identification. Because the communication between the extension and a remote server is not part of this section, we refer to our implementations that we show in Section 5.2 as *send method*.

5.1.1 User Tracking

User tracking refers to the linking of multiple web pages that were visited by the same user. Applying this technique to web page's that belong to the same domain allows to follow the user's path through the domain's pages and determine his entry and exit points. This is commonly used for web analytics to help the website's author to improve the usability of his layout. User tracking between different domains produces an overview about the user's movement through the Internet. It is often used by advertising companies who extract the user's personal needs and preferences from the websites he visits to provide personalized advertisements.

The general method for user tracking includes a unique identifier which is intentionally stored on the user's machine the first time he visits a tracking web page. If the identifier is retrieved on later occasions, it notifies the tracking party that the same user has accessed another web page. There exists several possibilities to store data inside the browser. We have listed some approach which we found in several research papers:

- **Tracking Cookies** A common used web technology to track users are HTTP cookies. If a user loads a web page any cookie that was stored in beforehand from the same origin is send along the request. This simplifies the tracking, because the browser cares about the storage of the cookie and sends it automatically back to the server.

- **Local Shared Objects** Browser plugins use a technique similar to cookies to synchronize data between different browser sessions. The data is locally stored on the user's system. This allows to track the user behind different browsers. To store and retrieve an identifier inside a browser plugin, additional JavaScript is necessary.

- **Evercookie** This is a JavaScript framework implemented to produce persistent identifiers in a browser that are difficult to remove [29]. 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

If a user loads a web page that includes a resource from a tracking third-party, any cookie that originates from the third-party's domain is send along the request that fetches the resource. This allows the third-party to track the user on every web page that includes their content. These kind of third-party content whose only purpose is user tracking are called `Web Beacons`. A small image, commonly one pixel in size and transparent, is often used for that purpose. Because of its size it requires less traffic and its transparency hides it from the user. It is also used in HTML emails and acts as a read confirmation by notifying the sender that the email's content was loaded. Other nowadays more commonly used web beacons originate from social media such as Facebook's "like" button.

To allow user tracking between different websites, the developers have to explicit include the Web Beacon into their web pages. For our design, we take use of the fact that we are able to access any web the user visits and can add new elements to its DOM. We implemented a component that uses only a content script in every web page. The script embeds an image in the web page which it fetches from our remote server. If our server sends a tracking cookies along the corresponding response, we are able to remotely track the user because our server gets notified every time the user loads a new web page.

Store Identifier In Extension

If we have successfully identified the current user with other techniques, we store an unique identifier inside his instance of the extension. This simplifies his identification next time. An extension has its own local storage and Chrome even provides a cloud based storage which we use to store the identifier. Contrariwise to the web page's storage technologies such as cookies or the HTML5 local storage, the browsers Chrome, Opera, and Firefox do not provide a user interface to clear the extension storage. The user has to manually delete associated files on his hard drive.

5.1.2 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 [24]. Collection and merging these pieces of information creates a fingerprint of the user machine. Creating a second fingerprint at a later point in time and comparing it to stored fingerprints allows to track and identify the user without the need to store an identifier on his computer in beforehand. Because the same kind of information taken from different users will probably equal, it is necessary to collect as much information as possible to create a truly unique fingerprint.

The technique of fingerprinting is nowadays mostly used by advertising companies to get a more complete view of the user and his needs than from simple tracking and by anti-fraud systems that detect if the currently used credentials or device belong to the current user and are not stolen.

There exists numerous scientific papers about fingerprinting from which we present a small subset of techniques with brief descriptions [41, 35, 37, 24, 36, 38].

- **Browser Fingerprinting** The browser provides a variety of technical information to a web page that can be used to generate a fingerprint of the currently used browser and machine. The following list shows examples of these 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 fonts installed on the user's machine 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 (`Font.enumerateFonts(true)`)[27]. If the Flash plugin is not available in a browser, JavaScript can be used to test whether particular fonts are available to the current web page or not. This approach needs a predefined list and may not cover unpopular fonts. It is implemented by writing a string with each font on the web page. If a font is not installed, the browser uses a fall-back font to draw the text. Comparing the width and height of the drawn font to those of the fall-back font gives an evidence whether or not the font is installed.

- **Canvas** Mowery er al. have notices that the same text drawn with canvas results in a different binary representation on different computers and operating systems [36]. They suppose the reasons for these different results are due to differences in graphical processing such as pixel smoothing, or anti-aliasing, differences in system fonts, API implementations or even the physical display. The basic flow of operations consists of drawing as many different letters as possible with the web page's canvas and executing the method `toDataURL` which returns a binary representation of the drawn image.

- **History Sniffing** Reading out the user's web history can not only serve as fingerprinting method but also to simplify user tracking. An outdated but back then common approach to test if a user has visited a particular web page was to use the browser's feature to display links to already visited web pages in a different color. A JavaScript adds a list or predefined URLs to the web page's DOM as link elements and determines the displayed color. Nowadays, link elements that were queried by JavaScript calls behave like unvisited links which prevents this sniffing attack. A current approach detects the redrawing of link elements to determine if the underlying web page was visited before [41]. 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 the information that the web page behind the link was visited before, it redraws the link element. The time it takes to redraw the element can be captured with JavaScript 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. Mowery et al. implemented a set of benchmark test suits to fingerprint different execution speeds [35]. Using these information, they could distinguish between major browser versions, operating systems and micro architectures.

Additional Fingerprinting Data

The before presented techniques for fingerprinting all work from within a web page. Therefore, we can also execute them using a content script in an arbitrary web page. We implemented a component that collects fingerprinting values that the browser provides to a web page. Additionally, we can support this method by collecting technical information that the browser provides to an extension but not to a web page. These pieces of information help us to generate a more accurate fingerprint of the user's browser and system. To access the desired information, we need further permissions. Table 5.1 shows these pieces of information and the permissions needed to access them.

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

Table 5.1: Additional fingerprint information available to an extension.

History Sniffing

The general method of history sniffing explicitly tests if a user has visited a particular web page. This strategy has the disadvantage that not all visited web pages are covered because a predefined list is necessary and often only contains popular web pages. To improve history sniffing, we use an extension to create a more complete list of the web pages that a user has visited and even capture additional information such as the time when and the order in which different web pages were visited. For that purpose, we can either use the *history* module or a content script in every web page. Each approach has its advantages and disadvantages.

With a content script, we can either execute the above described technique of history sniffing which uses a predefined list of web pages to explicit check, or store information such as the URL and the current time every time the content script is injected into a newly loaded web page. In Section 5.1.1, we have already presented a component that exactly executes this task. Our implementation of a Web Beacon notifies us every time the user has opened a new web page by fetching a

resource from our remote server. To get the URL of the visited web page, we can simply transfer it as parameter in the web request that fetches the resource. The disadvantage of using a content script for history sniffing is, that we can not retrieve visited web pages from before the extension's installation or while the extension is disabled. Therefore, it is not an ideal fingerprinting technique because it has to be active for some time to be effective. But, it is a simple alternative if the *history* module is not available because the extension does not have the corresponding permission.

Using the history module allows us to retrieve all visited web pages at once. It provides two for us useful methods `search` and `getVisits`. The first method allows us to retrieve the URLs of all web pages the user has visited and the second one gives us detailed information about every time the user has visited a particular URL such as the concrete time, the referring web page, and how the user has entered the web page. In comparison to using a content script, the history module gives us more pieces of information and executes at once. But the browser's history is vulnerable to deletion or disabling by the user and is disabled if the user uses an incognito window.

We implemented a component that executes a history sniffing attack using the history module. It queries for all visited web pages that the browser has stored and retrieves information about every time the user has visited each. The method that returns all web pages takes as arguments a start and end date that limit the returned list to visited web pages in the given time interval and the maximum number of returned entries. We set the start date to zero because it defaults to 24 hours in the past and the number of returned values to the maximum possible value. This returns us the complete list of stored entries.

Bookmark Sniffing

Similar to history sniffing, we retrieve the user's bookmarks as additional information for a fingerprint but also to get information about his preferred web pages. This allows us to explicit target web pages with our attacks that the user is high likely to visit. Our implemented component needs the *bookmarks* permission to access the corresponding module. The browser stores the bookmarks in a tree structure. Our component uses a recursive function to traverse the tree and extract the bookmarks title and URL. In every recursion step we check if the current node contains a title and an URL. If this is not the case, the current node is not a bookmark but a folder.

5.1.3 Personal User Information

Besides the before described techniques of user tracking and fingerprinting, an extension has more efficient ways to identify a user. The extension has full access to any web page that the user visits and is able to read out any information that is stored inside these pages. This allows us to even identify the person behind the web user by collection personal information such as his full name, address, or phone number.

While developing the components that extract user information we faced several challenges. First at all, an extension whose purpose is to obtain particular data from a web page retrieves the targeted values directly from the web page's DOM and thus heavily depends on the DOM's structure. Due to strong differences between the structures of different web applications, we did not implement a general approach that collects the targeted information from multiple applications.

Furthermore, many modern web pages are of a dynamic nature. They load additional content while the user navigates the page. To be able to collect information that are fetched dynamically, we periodically query the web page and extract desired values. Of course, this results in a flood of information which has to be evaluated by the receiving server. Another challenge that we faced were JavaScript driven text editors in separate iframes. To extract their content, we programmatically injected a small content script in all frames of the current web page. The script evaluates if it is active in the targeted iframe, collects the desired content, and returns it.

We have extracted three categories of web applications as worthwhile targets to collect user data:

- **Social Media** Many people use real names and other personal information for their social media account. If the user visits his account, we can read out his personal data. Furthermore, we can extract information such as the user's social or business environment while the user visits the social web pages of other people he interacts with.

We implemented a component that collects the user's personal information while he navigates through his Facebook account. Our component retrieves the user's name, the URL of his main page, the names of his friends, locations that he has shared with Facebook, and other personal information from his main page such as his date of birth, home town, or salutation.

- **Online Banking** We can identify the current user based on his account numbers if he uses his online banking account. Moreover, we can extract his financial status which gives us information whether or not an attack on his finances is worthwhile.

  We implemented a component that targets a specific online banking platform. It iterates the user's banking accounts and selects the account's name, identification number, and current balance.

- **Email Account** If the user sends and receives emails on his online email account, we can read those and collect probably valuable information that he shares with his acquaintance or other people.

  We implemented two components that target a specific online email client. The first one reads the user's outgoing emails. For that purpose, it registers a listener to the "email send" button and then collects all recipients and the emails body when the user sends the mail. The second component retrieves the emails in the user's inbox. It awaits that the user opens a received email and extracts the sender and body.

5.2 Communication

In the previous section, we have shown our implementations to collect information about the current user to identify him. We need to transfer these data to a remote server which is in charge of the identification. Furthermore, if the identification was successful, we want to attack the user by loading the source code for some malicious behavior from the server. In this section, we present different approaches to transfer information between the extension and a remote server and to fetch the source code for our explicit attacks.

5.2.1 Remote Communication

We often arrive at a point where we have to transfer information from the active extension to our remote server or vice versa. For example, if we want to transfer collected information for identification or stolen personal user data or we want to give the extension the command to execute an attack. We implemented several partly interchangeable components for that purpose and present them in this section.

XMLHttpRequest

Extensions are able to make a web request to a remote server with a XMLHttpRequest. If called from within a web page, the XMLHttpRequest will be blocked by the Same Origin Policy if the target does not match the page's origin. However, the same restriction does not apply to extensions. It is only restricted by the declared host permission in its manifest. For this component, the extension needs a host permission that matches any URL such as `http://*/*`, `https://*/*`, or `<all_urls>`.

Iframe

Another strategy that we use to transfer information to a remote host was described by Liu et al. [33]. They analyzed possible threats in Chrome's extension model by malicious behavior and conducted that an extension can executed HTTP requests to any arbitrary host without cross-site access privileges such as host permissions. For that purpose, they use 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 tries to fetch the web page by sending a request to the defined URL. Adding parameters to the URL allows to send data to the targeted server.

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 origin than the parent web page, the iframe's JavaScript can not access the parent web page and vice versa. The same restriction applies to contents script, too. But an extension can execute another content script in the iframe's web page to access its content. For that purpose, it has to enable the `all_frames` option for a content script either statically in the manifest or for a programmatically injection. We use this fact for our component and use it as a two way communication channel. Executing a second content script inside the iframe, allows us to read information that our server has embedded inside the fetched web page.

Automatic Extension Update

In previous researches, Liu et al. implemented extensions for major browsers that can be remote controlled to execute web based attacks such as *Denial of Service* or spamming. [32, 33]. To control the extensions and send needed information such as the target for a DoS attack or a spamming text, the attacker has to communicate with his extensions. Liu et al. use the automatic update of extensions for that purpose. The browser checks for any extension update on startup and periodically on runtime. The attacker can distribute an attack by pushing a new update and the extension can read commands from a file in it's bundle. This communication channel is on one hand more stealthy than previous approaches because no web request is executed between the extension and the attacker but on the other hand a new extension version is distributed which may be the target of an analysis and it contains the message.

5.2.2 Remote Script Fetching

Because our implementation relies on fetching the source code for a particular attack from our remote server we present our implemented components in this section. Fetching a script remotely from a server is also some kind of communication. Therefore, some components that we present in this section use similar to same approaches like before presented components for communication in Section 5.2.1.

Script Element In Background

HTML pages which are bundled in the extension's installation can include script elements with a source 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.

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

Script Element In Content Script

If we want to execute a remote loaded script only in the scope of a web page, we can take use of the DOM API. It allows us to add a new script element to the current web page. If we set the source attribute of the script element to the URL of our remote server, the browser will fetch and execute the script for us. Our component that executes this strategy does not need additional permissions besides a single content script in the targeted web page.

XMLHttpRequest

In Section 5.2.1 we have shown that an extension can execute a web request to a remote server with the XMLHttpRequest API. To execute the request, our component needs host permissions to any remote server. This approach allows us not only to transfer information to our remote server, but also to receive any data especially the source code for our attack components. We receive the source code in text form and have to forward it to the browser's JavaScript interpreter to execute it.

Before we can execute a remote loaded script, we have to consider what the scripts objectives are. Whether it should act in the extension's background or as a content script. If the first case applies, we can use the JavaScript method `eval` to execute the remote loaded code as a JavaScript application. The use of eval is frowned upon because it is a main source of XSS attacks if not used correctly [5]. On that account, the default Content Security Policy disables the use of eval in the extension's 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. The method `chrome.tabs.executeScript` executes a given string as a content script in a currently open tab. The use of this function is not restricted by a permission. But to access the web page in the tab, the extension needs a proper host permission that matches the web page's URL. Because we have fetched the script with an XHR, we have already declared host permissions that match any URL to execute the request.

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. [18]. The extension which executes the attack does not need the permissions to fetch the malicious script. Another extension can execute this task and then send the remote script to the executing extension. This allows to give both extensions less permissions and thus making them less suspicious especially for

automatic analysis tools. To detect the combined malicious behavior, an analysis tool has to execute both extensions simultaneously. This is a very unconventional approach, because an analysis often targets only a single extension at a time.

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

We implemented two components that work in union and execute the described strategy. The first one dispatches a message on the window object of the current web page and the second registers for the message event and reads the attached message. We admit that these components will have almost no scope of application, because a user needs to have two malicious extension installed for this strategy to work. It servers more as an example to show what strategies are possible.

5.3 Execution

We have already shown how we collect information to identify the current user, transfer it to our remote server which is in charge to evaluate the data, and how we fetch the source code of an attack in the case that the identification was successful. In this section we finally, present our concrete attack implementations. For that purpose, we first present known attack scenarios that make use of an extension's capabilities and then we present our implemented attack components.

5.3.1 Attack Scenarios

In this section, we present two attack scenarios that an extension facilitates. The *Man-in-the-browser* attack manipulates web requests and targets the user's online banking account and a *botnet* of browser extension allows to execute wide scaled web attacks.

Man in the Browser

The Man-in-the-browser (MITB) is a browser based attack related to man-in-the-middle attack (MITM) [22]. The MITM is an attack scenario in computer cryptography against the communication between two parties who directly communicate with each other. The attacker secretly either intercepts and 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 attacker 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 attack often aims to manipulate the traffic between the user and a trusted third party. A common target of an MITB attack is an online banking portal. An attack my look like the following example. The user logs into his account at the targeted banking website. He initiates a transaction and sends the data to the bank's server. The MITB code reads the outgoing request and changes the target account or the transfer amount. The bank's sever is not able to recognize the manipulated request because it receives the information with a valid authentication from the user. It executes the transfer and sends back a receipt. The MITB code reads the response's HTML code and changes the manipulated data back to its original state. To mitigate these attacks, modern banking systems use further security features such as a TAN generator that calculates an additional token from the targeted account's identification number, the transaction amount, and some secrete stored inside the user's banking card. Even if the MITB attack would know the calculation for the token, he has no access to the externally stored secret.

An MITB often comes in the form of a Trojan Horse that infects the browser with its code. An extension facilitates this attack, because it has full access to the displayed web pages and can modify web requests.

Botnet

A botnet is a network consisting of multiple compromised clients that can be controlled to execute large scaled web attacks such as Distributed Denial of Service (DDoS) or spamming attacks. DDoS is a web based attack targeted to make an web service unavailable by overwhelming it with requests. A botnet can be used to call the web service from different sources multiple times per second. The attack either targets the network and flood its bandwidth or the application itself using up the computer's resources. This results in the web service being unavailable for the general public. [32]

Browser extensions in a botnet were previously researched by Liu et al. [32]. As a proof-of-concept, they developed two extension that act as bots, one for Chrome and one for Internet Explorer. Their extension's can execute DDoS attacks, email spamming, and steal the user's credentials. To send an email, the extension uses the user's email account. If the user logs into his account, the extension executes a web request to send an email with the spamming content. The email server will actually execute the request because the user's credentials are send along.

A botnet needs a communication channel that allows the attacker to control the bots. For their extensions, the researchers use the automatic update and modify the content of a configuration file. The extension reads the file and acts accordingly. After the attacker publishes a new update which will start a new attack, the update will be installed as soon as the browser is active. Because the installation of the update is distributed in time, so is the attack which hampers the detection of the botnet.

5.3.2 Implementations

In this section we present our attack components that the extension fetches from a remote server after a successful identification of the current user. Depending on how we have fetched the attack's source code, we need different permissions to execute the attack. In the case that the attack is executed in the scope of a web page, we need either an already active content script that executes the attack or host permissions to inject the attack as independent content script into the web page.

In Section 5.1, we have already shown that we are able to obtain personal user information by extraction them from the web pages that the user visits. Before, we collected these information to generate a fingerprint of the current user and identify him based on this fingerprint. Of course, if we have already obtained sensitive information, we are able to misuse them and harm the user. Therefore, we also use the components presented in Section 5.1 as attack components to steal sensitive user data.

Steal Form Data

The user often transmits sensitive information to a web server using a web form. We can intercept and steal these transmitted data using the simple code snippet `$('form').submit(function() { send($(this).serialize()); })`. The *jQuery* library allows us to simply register an event on each form elements of the current web page that is dispatched if the form is submitted by the user. In this case, we again use jQuery to serialize the form's content and forward it to our send method. This attack is a very general approach that targets any form element in all web pages.

We find a more concrete application for the before described attack on authentication web pages. These usually contain a web form in which the user enters his credentials to authenticate against a web application. If we obtain his credentials, we can impersonate the user on this particular application. Our implementation for this attack is very similar to the before described attack. To explicitly target only authentication forms, we search for an input filed of type password on the page. These are a typical component of a web form in combination with a text field in which the user enters his username. If we have found one, we select the corresponding form element and again attach an event handler that is triggered if the user submits the form. The event handler will then read out the form's values.

The browser provides a storage for entered credentials to the user. If the user has stored his credentials for a particular web page and opens this page on a later occasion, the browser will fill in the stored credentials. We implemented a further component that steals the credentials from an authentication form after the browser has filled in the desired values. Again, we use a similar implementation than the before described attack. We first search for an input element of type password to identify the authentication form. Then, we use the function `setTimeout` and wait for 500 milliseconds to give the password manager the time to fill in the credentials. Finally, we extract the desired values and send them to our remote server.

Manipulate Web Requests

An extension is able to manipulate any outgoing web request. We can redirect a GET request to load a malicious lookalike of the original web page, redirect a POST request to obtain probably sensitive values, or manipulate the values of a request's body to harm the user. Most requests initiates the user from within a web page. Therefore, we use a content script to manipulate URLs and form values inside a particular web page. Additionally, we use the `webRequest` module to intercept and manipulate outgoing web requests.

We implemented a component that manipulates the value of a form field with this code snippet. For this attack, we need explicit knowledge about the targeted web page's structure. We need a CSS selector to find the targeted form and a CSS

selector to find the form's targeted value. This attack may be used to change the transaction amount if the user uses his online banking account for a transaction or change the transaction's target.

We can change the target to which the values of a form are transmitted by manipulating the forms `action` attribute. For that purpose, we use the following code snippet: `$('form').attr('action', TARGET_URL);`. First, we query for a form element and then we set its action attribute to our targeted URL. Similar, we can change the target web page of a link element. For that purpose, we change the value of the link's `href` attribute. Again, we us a small code snippet: `$('a[href~=' + TARGETED_URL + ']').attr('href', TARGET_URL);`. This time, we query for a link element whose current URL belongs to our targeted web page and exchange the URL with the one of our malicious server. We implemented an equal attack which uses the `webRequst` module instead of a content script. This allows us to redirect any web request. To execute this attack, the extension needs the `webRequest` and `webRequestBlocking` permissions and proper host permissions for the targeted web page such as `http://*/*` and `https://*/*`.

Execute Attack In Background

We implemented several components that open predefined web pages to execute particular attacks such as to steal probably stored credentials from the browser's password manager. Different strategies to hide the loading of a new web page were previously discussed by Bauer et al. [18]. We implemented three described approaches:

1. Load the targeted web page in an invisible iframe inside any web page.

2. Load the targeted web page in an inactive tab and switch back to the original web page after the attack has finished.

3. Open a new tab in an inactive browser window and load the targeted web page in this tab. Close the tab after the attack has finished.

The first approach is the least reliable one. There exists several methods to enforce that a web page is not displayed in an iframe. The standardized approach is to use the `X-Frame-Option` HTTP header which is compatible with all current browsers [40, 34]. This transfers the responsibility to enforce that the web page is not loaded into an iframe to the browser. Other approaches use JavaScript to deny the web page's functionality if it is loaded in an iframe or to move the web page's content from the iframe to the main frame.

To open a particular web page in an iframe, we use a content script with the `any_frame` option which enables the execution of the content script in all sub-frames of the current web page such as iframes.Our implementation consists of two steps. First, we check if the content script is currently active in the window's main frame with the conditional statement `window.self === window.top`. In this case, we send a message to the extension's background to retrieve the URL or a targeted web page. This is necessary because the content script itself can not store data - in our case a list of targeted URLs - between different instances of itself. Then, we create a new iframe element, turn it invisible for the user, attach it to the current web page, and load the targeted web page. Because we set the `any_frame` option, the browser will execute our content script in the newly loaded web page. The check if we the script is currently active in the main frame will fail and we can execute the second part of our script which executes another component to attack the loaded web page.

The second and third approach work very similar. Both use the browser's tab system to open a particular web page. Therefore, the components needs the `http://*/*` and `https://*/*` host permissions and for the second approach additionally the `tabs` permission.

To open a particular web page in an inactive tab, we use the browser's tab system provided through the `tabs` module. First, we search for an inactive tab using `chrome.tabs.query({active: false});`. We store the URL of the tab's current web page, to later switch it back. To access the web page's URL, we need the `tabs` permission. Then, we load the targeted web page into the tab and inject a content script that executes a particular attack. The content script will send a message to the extension's background to indicate that it has finished executing the attack. If we receive this message, we will load the tab's original web page into the tab to disguise our attack.

To open a particular web page in a background window, we use a similar implementation to the before described approach to load the web page into an inactive tab. For this approach, we do not search for an inactive tab instead we search for a

tab that is not in the current browser window using `chrome.tabs.query({currentWindow: false});`. We also do not need the `tabs` permission, because we do not access the URL of the tab. Instead, we extract the tab's window id and open a new tab inside the same window using `chrome.tabs.create({url: TARGET_URL, windowId: windowId});`. Simultaneously, we load the targeted web page into the newly created tab. Now we can execute our attack in the tab and close it when the attack has finished.

We tested our implementations in Chrome, Opera, and Firefox with our attack implementation to steal probably stored credentials from the browser's password manager which we have described in Section 5.3.2. To our surprise, the attack was only successful in Firefox. The reason that the attack does not work in Chrome and Opera is that JavaScript has no access to the value of a password input field before any user interaction with the web page occurred. What first seems like a bug is an intended security feature to prevent exactly this kind of attack [44].

Denial Of Service

An extension can execute unrestricted web requests from within a content script using an iframe element. We use this fact to implement a component that executes a Denial of Service attack. Our component uses a single content script that repeatedly creates a new iframe element and loads the targeted web page. To prevent that the browser fetches the web page from its cache, we add a random number as parameter to the target URL. This attack is especially effective if the extension is active in multiple browsers. We can the command to execute the attack to all active extensions, using a communication technique that we have described in Section 5.2.

Download Harmful Files

Using the *download* module, an extension is able to monitor and cancel the user's downloads or to initiate a download itself and even to open a downloaded file. We misuse this feature to get access to the user's machine which the extension itself does not have. For that purpose, we download a harmful file which contains malware or similar malicious content on the user's computer and execute it.

We implemented a component that downloads a file and opens it. For that purpose, the extension needs the `downloads` permission and additionally the `downloads.open` permission to open the downloaded file and the `downloads.shelf` permission to hide the download from the user. The `downloads.shelf` permission enables an extension to show or remove the shelf at the bottom of the browser that shows active downloads.
To open a downloaded file, the user has to interact with the extension and deliver some kind of input such as a mouse click. If no user input is given, the browser blocks the opening of a file. For our implementation, we use a content script that adds an on click event to each element of the current web page. If the user clicks any element, we send a message to our background script which also transfers the desired user input and use this to open the file.
We start our component by disabling the download status bar of the browser so that the user does not see the download. Then, we initiate the download of a harmful file. When the download has finished, we await a message from our content script and use the transmitted mouse click to open the downloaded file. Finally, we delete our file from the browser's list of downloads and re-enable the download status bar to prevent that the user notices the download.

We implemented another component that cancels a download which the user has initiated and then initiate a download ourself of a harmful file. Our goal is that the user does not notices that we cancel his download and initiate a second one and then opens the harmful file. For that purpose, we extract the name and mime-type of the file that the user wants to download and send it along the request to download the harmful file. This allows us to set the mime-type and filename of the harmful file. To further hide our attack, we may deactivate the status bar while we exchange the downloads if the extension has the `downloads.shelf` permission.

Steal Cookies

We implemented two components for our design, that follow different strategies to steal currently stored cookies from the user's browser. Cookies often contain a session id that authenticates the current user to a web application. If we obtain this information, we are able to impersonate the user on this particular web application.

JavaScript in a displayed web page has access to cookies that belong to the same domain as the web page with exception to cookies with the `httpOnly` key set. We use this in our first implementation that uses a single content script in any web page. The cookies are stored inside the DOM and we access them with with `document.cookie`. Obviously, this implementation is restricted to web page's that the user visits while the extension is active and without access to `httpOnly` cookies. Therefore, we implemented the second approach that uses the `cookies` module. Using this module, we have access to all cookies that are currently stored inside the browser without restrictions besides the extension's permissions. To use it, the extension needs the `cookies` permission and host permissions for any web page such as `http://*/*` and `https://*/*`. We use the method `{chrome.cookies.getAll` which accepts an object to filter retrieved cookies based on the cookie's values such as the URL, domain, or its secured state.

Disable Other Extension

In order that other, security relevant extensions do not block our attack implementations, we implemented a component that disables another extension. For that purpose, the extension needs the `management` permission which gives access to the correspondent module. To disable another extension, we need the targeted extension's name or id. The method `chrome.management.getAll` returns a list of currently installed extensions. We iterate the list and compare the name or id of each extension to determine the targeted one. If we find the matching extension, we use the method `chrome.management.setEnabled` to deactivate it.

Remove Security Headers

We implemented a component that removes security relevant response headers. We use this implementation to support our component that opens a particular web page in an iframe which we have shown in Section 5.3.2. This attack is hampered by web pages that use the `X-Frame-Option` header, because the browser disallows displaying the web page inside an iframe. By removing this header from the web request's response, we improve the attacks' success rate. Similar, we can support any content script that executes web requests to fetch content. If the web page uses a Content Security Policy that blocks the loading of scripts or other web pages, we can remove the CSP and allow our content script to execute its task.

Our component needs the `webRequest` permission to access the corresponding module and the `webRequestBlocking` permission to be able to intercept the request's processing until we have removed the targeted response header. Additionally, the component needs host permissions that match the request's target URL.

6 Extension Analysis

We want to show that our implementation is indeed applicable to real world extensions without the need to modify the extension's privileges. Therefore, we evaluated the privileges of popular extensions and analyzed which of our implemented components we can integrate into them. For that purpose, we compared the extension's declared permissions and content with the permissions and content needed for our components.

6.1 Preparations

To facilitate the analysis and in order that we do not have to list all components that we can integrate in the analyzed extension, we categorized our components and divided them into three groups:

A Content script or host permission for all web pages
B Host permission for all web pages
C API permission, combination of permissions, or modified CSP

Each group needs another set of privileges in order to work. Group A contains all components that use only a content script. If the extension hast host permissions to all web pages, we can programmatically inject the content script in any web page. Therefore, we can integrate a component that uses a content script in an extension that declares host permissions for all web pages. Group B contains all components that explicitly need host permissions. These often use an XHR or need access to the current web page but can not execute their behavior as a content script. Finally, group C contains all other components. Because it does not facilitate the analysis if we create a group for each component that uses a unique combination of privileges we decided to list these separately. Table 6.2 shows which component belongs to which group.

We fetched 15 extensions from the Google Chrome Web Store[1] for our analysis. Unfortunately, the web store does not provide the functionality to sort extension based on the amount of current users. Furthermore, the shown number of current users is cut if it is higher than 10,000,000. Therefore, we had to manually search through the web store and select extensions with as many users as possible to evaluate ourself. Table 6.1 shows the extensions that we have fetched and analyzed sorted by the amount of users.

Extension	**Version**	**Amount Users** (as of 2016-08-08)
AdBlock	3.1	10,000,000+
Adblock Plus	1.12.1	10,000,000+
Adobe Acrobat	15.1.0.1	10,000,000+
Avast Online Security	11.1.0.955	10,000,000+
Avira Browser Safety	1.11.0	10,000,000+
Grammarly for Chrome	8.670.558	4,834,105
Unlimited Free VPN - Hola	1.15.403	8,547,308
Google Hangouts	2015.1204.418.1	5,639,654
Google Translate	2.0.6	5,554,424
LastPass: Free Password Manager	4.1.25	4,249,870
Evernote Web Clipper	6.9	4,056,700
Google Mail Checker	4.4.0	3,802,536
Click&Clean	8.9.2	3,249,590
IE Tab	9.8.3.1	3,193,479
Save to Pocket	2.0.35	2,593,149

Table 6.1: Summary of analyzed extension sorted by users.

[1] `https://chrome.google.com/webstore/category/extensions`

Group	Component	Section	Privileges (only for group C)
Identification			
C	Store identifier	Section 5.1.1	storage
A	Web Beacon	Section 5.1.1	
A	History Sniffing	Section 5.1.2	
C	History Sniffing	Section 5.1.2	history
C	Bookmark Sniffing	Section 5.1.2	bookmarks
A	Fingerprinting	Section 5.1.2	
A	Read Personal User Data	Section 5.1.3	
Communication			
B	XHR	Section 5.2.1	
A	IFrame	Section 5.2.1	
C	XHR + eval	Section 5.2.2	host permission + `unsafe_eval`
C	script element in background	Section 5.2.2	remote address in CSP
A	script element in web page	Section 5.2.2	
Execution			
A	steal user data	Section 5.1.3, Section 5.3.2	
A	Manipulate requests	Section 5.3.2	
C	Manipulate requests	Section 5.3.2	host permissions + webRequest + webRequestBlocking
A	Executed concealed attack	Section 5.3.2	
A	DoS	Section 5.3.2	
B	DoS	Section 5.3.2	host permission
C	Download and open file	Section 5.3.2	downloads + downloads.open + downloads.shelf
C	Exchange downloaded File	Section 5.3.2	downloads
A	Steal cookies	Section 5.3.2	
C	Steal cookies	Section 5.3.2	cookies + host permissions
C	disable other extensions	Section 5.3.2	management
C	remove response headers	Section 5.3.2	host permissions + webRequest + webRequestBlocking

Table 6.2: Categorization of our components sorted by section.

6.2 Results

In this section, we present the results of our extension analysis. We extracted needed information for our analysis from the extensions' manifests. Table 6.3 and Table 6.4 show a summary of our analysis.

Our analysis revealed that we can integrate all components from group A into 13 (86%) and all components from group B into 12 (80%) of the 15 analyzed extensions. This allows us to execute the uppermost part of our implementations, especially from the identification section where we presume that the general fingerprinting techniques (Section 5.1.2) and reading out personal user information (Section 5.1.3) are the most efficient ways to identify the current user. Furthermore, we can integrate our component to store an unique identifier (Section 5.1.1) into 5 (33%), our component that executes history sniffing (Section 5.1.2) into 3 (20%), our component that remove security relevant HTTP response headers (Section 5.3.2) into 7 (46%), and our component that manipulates web requests using the `webRequests` module (Section 5.3.2) into 7 (46%) of the analyzed extensions.

Only 3 (20%) analyzed extensions have enabled the use of `eval` and related functions in their background pages. This hampers our attack components that run in the extension's background process because we can not execute the fetched source code using `eval` in most extensions. But, we can use our second component for this task that fetches and executes a remote script using a script element in the extension's background page. To use this component, the extension needs a modified CSP with the targeted server's URL. We found out that 8 of the 15 analyzed extensions declare an additional origin in their CSP to remotely fetch scripts.

We found one extension that declares neither a content script, nor host permissions to all web pages, nor a modified CSP to fetch scripts remotely using script elements. The extension only requests host permissions for `*://*.google.com/`. This restricts our components from group A and B to only access web pages and remote server that match the declared host permission. Furthermore, the extension declares the API permissions `alarms`, `tabs`, and `webNavigation` which none of our components use.

2 of the 15 analyzed extensions declare the `downloads` permission but neither the `downloads.open` nor the `downloads.shelf` permissions. Therefore, we can not integrate our component that downloads and opens an arbitrary file in any of the analyzed extensions (Section 5.3.2). Our other two components that use the *downloads* module to exchange a downloaded file can not hide their actions by disabling the browser bar that shows active downloads without the `downloads.shelf` permission. The user may notices that his genuine download was interrupted respectively removed and the extension has initiated another download itself. This reduces the success rate, but does not make the attack impossible.

Only one of the analyzed extensions has declared the `management` permission. Thereby, we are able to integrate our attack component that disables another extension into it (Section 5.3.2). The extension uses the permission to disable another extension that is known to be malicious. Because it declares the management permission as optional, the user has to explicitly allow the extension to disable other extensions. As soon as the user has granted the permission, our component can also use it to disable other extensions.

Our analysis revealed, that we can integrate our component that collects cookies from the web pages that the user visits in 86% of the analyzed extensions (Section 5.3.2). It uses only a simple content script and is limited to active web pages and cookies without the `httpOnly` flag enabled. To collect all cookies that the browser currently stores, we implemented another component that uses the cookies module. Our analysis revealed that 9 of 15 (60%) of the analyzed extensions request the `cookies` permission and thereby allowing us to integrate our component into them.

Declares modified CSP and `unsafe_eval` enabled?	3/15 (20%)
Declares modified CSP and remotely loaded scripts enabled?	8/15 (53%)
Uses a content script that matches **all** web pages?	11/15 (73%)
Declares host permissions for **all** web pages?	12/15 (80%)
Has host permissions or content script for **all** web pages?	13/15 (86%)

Table 6.3: Summary and comparing of declared content of analyzed extensions.

API Permissions	
tabs	14/15 (93%)
contextMenus	9/15 (60%)
cookies	9/15 (60%)
webNavigation	8/15 (53%)
notifications	7/15 (46%)
webRequest	7/15 (46%)
webRequestBlocking	7/15 (46%)
storage	5/15 (33%)
idle	4/15 (26%)
unlimitedStorage	4/15 (26%)
history	3/15 (20%)
nativeMessaging	3/15 (20%)
activeTab	2/15 (13%)
alarms	2/15 (13%)
downloads	2/15 (13%)
background	1/15 (6%)
bookmarks	1/15 (6%)
browsingData	1/15 (6%)
clipboardWrite	1/15 (6%)
clipboardRead	1/15 (6%)
identity	1/15 (6%)
management	1/15 (6%)
proxy	1/15 (6%)
privacy	1/15 (6%)
system.cpu	1/15 (6%)

Table 6.4: Summary and comparing of declared API permissions of analyzed extensions.

7 Countermeasures

In this chapter, we present two approaches to counter malicious extension. First, we briefly present existing analysis tools to detect malicious code in extension. And then, we describe proposed improvements of the cross-extension's permission system which restricts the extension's privileges by default even more and therefore hampers the integration of our malicious components.

7.1 Detect Malicious Behavior

Mozilla's security experts review Add-ons that they distribute on the official web store to reject malicious implementations. This could also be done for the cross-browser extensions. Indeed, a research team already developed a successful system called *WebEval* to identify malicious extensions. For three yeas, they collected each version of all extensions published to the Chrome Web Store and fed them into their system. They detected about 9,000 malicious extensions which were nearly 10% of all analyzed extensions.

Furthermore, WebEval uses a machine learning algorithm to evaluate each extension and potential malicious findings are verified by human experts. The researchers measured the overall accuracy of their system which increased over time although they experienced drops in the accuracy when new threats emerged.
A disadvantage of WebEval is that it executes the extensions in a sandboxed environment. Therefore, not all malicious interactions with web content can be monitored. For example, the system has no account creation on-the-fly and is therefore not able to analyze the extension's behavior on account-restricted web pages.

Other researchers developed systems to monitor the flow of information in JavaScript-base applications such as extensions [23, 25, 39, 45]. They blocked the flow if sensitive information are transfered to untrusted targets such as remote servers. However, all approaches modify the browser's JavaScript interpreter to enable the tracking and produce a not irrelevant overhead.

We designed our system to be difficult to detect by an analysis. For that purpose, we first identify the current user and only if it was successful we fetch the source code for explicit attacks. This allows us to bypass detection tools that work in sandboxed environments. If the browser itself would analyze extension while the user navigates through the world wide web and detect malicious behavior, we were not able to execute attacks.

7.2 Improved Permissions

Another strategy to increase the privacy of a user was proposed by Liu et al. [33]. They conducted a threat analysis of Chrome extensions and discovered that the extension's unrestricted access to the web page and the possibility to execute unrestricted web requests posse the most harm to the user's privacy. As we showed ourselves, a malicious extension is able to extract and transfer sensitive user information from any web page that the user visits using only a single content script. The researchers stated that the current permission system does not comply to the principle of least privileges and proposed a modified permission system to mitigate the found threats.

The extension's components namely the background and content scripts share the same set of permissions declared in the manifest. This leads to some components having permissions they do not need and therefore not being least-privileged. If the extension requests host permissions for a particular origin, the extension's background is allowed to inject scripts in matching web pages and both - the background and content scripts - are allowed to execute web requests to the origin. Liu et al. proposed separated permission sets for each component and to split host permissions based on particular operations such as to inject a script or execute a web request. However, a content script is still able to execute unrestricted web requests by adding an element to the DOM that fetches a resource from a remote origin. We also use this strategy for our design to execute a request to any arbitrary remote server. To mitigate this threat, the researchers proposed an additional permission that restricts a content script and allows that only listed origins can be added as sources to the DOM.

An extension's content script has unrestricted access to the web page and its content. This allows a malicious extension to extract sensitive information. Liu et al. proposed a categorization of DOM elements to identify container of sensitive information and restricted access to these elements by default. *High level* elements such as input elements of type password or hidden contain inherently sensitive information. *Medium level* elements may contain sensitive information. To identify these elements, the researchers proposed a catalog with regular expressions that match code words such as *username*. Finally, every element else is categorized as *low level*. By default, an extension should have only access to low level elements and the extension may request access to the other levels by a proper permission.

The proposed improvements of the permission systems would indeed hamper our design. If developers request only necessary permissions for their extensions, the amount of components that we can integrate into their implementation would decrease. But still, many extensions need permissions for their legal purpose that also our components need. This is a general problem that most functionality can be used in a benign or malicious way. Therefore, a permission system can not protect a user from a malicious extension because the permissions alone do not provide an accurate statement whether or not an extension is indeed malicious.

8 Further Work

For this paper, we implemented the different components of our designed system and presented a brief description. Furthermore, we showed that real-world extensions exists with the privileges needed for our components. The next step would be to implement a system that takes an extension as input, analyses its privileges, and automatically integrates matching components.

We designed our system in a way that most of the malicious components are not present at the extension's implementation to prevent their detection. But the core components that collect the user information can still be detected by an analysis tool or similar. To mitigate this risk, a strategy should be implemented that handles the order in which the components are used. It should first try to detect analysis tools such as *Hulk*[30] or *WebEval*[28] and then fetch components to identify the current user.

We implemented several attack components for our system as a proof that we can misuse functionality provided by the browser's API modules. There are still modules that pose threats and for which we did not implement components. Therefore, a further step would be to implement additional components with the target to cover all possible threats.

9 Conclusion

We provided an overview of the cross-browser extension architecture, outlined its general structure and the separation between the extension's background and content scripts, and described the security features that should protect the user if an extension is compromised. Then, we presented the results of our threat analysis which revealed that many modules of the browser API contain threats to the user and his privacy and that the biggest threat arises from the extension's full access to displayed web pages.
As a proof-of-concept, we designed a system to integrate malicious behavior into existing extensions. Our design consists of three steps and each step consists of interchangeable components that need different privileges. This structure allows us to integrate our implemented components into a an extension based on the extension's already available privileges. Therefore, we do not change the extension's current privileges and the user will not be notified if the extension is updated. Furthermore, most of our system's code is not present on the extension's installation but instead is loaded remotely if the current user is successfully identified. This reduces the risk of detection because malicious code is only presen
Finally, we showed the applicability of our design to existing extensions. We analyzed the requested privileges of extensions with a high number of users which we fetched from an official extension platform. Our results revealed that many extension indeed possess privileges that match the needed privileges for our implemented attacks.

With our work, we showed that extensions supported by the majority of modern browsers pose a threat to the user and his privacy and that existing, benign extensions currently possess the privileges to execute harmful behavior.
Our implementations proof that many privileges an extension can request may be used to violate the user's privacy or execute web attacks against him. Our design and extension analysis revealed that a benign extension is able to include harmful behavior without the user noticing. Especially remotely fetched scripts in combination with a user identification are difficult to detect, because the malicious code is not present most of the time.
The security features of cross-browser extensions mitigate some threats by default. But, like our extension analysis revealed, many extensions override the default privileges which facilitates the integration of malicious behavior.

List of Tables

List of Figures

Bibliography

[1] Adios, hola! or: Why you should immediately uninstall hola. http://adios-hola.org/index.html#problem_rewriting_history. [accessed 2016-08-16].

[2] Browser & platform market share. https://www.w3counter.com/globalstats.php. [accessed 2016-08-18].

[3] Market share reports. https://www.netmarketshare.com/. [accessed 2016-08-18].

[4] MDN Add-ons - Review Policies. https://developer.mozilla.org/en-US/Add-ons/AMO/Policy/Reviews. [acessed 2016-01-18].

[5] 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].

[6] Microsoft Edge extension API roadmap. https://developer.microsoft.com/en-us/microsoft-edge/platform/documentation/extensions/extension-api-roadmap/. [acessed 2016-07-05].

[7] Mozilla Wiki - WebExtensions. https://wiki.mozilla.org/WebExtensions. [accessed 2015-12-30].

[8] Safari Developer Library - Safari Content-Blocking Rules Reference. https://developer.apple.com/library/safari/documentation/Extensions/Conceptual/ContentBlockingRules/Introduction/Introduction.html#//apple_ref/doc/uid/TP40016265. [accessed 2016-01-07].

[9] Statcounter global stats. http://gs.statcounter.com/. [accessed 2016-08-18].

[10] Trojan: Js/febipos.a. https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Trojan:JS/Febipos.A. [accessed 2016-08-16].

[11] Trojan: Js/kilim.a. https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Trojan%3aJS%2fKilim.A. [accessed 2016-08-16].

[12] D. Akhawe and A. P. Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 257–272, Washington, D.C., 2013. USENIX.

[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. The web origin concept. https://tools.ietf.org/html/rfc6454#section-2. [accessed 2016-07-22].

[15] A. Barth. More secure extensions, by default. http://blog.chromium.org/2012/02/more-secure-extensions-by-default.html, February 2012. [accessed 2016-08-08].

[16] 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.

[17] A. Barth, J. Weinberger, and D. Song. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 187–198, Berkeley, CA, USA, 2009. USENIX Association.

[18] 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.

[19] 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].

[20] F. Brennman. 8chan - hola. `https://8ch.net/hola.html`. [accessed 2016-04-10].

[21] 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.

[22] K. Curran and T. Dougan. Man in the browser attacks. *Int. J. Ambient Comput. Intell.*, 4(1):29–39, Jan. 2012.

[23] 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.

[24] 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.

[25] 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.

[26] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document object model (dom) level 3 core specification. `https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`. [accessed 2016-08-12].

[27] A. S. Incorporated. Actionscript® 3.0 reference for the adobe® flash® platform. `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/text/Font.html#enumerateFonts%28%29`. [accessed 2016-06-03].

[28] 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.

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

[30] 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.

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

[32] 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.

[33] L. Liu, X. Zhang, V. Inc, G. Yan, and S. Chen. Chrome extensions: Threat analysis and countermeasures. In *In 19th Network and Distributed System Security Symposium (NDSS '12*, 2012.

[34] MDN. The x-frame-options response header. `https://developer.mozilla.org/en-US/docs/Web/HTTP/X-Frame-Options`. [accessed 2016-05-24].

[35] 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.

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

[37] 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.

[38] 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.

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

[40] D. Ross, T. Gondrom, and T. Stanley. HTTP Header Field X-Frame-Options. `https://tools.ietf.org/html/rfc7034`. [accessed 2016-05-24].

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

[42] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *USENIX Security Symposium*, pages 399–416, 2009.

[43] 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.

[44] vabr@chromium.org. Chromium blog issue 378419. `https://bugs.chromium.org/p/chromium/issues/detail?id=378419`. [accessed 2016-07-18].

[45] 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.

[46] M. West, A. Barth, D. Veditz, and B. Sterne. Content security policy level 2. `https://www.w3.org/TR/CSP2/`. [accessed 2016-07-22].