Bachelor Thesis

On the Security of the SOP-DOM Using HTML and JavaScript Code

Mario Korth

Date: 30.10.2017
Supervisor: M.Sc. Marcus Niemietz

Ruhr-University Bochum, Germany

Chair for Network and Data Security
Prof. Dr. Jörg Schwenk
Homepage: www.nds.rub.de

Eidesstattliche Erklärung

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

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

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

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other Institution of University.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure, that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

Ort, Datum

Unterschrift

Contents

1. Introduction **1**

2. Related Work **5**

3. Fundamentals **6**
3.1. Same-Origin Policy 6
3.2. Document Object Model 8
3.3. CORS 9
3.4. JavaScript Pseudo Protocol 10
3.5. Self-Registered Protocol Handler 10
3.6. Examined HTML Elements 11

4. Methodology **15**
4.1. Notation 15
4.2. Coverage and Restrictions 15
4.3. Our Approach 16
4.4. Privileges 16
4.5. Selecting Properties for Test Cases 19

5. Implementation **22**
5.1. Testing Framework 22
5.2. Newly Added Test Cases 25

6. Evaluation **31**
6.1. Setup 31
6.2. Results 32

7. Conclusions and Future Work **36**
7.1. Conclusions 36
7.2. Future Work 36

A. Appendix **38**
A.1. Call Stack 38

Bibliography **40**

1. Introduction

The Same-Origin Policy (SOP) has been subject of a lot of security researches [1, 2, 3]. Many bugs have been discovered, reported and fixed [4, 5, 6, 7]. Since its introduction 1996, the SOP is being extended and improved to cover new technologies and larger parts of the JavaScript root object. Two resources that do not share the same origin are isolated from each other by the SOP.

Yet, even though it is constantly improved, according to Schwenk et al., "Today there is [still] no formal definition of the SOP itself. Web Origins as described in RFC 6454 are the basis for the SOP, but they do not formally define the SOP. Documentation provided by standardization bodies [8] or browser vendors [9] is still incomplete. Our evaluation of related work has shown that the SOP does not have a consistent description - both in academic and non-academic world." [1].

Even when merging the documentations from standardization bodies [8] and browser vendors [9, 10, 11] we still would not arrive at a complete documentation of the SOP. Therefore, it is important to have security researchers who constantly analyze and test the behavior of the SOP in different aspects. In Table 1.1 we present an overview of related work that is categorized according to those aspects.

Previous Work. Previously Schwenk et al. [1] focused on the SOP for the Document Object Model (DOM). The key questions they focus on in their work were:

- *How is SOP for DOM access (SOP-DOM) implemented in modern browsers?*
- *Which parts of the HTML markup influences SOP-DOM?*
- *How does the detected behavior match known access control policies?*

They investigated seven Hypertext Markup Language (HTML) elements that have an attribute comparable to the `src` attribute and therefore, can have a different origin. For every element they considered *read*, *write* and *execute* privileges as well as *partial read* and *partial write* privileges. They developed a framework with which they have shown that in the context of the SOP-DOM the access is determined not only by the origin, which consists of `protocol`, `host` and `port`, but also by the kind of Embedding Element (EE) as well as the Cross-Origin Resource Sharing (CORS)

SOP Subset	Description	Related Work
DOM access (this work)	This subset describes if JavaScript code loaded into one "execution context" may access web object in another "execution context". This includes modifications of the standard behavior by changing the Web Origin, for example, using `document.domain`.	[1], [3], [4], [8], [9], [12], [13]
Local storage and session storage	This subset defines which locally stored web object ([name, value] pairs) may be accessed from a JavaScript execution context.	[2], [14], [15], [16]
XMLHttpRequest	This subset imposes restrictions on cross-origin HTTP network access. It contains many ad-hoc rules and its main concepts have been standardized in CORS.	[2], [14], [17], [18]
Pseudo-protocols (this work)	Browsers may use pseudo-protocols like `about:`, `javascript:` and `data:` to denote locally generated content. A complex set of rules applies for the definition of Web Origins here.	[4], [14], [15], [18]
Plugins	Many plugins like Java, Flash, Silverlight, PDF come with their own vaiants of a SOP.	[7], [14], [15], [16]
Window/Tab	Cross-window communication functions and properties: `window.opener`, `open()` and `showModalDialogue()`.	[4], [14], [18]
HTTP Cookies	This subset, with an extension of the Web Origin concept (path), defines to which URLs HTTP cookies will be sent. This defines their accessibility in the DOM for non-httpOnly cookies.	[15], [16], [19], [20], [21]

Table 1.1.: Different subsets of SOP rules from Schwenk et al. [1]. We highlighted the subsets our work focuses on and added more references to related work.

and `sandbox` attributes. Additionally they discovered that in 23.71% of their test cases browsers behaved differently. Using some of those differences they were able to create a login oracle attack for Internet Explorer and Edge. Finally they discussed which known access control model reflects the detected behavior best. They came to the conclusion that enhanced Role-Based Access Control (eRBAC) and Attribute-Based Access Control (ABAC) seem to be the most feasible models to describe the SOP and that further test cases would be needed to decide which of those two models is the better approach.

Need for Testing. Schwenk et al. have shown that the SOP-DOM is influenced by the EE and discovered edge cases in which, contrary to the expectation, access was granted. They were able to leverage one of those edge cases to create a cross-origin login oracle attack. This clearly demonstrates that creating such a testbed is an important task. But while there are more than 15 HTML elements that can have a different origin they only tested seven.

Therefore, it is important to test the remaining elements to ensure that no further, possibly exploitable, edge cases exist. Schwenk et al. already mentioned this as future work. Additionally they stated that, in the context of the SOP-DOM, pseudo protocols and self-registered protocol handler could be researched. This leads to the following research questions:

- *Is the SOP-DOM access for the* `javascript:` *pseudo protocol the same as for native JavaScript code?*
- *How is the SOP(-DOM) implemented for self registered protocol handlers?*

Currently the first question cannot be answered with complete certainty. The HTML5.1 [22] and HTML5.2 [23] standards both specify that the `javascript:` pseudo protocol inherits the origin of the active document. Yet it is unknown how exactly the execution of the `javascript:` pseudo protocol is handled by browsers internally. Therefore, one can only assume that there is no difference from native JavaScript code. We intend to prove that both execution are the same in the context of the SOP-DOM. To prove this we will extend and modify the existing framework such that every test case can be executed with native JavaScript code in the HTML document and within the URL with the `javascript:` pseudo protocol.

Regarding the second question there is even less information available which can be used to properly answer it. Even though the function `registerProtocolHandler` is documented by standardization bodies [22] and browser vendors [24] [25], sometimes implementations do not behave according to those documentations. Therefore, to properly answer the second question we will analyze the possibilities offered by self registered protocol handler using `registerProtocolHandler` and investigate if those protocols are applicable in the context of the SOP-DOM.

Contributions. In this work we contribute with the following aspects:

- We create test cases for currently untested elements (`video`, `audio`, `track`, `picture`) and therefore, increase the test coverage of the framework from Schwenk et al. [1].
- We extend the framework such that one can run all test cases with the `javascript:` pseudo protocol.
- We used the framework to evaluate 645 test cases in eleven different browsers. In 20.93% of the test cases we could observe different behavior across the browsers. Roughly half of those differences are caused by different CORS implementations. Most of the remaining half can be attributed to incompatibilities with the `video`, `audio` and `track` element.

- We discovered that Google Chrome and browsers based on it do not behave according to the CSS Object Model (CSSOM) standard [26]. They allow write access to an cross-origin `link` element if CORS is not used. This issue was acknowledged as a correctness bug in issue 775525 [27].

Structure. In the thesis we will first present related work in Chapter 2. Afterwards we will cover the fundamentals in Chapter 3. In this chapter we will explain core technologies such as the SOP, the DOM and CORS. Additionally we will introduce the `javascript:` pseudo protocol and self registered protocol handler. Finally we will give a comprehensive overview over all HTML elements that can have a different origin and were not tested previously.

After the fundamentals we will thoroughly explain our methodology in Chapter 4. We will introduce the general notation that is used, clarify the test coverage and restrictions, and introduce our approach for test cases. Afterwards there will be an explanation how each privilege of read, partial read, write, partial write and execute should be interpreted and can be generally tested. Finally we will clarify the process used to determine if an element can be tested and how the property to test a specific privilege is selected.

In Chapter 5 with the implementation part of the thesis. This chapter will cover the structure of the framework and will present an explanation on the major changes that we made to the framework. In addition it will clarify how and why we decided to test a specific HTML element. Next to last we will evaluate our results obtained in Chapter 6. While doing so we will explain how we obtained those results. Additionally we will point out the differences that we observed between browsers and analyze how these differences can be leveraged to attack an user. Last but not least we will summarize our work and present an outlook on future work in Chapter 7.

2. Related Work

In the past a lot of security researchers dealt with the SOP. Some of them focused on uncovering new problems and proposing solutions to those problems and other directly proposed approaches to improve the SOP.

SOP Bypasses. Baloch [4] presented multiple bugs that could be used to bypass the SOP. Using the `javascript:` pseudo protocol he was able to execute JavaScript code in the context of an embedded cross-origin `iframe` and `object` element. DNS rebinding is used by Johns et al. [28] to bypass the SOP completely. Karlof et al. [29] describe a new attack called dynamic pharming that leverages DNS rebinding to bypass the SOP and attack an user. Schwenk et al. [1] discovered that the SOP is not enforced correctly for a `link` element that is used to load a Cascading Style Sheets (CSS) file, leading to a cross-origin login oracle attack. Lekies et al. [13] have shown that it is possible to attack an user using dynamically generated JavaScript files.

SOP Replacements. Yang et al. [3] pointed out the shortcomings of the SOP leading to attacks such as Cross-Site Scripting (XSS) and Cross Site Request Forgery (CSRF). They proposed a new approach to isolate content from different origins called information flow control (IFC), and argue that this approach can replace the SOP while being more flexible and secure.

SOP Enhancements. To improve the security of the SOP and prevent dynamic pharming Karlof et al. [29] proposed the locked SOP. In case of the strong locked SOP they propose that the public key of the server is associated with web objects and access from one object to another is granted if and only if the SOP would allow access and the public keys are identical. Another approach to improve the SOP was presented by Johns et al. [28]. They propose the extended SOP which incorporates the `server-origin` into the origin tuple. The `server-origin` is a value transmitted by the server to express a trust boundary. Wang et al. [30] created the browser Gazelle with a multi-principal operating system architecture. They implemented extended access control policies and moved all resource protection into the kernel of the browser. Additionally they showed that it is realistic to modify existing browsers, such that those also implement the concept of multi-principal operating systems.

3. Fundamentals

In this chapter we will explain the core technologies important for the thesis. This covers the SOP, the DOM, CORS, the `javascript:` pseudo protocol and self registered protocol handler. Additionally we will introduce the HTML elements that we considered to test. In those introductions will cover the basic functionality of the element and which of its attributes allow the inclusion of remote resources.

3.1. Same-Origin Policy

Introduced 1996 with the Netscape Navigator 2.0 and JavaScript the general concept behind the SOP was quite simple. Two documents A and B are isolated from each other as long as they do not share the same origin. Yet over time this idea proved to be too simple. The SOP was required to protect more than just the DOM. Continuous improvements resulted in the SOP as we know it today. Today the SOP governs not only read, write and execute privileges for large parts of the JavaScript root object but also sending and receiving network messages using JavaScript. As long as two entities A and B have the same origin neither of the former is forbidden.

Since the core idea of the SOP is still to grant access as long as the two entities A and B have the same origin, and otherwise to deny access, it seems quite simple. Yet when considering in which way the SOP is applied to a specific HTML element and what edge cases and exceptions exist, the SOP grows more and more complex.

Considering network messages, sending GET and POST requests is always allowed. But reading the respective responses is only allowed in the same origin case. Yet again sending cross origin PUT and DELETE requests is forbidden [8]. Using custom headers in requests initiated with JavaScript in only allowed if the source document A and the destination document B share the same origin. Considering the DOM, cross origin writes to an embedded document are forbidden. Yet even though it might not be possible to read every property on an embedded image, `partial read` to the image is allowed, granting access to properties like the *width* and *height* of the image. Disregarding `X-Frame-Options` and framebuster for `iFrames`, embedding a document or resource is generally allowed.

To decide whether to deny or grant access, the SOP compares origins. The concept of web origins is defined in RFC 6454 [31]. An origin is the tuple of `scheme`, `host` and `port`. Section 4 of the RFC clearly defines the algorithm to compute the origin of an Uniform Resource Identifier (URI) while section 5 defines the algorithm to compare two origins. But the HTML5.1 [22] and HTML5.2 [23] standards define the calculation of an origin in a slightly different way. According to those two standards an origin is the tuple of the `scheme`, `host`, `port` and `domain`. This is due to the fact that those two standards allow to relax the SOP using CORS and `document.domain`. A document can set its `domain` to any right hand parent domain that is not a Top-Level-Domain (TLD). This allows cross origin communication if A and B set their respective `domain` to the same value. In this case the SOP also grants access even though the two entities A and B are not `same origin`. Instead they are `same origin-domain`, an additional case defined in the two HTML standards. The comparison of two origins leads to the `same origin-domain` case if and only if both `domains` are equal and not null, or if both origins are `same origin` and the `domain` is null. `Same origin` is achieved if both origins have the same `scheme`, `host` and `port`.

Another difference between the RFC and the two standards is that the RFC defines the `port` as the standard port for the respective `scheme` if no port was provided in the URI while the two HTML standards define the `port` as null if no port is provided in the URI. Table 3.1 is an example of the SOP according to the HTML5.1 and HTML5.2 standard. It is noteworthy that even though one can specify the standard port explicitly in an Uniform Resource Locator (URL) current user-agents ignore the port. For example browsing `http://a.com:80` results in the user-agent loading `http://a.com`. Therefore the port would be null according to the two HTML standards and the resulting documents would be `same origin`.

URL A	URL B	Origin A	Origin B	same origin	same origin-domain
http://evil.com	http://evil.com	(¨http¨, ¨evil.com¨, null, null)	(¨http¨, ¨evil.com¨, null, null)	✓	✓
http://evil.com	http://evil.com:81	(¨http¨, ¨evil.com¨, null, null)	(¨http¨, ¨evil.com¨, 81, null)	×	×
https://evil.com	http://evil.com	(¨https¨, ¨evil.com¨, null, null)	(¨http¨, ¨evil.com¨, null, null)	×	×
http://evil.com:421	http://evil.com:321	(¨http¨, ¨evil.com¨, 421, ¨evil.com¨)	(¨http¨, ¨evil.com¨, 321, ¨evil.com¨)	×	✓

Table 3.1.: SOP according to the HTML5.1 and HTML5.2 standards. Access is granted if at least one of the two cases `same origin` or `same origin-domain` is true.

One of the main reasons for the complexity of the SOP of today may be that “There is no single same-origin policy.” [8]. The SOP has many exceptions and edge cases and some of them are exclusive to a single user-agent. The most common example is that Internet Explorer does not include the port number in the calculation of the origin [9]. This leads to Table 3.1 not representing the SOP for Internet Explorer and Edge. In case of those two browsers row two would be `same origin` and `same origin-domain`. Additionally row four would also be `same origin`.

Another reason for the complexity of the SOP may be that even though the calculation of an origin for an URI consisting of `scheme`, `host` and `port` is relatively straightforward, calculating the origin for a pseudo protocol like `javascript:` or `data:` is not as easy. While the `javascript:` pseudo protocol always inherits the origin of the active document or the navigating browsing context, the rules for `data:` URIs are more complex. As long as one implements the HTML5.1 standard a `data:` URI originating from a document or script inherits the origin of its source. When the `data:` URI does not originate from a document or script, i.e. is entered manually in the address bar, an opaque origin is assigned. An example for a `data:` URI which inherits the origin of its source would be a link. Upon clicking the link the user-agent navigates to the `data:` URI. The origin assigned to the new document is the origin of the anchor element which triggered the navigation and therefore, the origin of the anchor elements parent document. The HTML5.2 standard removes this feature of inheriting origins for `data:` URIs and always assigns an opaque origin to the document created by a `data:` URI.

3.2. Document Object Model

The DOM is often described as a tree like structure which represents the HTML markup of the associated HTML document and allows interaction which said document. This description is partially correct as the ¨DOM defines a platform-neutral model for events, aborting activities, and node trees¨ [32]. Therefore the DOM is actually an Application Programmable Interface (API) which can be accessed through JavaScript and the tree like structure is the DOM tree. As the DOM tree represents the HTML document any changes that are made to it through the DOM will modify the associated HTML document. For instance one could use a piece of JavaScript code to append a new image to the DOM tree. This change will modify the HTML document and, neglecting CSS, result in an image visible to the user. Although the DOM tree represents the HTML markup there can be differences in the hierarchy. An example used by Schwenk et al. [1] is that an `iframe` is accessible through `window.frames` even though it is a child of the HTML document. Yet as the `iframe` is still a child of `document` it can also be accessed via the latter (see Figure 3.1). This shows that there are multiple ways to access a given element in the DOM tree and that the DOM tree is not loop-free.

In 1996 the DOM was introduced with the Netscape Navigator 2.0 alongside the SOP and JavaScript. Similar to the SOP the DOM was improved continuously. In 1998 the DOM was standardized in the standard for the DOM Level 1. Some years later the standard for DOM Level 2 was published. This was followed by the standard for DOM Level 3 and ultimately the living standard for the DOM Level 4.

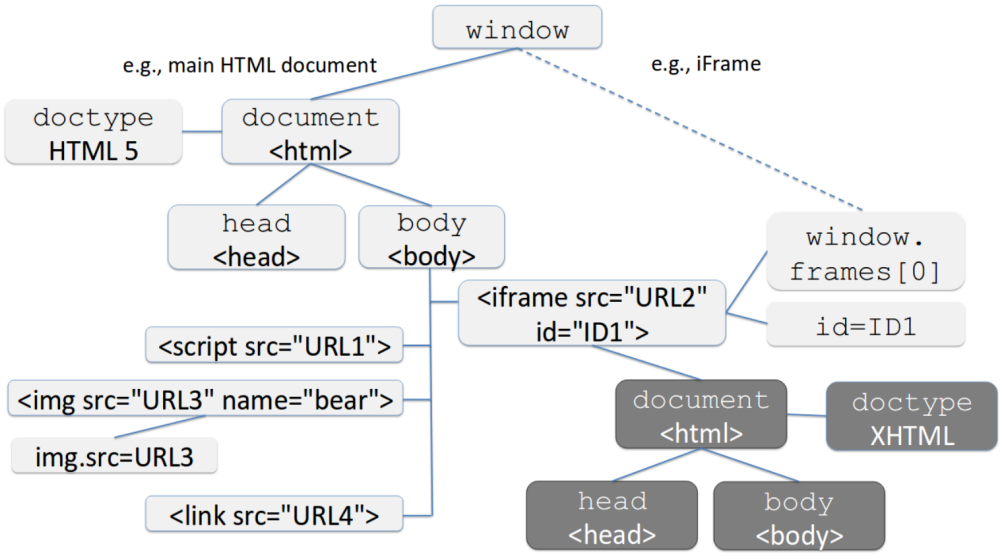

Figure 3.1.: A small extract from the DOM tree displaying multiple ways to access an iframe. Source: Schwenk et al. [1]

3.3. CORS

The HTML5 standards [22, 23] mention different possibilities to relax the SOP. One is to set `document.domain` to a parent domain that is not a TLD. Another option for relaxing the SOP, and therefore enabling cross-origin communication, is CORS [17].

When a request should be performed with CORS the browser adds the `Origin` header to the request and, if necessary, performs a CORS-preflight request using the `OPTIONS` method to determine if CORS is supported by the server.

In the preflight request it uses the two headers `Access-Control-Request-Method` and `Access-Control-Request-Header`. The former is used to tell the server which methods future CORS requests might use for the requested resource. The latter indicates which headers the client might use in future CORS requests for the same resource [33]. In response to this request the server is expected to answer with any CORS headers to indicate that it supports CORS. If the server adds the `Access-Control-Allow-Credentials` header to the response it indicates that future CORS requests can be performed using credentials. In the response to the actual CORS request the header has to be either set to `true`, allowing to expose the resource to the website while using credentials, or not set at all. Additionally a CORS response that wants to allow access to the resource has to contain the `Access-Control-Allow-Origin` header. The value of this header is either the value of the `Origin` header from the request or the asterisk (*) allowing access for all origins. Yet this wildcard value is only allowed for requests without credentials [34].

3.4. JavaScript Pseudo Protocol

The `javascript:` pseudo protocol is barely documented across the Internet. Even though organizations like Microsoft [35] document its existence and usage they do not publish any information on its actual implementation. To the best of out knowledge, the only documentation regarding the implementation of the pseudo protocol can be found in the section *Loading Web pages* of the HTML5 standards[1]. To process a `javascript:` URI one is supposed to first copy everything following the protocol into a new script source object. Afterwards one should remove the URL encoding and decode UTF-8 characters. Thereafter a classic script object is ought to be created with the current browsing contexts settings object and the script source object. Finally the classic script is run and the result should be parsed according to its type. This indicates that JavaScript executed with the `javascript:` pseudo protocol is the same as so called native, more precisely classic, JavaScript code. Therefore, it is expected to find no differences in the execution.

3.5. Self-Registered Protocol Handler

Self-registered protocol handler can be registered using the `registerProtocolHandler` function which is thoroughly documented in the current HTML standards [22, 23]. The function expects three arguments, the protocol name, the target URL and the title. Unless the protocol name is part of a whitelist it has to start with the prefix `web+`, be entirely lowercase and including the prefix has to have a length of at least five characters. For the target URL the `scheme` has to be either HTTP or HTTPS and the `domain` has to be the current `domain` or any parent `domain`. The latter condition does not hold true when the protocol is registered from within a browser extension. Additionally the target URL has to contain the placeholder `%s`. When navigating a link using the self registered protocol handler the placeholder is replaced with the complete URL that was navigated. The argument for the title is not subject to any restrictions as it is only a descriptive title for the user.

An example of registering a protocol handler is given in Listing 3.1. When navigating to a link like `web+exampleprotocol://HelloWorld` the resulting URL which will be loaded is `http://example.com/?q=web+exampleprotocol://HelloWorld`. A common use case for self registered protocol handler is to register a protocol handler for the `mailto:` protocol.

[1] `https://www.w3.org/TR/html51/browsers.html#javascript-urls`, `https://www.w3.org/TR/html52/browsers.html#javascript-urls`

```
navigator.registerProtocolHandler("web+exampleprotocol",
                                  "http://example.com/?q=%s",
                                  "Protocol from example.com");
```

Listing 3.1: JavaScript code to register a protocol handler.

3.6. Examined HTML Elements

In this section we are going to present all HTML elements that we considered for testing. We will describe their general usage and explain which of its attributes can be used to load cross-origin resources. Regarding the SOP-DOM, most of the following HTML elements were not tested previously. Only the `link` element was tested by Schwenk et al. with the `rel` attribute set to `stylesheet`.

3.6.1. Video Element

The `video` element is used to play videos or subtitled audio data. While content may be provided within the element it is not meant to be shown to the user. This content is comparable to an images `alt` attribute. But while in case of the image the content is displayed if the resource cannot be loaded, the `video` elements content is shown if the element itself is not supported by the browser.

In addition to the normal global attributes the `video` element also has the attributes `src`, `crossorigin` and `poster`. The `src` attribute allows to specify the source of the element and the `crossorigin` attribute specifies the CORS settings. The `poster` attribute is comparable to a movie cover since it is used to specify an image which is displayed before the movie is played.

One may think that the `src` attribute is mandatory but actually the attribute can be omitted. If the attribute is omitted the parser expects zero or more `source` elements. Those elements can be used to specify different sources for the parent `HTMLMediaElement`. Typically the user-agent loads the first compatible source. In addition to the `source` elements a `video` element may have zero or more `track` elements as children. A `track` element is used to specify content like captions, separate audio streams or subtitles.

3.6.2. Audio Element

Similar to the `video` element the `audio` element is relatively self explaining. As the name suggests it is used to play sounds or audio streams. In the same way as in the `video` element content can be provided between the elements tags, which will be displayed to the user only if the

browser does not support the element. It also, in addition to the global attributes, supports the `src` and `crossorigin` attributes. While the former allows to specify the source for the audio file the latter is used to define the CORS options.

3.6.3. Source Element

The `source` element is used as a child element in media type elements only. Its purpose is to specify the source for the parent media element which has omitted its `src` attribute. In addition to the global attributes the `source` element has the `src` and `type` attribute. While using the `src` attribute to specify a source the `type` attribute should be used to specify the MIME type of the resource allowing the browser to evaluate if the specified resource is supported before fetching it.

3.6.4. Track Element

The `track` element is allowed as a child of any `HTMLMediaElement`. It is used to specify external text resources like subtitles, captions or metadata. The key attributes of the `track` element are the `src`, `kind` and `srclang`. While the `src` attribute specifies the source of the text resource, the `srclang` attribute specifies the language of the text resource. The `kind` attribute specifies the type of the included text resource, for example, subtitles or captions. If the kind attribute is omitted the content is assumed to be a subtitle.

3.6.5. Input Element

The `input` element is used to interact with the user. Based on its `type` attribute the element has different appearances and attributes. The default value for this attribute is *text* and its corresponding appearance as a simple text field is well known. Additionally when setting the `type` attribute to the value `submit` the `input` element appears as a submit button. Yet the only type that is relevant for this thesis is the `image` type that creates an image button. It uses an image as the visible indicator for the button and is the only type capable of including external resources using the `src` attribute.

3.6.6. Picture Element

The `picture` element itself is only a container for its child `source` and `img` element. It may contain zero of more `source` elements which specify different sources for the contained `img` element. Which source is selected is determined by evaluating the expression in the `media` attribute of the `source` element. For instance an expression could evaluate the screen width and present a

different image according to the result. As a child element of the `picture` element the `source` element does not use the `src` attribute but the `srcset` attribute.

3.6.7. Link Element

The `link` element allows, the same way as the `a` element, to specify a link with the `href` attribute. Based on the `rel` attribute the user-agent decides how to handle the `link` element. The most common value for the `rel` attribute is `stylesheet` which indicates that the specified resource is a stylesheet. The browser then expects a CSS file at the destination of the URL specified in the `href` attribute. Another relatively common usage is to specify an icon that should be displayed for the tab when opening the document. To specify an icon on has to set the `rel` attribute to the value `icon`.

3.6.8. Table Element

The `table` element is used to display data in form of a table and consists of three parts, the table head (`thead`) specifying the column names, the table body (`tbody`) containing all the table data and the table footer (`tfoot`) defining the last row of the table.

Apart from the global attributes the `table` element mainly supports CSS attributes. The only attribute that allows to specify an external resource is the `background` attribute which allows to specify a background image for the table. Since this image can be cross-origin the `table` element is a suitable candidate for testing the SOP-DOM.

3.6.9. Image Element

The `image` element is no longer a standardized HTML element. The only hint of its existence is contained within the parsing instructions of the HTML5.1 [22] and HTML5.2 [23] standards which state that a standard compliant user-agent is expected to generate a parsing error when encountering an `image` element and change it to an `img` element. Afterwards it is supposed to reprocess the element. We tested this in several browsers and discovered that all those browser do change the element to the `img` element. Therefore, as the `image` element is only an alias for the `img` element, it also possesses the same properties as the latter.

3.6.10. Style Element

The `style` element is mainly used to specify CSS for a document and may only appear in `head` of the document. In addition to the global attributes it has the attributes `media`, `nonce`, `type` and `title`. The `media` attribute defines to which types of media the styles apply to. If the attribute is omitted the default value `all` is used which means that the styles are applied to all types of media. The `nonce` attribute is used to store a nonce which is used for Content Security Policy (CSP). The `type` attribute defines the styling language used within the element. If the attribute is not set default value is `text/css`. Last but not least the `title` attribute can be used to define `alternative style sheet sets`. This is comparable to grouping `style` elements. All elements that share the same `title` can be en- or disabled together allowing to switch easily between different style sets.

3.6.11. Canvas Element

The `canvas` element provides a bitmap canvas. This resolution-dependent canvas can be used to draw graphs or images dynamically. The element itself has no attribute to specify a source. Instead for every `canvas` element one needs to create a rendering context and draw the respective image on the context. Possible source elements which can be drawn on a context are another `canvas` element, an `img` element, a `video` element or any other `ImageBitmap`. To determine if a source contains cross-origin data the `canvas` element has an origin-clean flag which is inherited from the source element. An origin-clean flag that is set to `true` indicates that the source data is same-origin and access can be granted accordingly.

4. Methodology

In this chapter we will introduce the general notation used across this work and in the framework. Additionally we will present our test coverage and the restrictions on which we decided beforehand to keep the number of test cases reasonable. After explaining our approach we will we will present comprehensive examples for every privilege, covering read, write and execute privileges as well as `partial` read and write access. For every privilege we will present a general example test case which will be explained thoroughly. Last but not least we will clarify the process that we used to determine which property we can use to test an element for a specific privilege.

4.1. Notation

Since we work with and extend the framework from Schwenk et al. we use the same notation. Therefore, $\overline{\text{HD}}$ denotes that the Host Document (HD) has the origin A. The same applies for the Embedded Document (ED) where $\overline{\text{ED}}$ denotes that the ED has origin A. Likewise $\underline{\text{HD}}$ and $\underline{\text{ED}}$ denote that the respective document has origin B. Following this line of thought, test cases that are "from $\overline{\text{HD}}$ to $\overline{\text{ED}}$" or "from $\overline{\text{ED}}$ to $\overline{\text{HD}}$" test the same-origin case. Likewise do test cases "from $\overline{\text{HD}}$ to $\underline{\text{ED}}$" or "from $\underline{\text{ED}}$ to $\overline{\text{HD}}$" test the cross-origin case.

4.2. Coverage and Restrictions

Considering the sheer amount of test cases resulting from the combinations of the HTML elements in Section 3.6 with all the variations for an origin we need to impose some restrictions, therefore, limiting the amount of test cases to a reasonable number. Schwenk et al. stated that "Web Origins are well understood and have been covered in numerous other publications, we have only covered two different origins with the same `protocol` (HTTP) and two different domains with different `domain` values." [1]. Adapting this argumentation we will focus on the EE and only change the `domain` of the origin. Although we will execute test cases using the `javascript:` pseudo protocol this does not change the `protocol` of the origin. As specified in the two HTML standards the `javascript:` pseudo protocol inherits the origin of the active document.

4.3. Our Approach

With the framework of `your-sop.com` we want to test access rights in two directions, from HD to ED and vice versa. We followed the approach from Schwenk et al. (Figure 4.1) and created test cases which test for (`partial`) read and write access as well as execute privileges. Every test case is implemented with JavaScript code and only covers one privilege for one direction for a single EE with a single set of `sandbox` and CORS settings. To test if the `javascript:` pseudo protocol is the same as native JavaScript code we executed all test cases twice, once with the pseudo protocol and once with native JavaScript code.

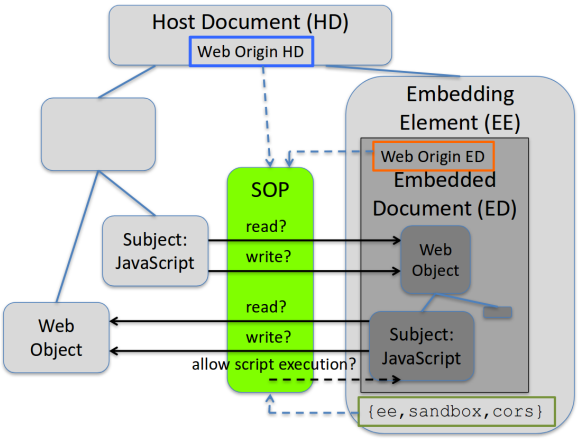

Figure 4.1.: Setup Schwenk et al. used for the test cases in the framework. The EE is part of the HD. As we extended their framework we adapted this setup. Source: Schwenk et al. [1]

4.4. Privileges

All privileges that we consider are from either HD to ED or from ED to HD. At no point do we test the privileges of the EE. Although we later check the properties of the EE, we do so to obtain some information on the ED. As elements differ in the properties they have and the possibilities to access the ED we only specify a general concept for each privilege in this section. The more detailed explanations for each HTML element can be found in Section 5.2.

4.4.1. Full Read Access

Full read access to the target document implies that we can read any information available regarding the ED or HD. An example of full read access from the HD to the ED is a same-origin iFrame without the `sandbox` attribute. But we consider not only the direction from HD to ED but also from ED to HD. Conveniently an example for full read access from the ED to the HD is also a same-origin iFrame without the `sandbox` attribute. The general example code to test for full DOM read access is shown in Listings 4.1 and 4.2.

```
<html>
<head>HD from HD.com</head>
<body>
    <script>
        function test_general_read() {
            ED = document.getElementById("EE").contentDocument;
            HD2ED = ED.getElementById("ID2");
            read_success = (HD2ED.textContent === "Text in ED");
        }
    </script>
    <element id="ID1">Text in HD</element>
    <EE id="EE" src="ED.com/ED.mime" onload="test_general_read()"></EE>
</body>
</html>
```

Listing 4.1: HD testing for full read access to ED. Code is primarily from Schwenk et al. [1]. We moved the JavaScript code in a function that is executed with the onload handler to be more similar to the actual test cases.

```
<html>
<head>ED from ED.org</head>
<body>
    <ED>
        <element id="ID2">Text in ED</element>
    </ED>
    <script>
        ED2HD = parent.getElementById("ID1");
        read_success = (ED2HD.textContent === "Text in HD");
    </script>
</body>
</html>
```

Listing 4.2: ED testing for full read access to HD. Source: Schwenk et al. [1].

At first we define a function in the HD that is supposed to be executed when the EE finished loading. When the onload handler is executed it is fair to assume that the ED is accessible as the handler is executed when the target resource has loaded [36]. There exist exceptions from this behavior but those will be covered separately in the subsections of the respective elements in Section 5.2. At first the function `test_general_read` tries to access the `contentDocument` of the EE, the ED. We then try to receive the HTML element with the id `ID2` from within the ED. In case that we can read the `textContent` prepared in this element we have full read access from the HD to the ED. In addition to the definition of the function the HD also contains an HTML element with a test string and the EE which is used to embed the ED.

Likewise, the ED also contains an HTML element with a test string but it does not contain an EE. Yet it also contains a script which tests for full read access. This script tries to access the HTML element in the HD that has the id `ID1` and tries to read its `textContent`. If the `textContent` equals the test string we have full read access from the ED to the HD.

4.4.2. Full Write Access

Full write access to the can be interpreted as being able to modify all information available regarding the ED or HD. Apart from a few exceptions full read access always implies full write access [1]. Testing full write access can be accomplished by first writing a property or value and afterwards reading the same value, verifying that the write operation was successful. Therefore, the general example code for full write access is very much similar to the one from full read access. In Listing 4.1 we would add `HD2ED.textContent = "Overwritten"` in between the lines eight and nine. Additionally we would modify line nine such that we test for the string ”Overwritten” instead of ”Text in ED”.

Similarly to the HD the ED only needs a few modifications. In Listing 4.2 we have to add `ED2HD.textContent = "Overwritten"` between line nine and ten and modify line ten to test for the string ”Overwritten”. If the HTML element contains our overwritten string we have full write access.

4.4.3. Partial Read Access

Partial read access can be considered as only being able to obtain some information on the ED or HD. Testing for partial read access is not as simple as testing for full read access. In the case of full read access one can select the most convenient property that accesses the target document to test the privilege since all properties should be readable anyway. But in the case of partial read access there might be only one property or sub-property which can be read and allows to obtain

information on the target document.
An example for partial read access is reading the `width` of a cross-origin image. Another example is reading `window.top.location` from within a cross-origin iFrame. To obtain the properties to test specific elements for partial read access we relied on the work from Schwenk et al. [1] as well as the attributes documented in the HTML5.1 and HTML5.2 standards. In addition to those two sources we relied on our method described in Section 4.5 to obtain feasible properties.

4.4.4. Partial Write Access

Partial write access is similar to partial read access. Likewise to the latter we can only modify some information of the target document. Properties that are partially writable are `window.top.location` and the corresponding sub-properties.

4.4.5. The Execute Privilege

Schwenk et al. [1] stated that current sandboxing mechanisms only consider JavaScript execution but not CSS execution. Therefore, they argued that an EE only grants execute privileges to an ED when JavaScript code contained within the ED is executed. To test this privilege for an iFrame with the `sandbox` attribute they sent a message to the HD using the PostMessageAPI. As this line of thought is completely reasonable we adapted their definition and procedure regarding the execute privilege.

4.5. Selecting Properties for Test Cases

Every HTML element that we are testing has various properties. Some of those properties can be used to directly access the target document (ED or HD) and some only present us with some information on the target document, e.g. the `width` of an embedded cross-origin image. Therefore, we need some way to obtain a list of all properties an element has. After obtaining this list we have to remove all entries that cannot be used to test a privilege.

4.5.1. Obtaining a List of Properties

To obtain a list of properties an object has we used the code in Listing 4.1. At first we declare an array called `properties` in line one. In line two we iterate over all properties of an element and append them to the previously declared array. After appending all properties to our array we use the internal `sort` function of JavaScript to sort the array alphabetically.

```
var properties = Array();
for (var i in element) {
    properties.push(i.toString());
}
properties.sort();
for (var i in properties) {
    output.textContent = output.textContent + properties[i] + ", ";
}
```

Listing 4.1: JavaScript snippet to iterate over all properties of an object and print them.

Last but not least we iterate over the sorted array and append every property to the `textContent` of an output element.

4.5.2. Filtering the List of Properties

After we obtained a complete list of properties we need to remove every property that cannot be used to test a privilege. Some properties are useful in one test case while they are not in another. Therefore, we cannot remove those properties in this example even though they might be removed during the elimination process for a specific element. In this example we used a list of properties shared across all elements and will only remove the properties that under no circumstances can be used to access either the target document.

The first entries that we remove from the list are the constants. Following the constants we remove all event handler since they do not allow any access to the target document. Afterwards we remove all entries that relate to namespaces. From the remaining list of elements we remove the browser specific ones like `mozRequestFullScreen`. Thereafter we remove entries regarding CSS like `style`, `className`, `hidden` or `draggable`. We then remove the `webkit` properties as well as the different `Event` entries. After properties relating to attributes are removed we are left with a list of 95 properties that are shared across the elements. From this list we remove the properties that are related to scrolling, e.g. `scrollBy` or `scrollHeight`. Reducing the list further by properties that are used to manipulate the HTML on the same level, e.g. `nextSibling` or `compareDocumentPosition`, we are left with 78 entries. From this list of 78 properties we remove all properties that are related to the HTML markup since all elements that can embed HTML documents or other documents that allow script execution (SVG) already have been tested by Schwenk et al. [1].

Removing properties similar to `accessKey`, `querySelectorAll`, `contextMenu`, `tagName` and `setCapture` we are left with a list of 13 entries:

- `clientHeight`, `clientleft`, `clientTop`, `clientWidth`
- `dataset`
- `getBoundingClientRect`, `getClientRects`
- `lang`
- `offsetHeight`, `offsetLeft`, `offsetParent`, `offsetTop`, `offsetWidth`

Such a list represents the last few candidates that are most likely to allow some sort of access to the ED. In the sections discussing the implementation of the test cases we will discuss such list of properties for every element and hopefully obtain at least one property that allows us to access the ED and therefore, test the element.

5. Implementation

In this chapter we will explain step by step how we decided which HTML elements to examine and how we decided how to test a specific element. Additionally we will describe the structure of the framework and the modifications we made to allow for the test cases to be executed with the `javascript:` pseudo protocol.

5.1. Testing Framework

The framework follows a certain structure and requires a specific layout of the test cases. This section will explain the framework itself and the constrains it requires for test cases. The framework is written in PHP5.6 and HTML with JavaScript.

5.1.1. General Structure

The framework aims to be as modular as possible to prevent redundant code and allow easy modifications. Therefore, our important variables are stored in a global configuration file that is included in any other PHP file. These variables are the protocol used to access the web server as well as the two domain names the framework uses for same- and cross-origin access between the two parties A and B. Last but not least is a variable defining the path which is used to access the web application on the specific hosts as well as some variables consisting of multiple of the former variables, such as `$MAIN_FILE` which ultimately consists of the protocol, the domain name for A, the path and the string `index.php`. Additionally since the config file is included in any other PHP file it is used to start the PHP session.

In the framework the test cases are grouped in PHP files according to the EE and ED. Every of those files defines the test cases for a pair of EE and ED in both directions, HD to ED and vice versa. Additionally each file defines the HTML table which will be used to display the results of the test cases. All these files are included in the main file (`index.php`). Yet even though test cases for the direction ED to HD are defined in above pages, the code finally testing the privilege has to be in the ED. Therefore, the test cases use wrapper pages which according to the arguments provided generate a test case dynamically. The last two important pages are the `cors.php` as well as the

`call_stack.php`. While the `cors.php` file is included in any PHP file that is required to use CORS, the `call_stack.php` is included in almost every file that executes JavaScript. The `cors.php` provides the necessary CORS functionality by setting the CORS header according to the provided get parameters. The `call_stack.php` provides the functionality to execute the test cases with both native JavaScript and the `javascript:` pseudo protocol.

5.1.2. Call Stack

The call stack was developed to solve two major problems. The first is to allow for the test cases to be executed with the `javascript:` pseudo protocol. This in the first place wouldn't require us to develop a central calling interface. We decided to use the concept of a central calling interface because of the second problem.

To execute code with the `javascript:` pseudo protocol in the same origin as the current document we had to overwrite the documents location (`document.location/window.location`) with the pseudo protocol and the corresponding JavaScript code for a test case. This does not cause any problems in an user-agent like Mozilla Firefox but in an user-agent like Google Chrome the preceding JavaScript code is aborted if it had not finished executing the moment the location is overwritten. We concluded that to solve this problem we had to synchronize the execution of all test cases in the HD and ensure that every test case finished executing before starting the execution of the next one. Therefore, we decided on a central calling interface that depletes a documents call stack.

For the call stack we decided on a simple queue implemented as an array (`document.callQueue`). Every entry in the queue is an array which contains the function name in the first position and the arguments array for the function in the second position. To add a new entry to the queue we created the function `call` which takes the function name and the arguments array as arguments. In the beginning the function checks if the call stack is already defined. If the call stack is undefined it creates it before appending a new function to the queue.

To ensure that the previous test case finished executing before calling another we used the concept of a mutex. For our mutex we used the global variable `document.free` and used the value `true` to indicate that currently there is no other test case executing. Therefore, every test case is required to set this variable to `true` when it finished executing. Additionally we would have to set the variable to `false` before we start the execution of a test case.

To deplete the call stack the function `depleteQueue` is used. This function is defined as `async` which allows it to run asynchronously. This might seem to be the opposite of what we want to achieve but it allows the usage of `await`. Await can be used to halt a function until a `Promise` is returned. Using a `promise` we can build a genuine sleep function, something

JavaScript normally lacks [37]. Using our self defined `sleep` function we can truly sleep in JavaScript in an asynchronous function. This is used to sleep a few milliseconds before checking again if the previous function finished its execution reducing the CPU load. In addition to the global variable `document.free` we use another important global variable. To prevent race conditions as much as possible we defined the variable `document.working`. When an instance of the function `depleteQueue` is called it at first checks this variable. If the variable is `true` it aborts its execution with `return 0` to ensure that there is always at most one instance of the function running.

The last task performed by the function is to abort test cases that take too long to execute and therefore, can be considered to be malfunctioning. To achieve this the function saves the start time of an execution in the variable `startTime`. Afterwards in every iteration of the loop that checks if the execution finished it is evaluated if more than ten seconds passed since then. If so then the test case is ”aborted”. This means that the variable `document.free` is set to true which might lead to the abortion of the previous test case.

5.1.3. Central JavaScript Functions

Apart from the functions described in Section 5.1.2 the framework provides some more JavaScript functions which are used directly or indirectly in most test cases. Those JavaScript functions defined in the `index.php` are `set`, `setDirectly`, `getFunctionName`, `postMessageEvt`, `onhashchange` and `generateReport`.

The function `generateReport` is only used to generate a JavaScript Object Notation (JSON) report from the displayed HTML. The `set` function is used to set the value of a cell in the HTML table displaying the results of the test cases. It takes three arguments, the `id` of a test case, the `value` which is the result of the test case and `additionalInfo`. The `id` is used to determine the cell in the HTML table for which the `value` should be set. Additionally the `id` is used to obtain the source code of the respective test case. The source code and the `additionalInfo` are concatenated and written into the `title` attribute of the respective cell. The function `setDirectly` is a lightweight `set` function. The only difference between those two functions is that the former does not add the source code to the `additionalInfo`. The `getFunctionName` function only returns the name of the current function and `postMessageEvt` is the event handler for the `postMessage` event which is used for cross-origin communication. It accepts cross-origin JSON data which contains the results of a specific test case. This data is decoded and inserted into the respective cell using `set` or `setDirectly`. For certain test cases the `onhashchange` function is used instead of `postMessageEvt`.

5.2. Newly Added Test Cases

During the thesis, we had to decide which HTML elements we are going to test. Schwenk et al. said that there are a total of 15 HTML elements which have an attribute that is comparable to the `src` attribute, and therefore can be cross-origin. Since they tested seven of those 15 elements we intended to test the remaining eight. But through thorough analysis of the HTML standards we discovered that there are more than just 15 elements that can have a different origin than the host document. Therefore, we decided to test every element that can have a different origin and was not tested yet.

We took all the elements described in Section 3.6 and examined how we can test if the SOP allows read, partial read, partial write, write or execute access. For every element we first consider which privileges are applicable and then check if the element or resulting object possesses properties that allow us to test the access in the given scenario. To obtain a comprehensive list of possible properties that can be used to access the target document (ED or HD) we used the method described in Section 4.5. Below we will thoroughly explain why or why not we chose to test an element and how we tested an element once we decided on the properties to test for the privileges. Please note that even though some properties might be able to tell us if the ED was loaded we only consider properties that provide us with information on the ED itself, for example, the width of an embedded image or its pixel data.

5.2.1. Video Element

As a member of the `media element` family the `video` element allows to embed media content, namely video data or captioned audio data. As the embedded data is video or audio data it is not necessary to test for the execute privilege since video or audio files do not allow JavaScript execution. Additionally, the same way as with the `img` element, write privileges do not apply. Therefore, we only need to test for read and partial read access. In the context of the `video` element the read privilege can be interpreted as being able to read pixels or frames while partial read might be that one is only able to learn the `width` of the original video file.

At first we followed the process described in Section 4.5.2 to obtain a list of properties that could expose information regarding the ED, and therefore are most likely to grant read or partial read access to the ED. Through this we obtained a list of seven properties which are:

- `clientHeight`, `clientWidth`
- `height`, `width`
- `srcObject`

- `videoHeight`, `videoWidth`

While `clientHeight` and `clientWidth` can contain information related to the ED they actually contain the `height` and `width` of the EE which is only related to the ED if no `height` or `width` was specified for the EE and no CSS was applied. The same applies to `height` and `width`. The `srcObject` can be a `MediaStream` which represents the current media or null. As it was null in all out test cases we chose to use the `videoHeight` and `videoWidth` to determine if we can obtain information on the ED. The `videoHeight` and `videoWidth` always contain the `width` and `height` of the ED regardless of the size the ED is currently displayed by the EE. Using those two properties we can obtain partial read access which we tested with the code in Listing 5.1.

```
1 function [test_case_name . '_onload'](video, id) {
2     set(id, (video.videoWidth == 112) ? 'partial' : 'no');
3     document.free = true;
```

Listing 5.1: JavaScript code to read the width of a video. Code is similar to the test for the img element from Schwenk et al. [1].

But we can actually also obtain full read access. As mentioned in Section 3.6.11 the `canvas` element allows not only an `img` element as its source but also a `video` element. Therefore, we can use the `canvas` element to read pixels from a frame from the video. To test the `video` element using the `canvas` element we use the code in Listing 5.2.

```
1 var c = document.createElement("canvas");
2 c.width = video.videoWidth;
3 c.height = video.videoHeight;
4 var ctx = c.getContext("2d");
5 ctx.drawImage(video, 0, 0);
6 var pixel = ctx.getImageData(2,3,1,1);
7 var data = pixel.data;
8 set(id, (data[0] == 253 || data[0] == 254 || data[0] ==
  ↪ 255)?'yes(pixel)':'no');
```

Listing 5.2: JavaScript code to read a pixel from a video. Large parts originate from Schwenk et al. [1]. We allow more values for the pixel color.

The video variable contains a video element which can be obtained in various ways. In our test cases it is handed to the code as a function parameter. At first we create a new `canvas` element in line one and copy the height and width from the source video. Afterwards in line four we create a

new `CanvasRenderingContext2D` on which we draw the current frame of the video element at the top left corner. We then obtain a single pixel by reading the image data at the position 2, 3 with the size of one times one pixel. Last but not least we obtain the RGBA data from the pixel in line seven. To determine if we have read privileges we check if the R value of the pixel data equals the expected value. Therefore, the overall privileges that we are able to test for the `video` element are partial read and full read access.

5.2.2. Audio Element

Since the `audio` element is also an element of the `media element` family it embeds media data, more precisely ostensible audio data. For audio data the write and execute privileges are not applicable. After applying the process described in Section 4.5.2 we are left with two properties, `currentSrc` and `duration`. The former only provides us with the URL of the source of the ED which does not allow us to access the ED itself. The latter allows us to obtain the duration of the ED. Therefore, we decided to test the `audio` element with the code in Listing 5.3. At first we mute

```
1 audio.muted = true;
2 audio.play();
3 audio.pause();
4 if(audio.duration >= 4) {
5     set(id, "partial");
6 }
```

Listing 5.3: JavaScript code to test for partial read access on an audio file.

the audio file since we do not want to play any sounds to the user of the framework. Afterwards we play and pause the audio file. This increases the reliability of the test case. Before adding this code we often experienced that the duration was not loaded correctly, and realized that most of the time this was fixed by starting the audio file. In line 4 we then check if the duration of the ED is greater than four seconds. Ar first we tried to test for a more specific value, but we discovered that the duration in Google Chrome and Mozilla Firefox differs by more than 20 milliseconds for the same audio file. Therefore, we decided to relax the condition for the duration since we only want to know if we can read the duration and do not necessarily need the exact duration. Last but not least we set the privilege accordingly. Ultimately we can test if we have partial read access or no access at all.

5.2.3. Source Element

We decided to not create test cases for the `source` element. The source element only provides one or more sources for its parent element. While as a child of the `picture` element it also has the `media` attribute, this attribute is not relevant in the context of the SOP as it is only used to determine which source is used according to the expression in the attribute. Since the element itself does not load the specified resource it cannot be tested.

5.2.4. Track Element

Even though the `track` element technically allows to specify a cross-origin source for the WebVTT file in the end it still requires the file to be same-origin. But since this might not be the case in all user-agents we decided to create a few test cases for the `track` element. We use the `track` element to add subtitles to a `video` element. After going through the property list of the `track` element we were left with only the `track` property. Using this property we can access the `TextTrack` provided by the `track` element. We then use the `cues` property of the `track` property to access our test cue. With the code in Listing 5.4 we check if we can read the test string. Modifying (writing) the string works similar, we simply overwrite the first cue with a new string and verify if our modification worked.

```
1 var access;
2 if(track.track.cues[0] && track.track.cues[0] === "Teststring for
  ↪ WebVTT") {
3     access = "yes";
4 } else {
5     access = "no";
6 }
7 set(id, access);
```

Listing 5.4: JavaScript code to test for read access on an track element.

5.2.5. Input Element

As we already stated in Section 3.6.5 we intend to examine the `input` element with the `type` attribute set to `image`. Since the ED is a PNG image we can discard write and execute privileges for the test cases. After going through the properties we are left with only `height` and `width`. Both properties allow us to read the respective information of the image that we embedded resulting in partial read access. But we cannot access the ED any further which prevents us from using the

canvas element to read pixel data. Since we can only test for partial read access we decided against building test cases for the input element since Schwenk et al. already tested partial read access for the img element with the same ED.

5.2.6. Picture Element

For an img element to choose from different URLs requires a special context that is provided by a picture element. Since the picture element provides this context we decided to implement test cases for it even though the element itself does not load any resources. We tested if the contained img element behaves different in the case of partial read access. Therefore we used the code in Listing 5.5 to determine if the img grants partial read access as a child of the picture.

```
1 function [test_case_name . '_onload'](img, id) {
2     set(id, (img.width === 111) ? "partial" : "no");
```

Listing 5.5: JavaScript code to test for partial read access on an img contained in a picture element. Code is equal to the test for the img element from Schwenk et al. [1].

5.2.7. Link Element

Although Schwenk et al. tested the link element using ref=stylesheet one could still test ref=icon. In the same way as with images neither execute nor write privileges apply. After applying the process described in Section 4.5.2 we are left with two groups of properties:

- clientHeight, clientLeft, clientTop, clientWidth
- offsetHeight, offsetLeft, offsetParent, offsetTop, offsetWidth

But since these properties only apply for properties with CSS [38] we cannot use them to obtain information on the ED. Since we did not find a way to access the icon we did not build any test cases for this element.

5.2.8. Table Element

The actual functionality of the table element does not allow to include cross-origin resources. But specifying a background image for the element allows to include an image which can be cross-origin. Since we include an image write and execute privileges do not apply.

As always we obtain a list of properties that are likely to grant read or partial read access.

- `clientHeight`
- `offsetHeight`
- `width`

In the case of the other elements that include images the above three properties often related to the ED. But in the case of the `table` element those properties are completely unrelated to the ED. Since we cannot access the ED we did not create any test cases.

5.2.9. Image Element

In contrast to most of the previous elements to decision to create test cases for this element is not based on the availability of a property that allows to test the access. As explained in Section 3.6.9 the `image` element is only an alias for the `img` element. Since the `img` element was already tested and our own tests have shown that all tested user-agents replace the `image` element with the `img` element we decided against testing the element in the framework.

5.2.10. Style Element

The idea of testing the `style` element was to set the background image of the `table` element and test if the behavior regarding the SOP-DOM is different from when the background image is set using the `background` attribute. But since we cannot access the background image of the table we cannot test if there are differences when using CSS to add the ED to the EE. Therefore, we decided against creating test cases with the `style` element.

5.2.11. Canvas Element

Even though one could test the `canvas` element on its own we decided to use it complementary. This means that we only use the `canvas` element to test other elements for full read access, e.g. the `video` and `img` elements.

6. Evaluation

In this chapter we will explain our test setup and evaluate the results obtained with the former. This will cover the results per HTML element as well as the results for the `javascript:` pseudo protocol. Additionally we will elaborate what results we obtained regarding self registered protocol handlers and why those are not present in the test cases. After we covered those topics we will present the differences that we observed between the different browsers and discuss how one could leverage some of those differences to attack an user.

6.1. Setup

In our setup we evaluated four different HTML elements. Originally we intended to evaluate ten different HTML elements (see Section 3.6) but due to the lack of technical possibilities we were left with the four HTML elements `video`, `audio`, `track` and `picture`. The exact reasons why we decided for or against testing a specific element are explained in Section 5.2.

For each of those element we used either the `src` attribute to specify the source or one or more `source` elements. If an element supported CORS we tested the cross-origin behavior by either not setting the `crossorigin` attribute or by setting it to either `use-credentials` or `anonymous`. For every of those three cases we had the server respond with all the combinations for the `Access-Control-Allow-Origin` and `Access-Control-Allow-Credentials` headers. The former was either not set, set to `your-sop.com`, set to `other-domain.org` or set to the wildcard value, an asterisk (*). For the latter we either did not set it or used the values `true` and `false`. All these combinations lead to an enormous amount of test cases for elements that support CORS. A single value for the `crossorigin` attribute resulted in twelve test cases. Since the `crossorigin` attribute itself allows three values a single test case was expanded to 36 test cases.

To obtain the result we ran all 645 test cases in a browser and exported the result as JSON. We did this for eleven browsers across four different operation systems using native JavaScript code and the `javascript:` pseudo protocol. Afterwards we imported the JSON data into an overview which displays the all results that we obtained. Our results are accessible at `your-sop.com`.

6.2. Results

In this section we will present our results obtained. At first we will cover each EE, neglecting the ones already tested by Schwenk et al. [1]. Afterwards we will describe our results concerning the `javascript:` pseudo protocol. Thereafter we will explain why we did not test self registered protocol handler. Last but not least we will elaborate on the differences that we observed between browsers and conclude the section with an evaluation if some of those differences can be used to attack an user.

6.2.1. Video Element

As long as it is loaded successfully the `video` element allows `partial` read access in the same-origin case as well as the cross-origin case. It behaves similar to an `img` element. Yet one should not consider script execution for the `video` element since the supported video formats do not allow embedded scripts, unlike the image format Scalable Vector Graphics (SVG). But similar to the `img` element we can use the `canvas` element to extract pixel data for various video file formats, e.g. `mp4` or `ogg`. But this only applies if the video file is same-origin or the CORS options allows the access to the video.

6.2.2. Audio Element

Likewise the `video` element the `audio` element only allows `partial` read access. In all cases in which the element was loaded the access was granted and we could read the `duration` of the embedded `mp3` file.

6.2.3. Track Element

The `track` element is somewhat special. It can be used to load different kinds of complementing resources for a `video` or `audio` element. Instead of metadata or caption infos we loaded subtitles for a video. We found that the `track` element can load resources for cross-origin parents but that the resource loaded by the element itself has to be same-origin. In case that we loaded the subtitles from the same-origin we obtained full read and write privileges. The only exception was that we did not obtain write privileges in Edge even though we had read privileges.

6.2.4. Picture Element

Images are commonly included using the `img` element. But when one wants to include different images for different screen sizes one can use the `picture` element which provides a context for an `img` element in which it can choose from multiple sources. We found that when using a `picture` element the behavior the same as when using only an `img` element. We could obtain partial read privileges in both cases, same-origin and cross-origin.

6.2.5. JavaScript Pseudo Protocol

Normally to achieve JavaScript executions developers place JavaScript code in between `script` tags or in event handler. But one can also use the `javascript:` pseudo protocol to execute JavaScript code. It is common knowledge that, as specified in the HTML standard, the `javascript:` pseudo protocol inherits the origin of the active document. But it is unknown if the privileges are the same in both cases, especially regarding the SOP-DOM. We found that the privileges of native JavaScript code and JavaScript code executed with the `javascript:` pseudo protocol are the same. But some test cases regarding sandboxed iFrames still yielded different results in Mozilla Firefox. After performing a few tests we discovered that those test cases yield different results because even though native JavaScript code could be executed inside the iFrames, executing code using the pseudo protocol was blocked by Mozilla Firefox. Since the cause is that Mozilla Firefox is blocking the usage of the `javascript:` pseudo protocol we did not consider this a difference in privileges regarding the SOP-DOM.

6.2.6. Self Registered Protocol Handler

During our work we considered to implement self registered protocol handler into the framework. We tested if self registered protocol handler are applicable in the context of the SOP-DOM. We tried to register protocol handler that point to either a `data:` or `javascript:` URI. Additionally we tried to overwrite existing protocol handler, including `http` and `https`. But since all those tries ended with failure we decided against implementing self registered protocol handler into the framework. But during our testing we discovered that even though self registered protocol handler are subject to strong restrictions some browsers fail to enforce those restrictions properly. Although the documentations state that every non whitelisted protocol name has to start with `web+`, be at least five characters long and only contain lower case ASCII letters, Mozilla Firefox does not enforce those restrictions properly. It allows any non blacklisted protocol name and does not enforce the `web+` prefix. Even an empty string or a protocol name consisting of only a character and a number (e.g. `R2`) is allowed.

6.2.7. Differences in Browser Behavior

We implemented 101 test cases. Together with the 544 test cases implemented by Schwenk et al. the framework contains 645 test cases. Out of those 645 test cases, 135 (20.93%) differ across browsers. Two of those differences can be attributed to the `canvas` element in combination with SVG and PNG images in Opera 47 and Comodo Dragon 58, both on Microsoft Windows 10. Both times these two browsers grant read access when the server responds with `Access-Control-Allow-Origin: your-sop.com` and `Access-Control-Allow-Credentials: true` while the actual request was no CORS request (crossorigin attribute not set). Ten more differences are caused by Mozilla Firefox 55 because the execution of JavaScript with the `javascript:` pseudo protocol is blocked (see Section 6.2.5). Safari 11.0 causes another 46 differences because the browser does not support the `video` element with `mp4` and `ogg` video files. Unexpectedly another 16 differences are caused by Safari 11.0. The same way as in Safari 9 the browser does not support embedding SVGs with the `embed` and `object` element. Two more differences can be attributed to Safari 11.0 on iOS because the browser does not support the `audio` element with a `mp3` file as ED. An additional difference is caused by Microsoft Edge 40 which does not allow to change the subtitles of a same-origin `track` element. The remaining 58 differences can be attributed to the `link` element with different CORS settings. Of those 58 differences, 12 differences appear because Mozilla Firefox 55 denies write access to the loaded CSS if the actual request is no CORS request but the server answers with CORS headers. Yet, actually those differences are not caused because Mozilla Firefox behaves wrongly, but because the other browsers do not behave according to the standard. The remaining 46 differences exist because Microsoft Edge 40 grants read or write access to the cross-origin CSS file while the other browsers deny read or write access.

Security. Those 46 differences are critical in terms of security because, regardless of the CORS settings, read access to the embedded cross-origin CSS file is always allowed. The biggest threat is posed by the read access to the remote CSS file without any CORS settings, i.e. a normal cross-origin request, which can be exploited as mentioned in Section 6.2.8. The two differences caused by Opera 47 and Comodo Dragon 58 can pose some threat but have to rely on a mis-configured server that answers with valid CORS headers to a non-CORS request.

In contrast to the above differences, the differences caused by Safari 11.0 are harmless. As they result from the ED not being loaded they cannot be exploited. The ten differences caused by Mozilla Firefox by blocking code execution with the `javascript:` pseudo protocol in `iframes` with certain sandbox settings are also harmless. Since they do not originate from different access rights to the HD they cannot be used to attack an user. The twelve cases in which Mozilla Firefox

55 denies write access are not security critical and are in line with the definitions in the CSSOM standard [26]. As pointed out this is a misbehavior of the remaining browsers which mostly are based on Chrome. We reported this misbehavior to Google and it was acknowledged as a correctness bug in issue 775525.
Those browsers, excluding Edge 40, grant write access to the embedded cross-origin CSS file as long as the original request is no CORS request, even if the server answers with CORS headers. If the request was a CORS request they grant read and write access according to normal CORS behavior.

6.2.8. Cross-Origin Login Oracle Attack

When testing Internet Explorer 11 and Edge 20 Schwenk et al. [1] discovered a cross-origin login oracle attack which was acknowledged by Microsoft in the Microsoft Research Center (MSRC) case 32703. This attack is based on the fact that Internet Explorer 11 and Edge 20 grant full read access to a cross-origin CSS file without CORS. When testing the current Edge 40 we discovered that this attack still works.because Edge still allows full read access in above scenario. Therefore, the `cssRules` are still not null like in other browsers.

7. Conclusions and Future Work

In this chapter we are going to present the conclusions that we reached from our work also considering the previous work from Schwenk et al. [1]. Additionally we will give an outlook which topics can be covered by future work.

7.1. Conclusions

In our work we evaluated 645 test cases for both, native JavaScript code and JavaScript code executed with the `javascript:` pseudo protocol. We observed that in regards of the SOP-DOM both approaches lead to the same results. While this does not answer the question if those two scenarios are always the same, it at least proves that they are the same regarding the SOP-DOM.

From the 135 differences that we found slightly less than half can be attributed to differences in CORS implementations. Compared to the results from Schwenk et al. this is a great improvement in browser security. Back then of the 129 differences they observed the majority were caused by different CORS implementations. This not only demonstrates the value of the framework, but also shows that through the extension of the framework we could discover new differences. Even though those new differences are not relevant to security, there might be currently undiscovered differences that are relevant to security. Therefore, to increase browser security, it is important to further extend the framework.

7.2. Future Work

There are several aspects in which our work can be extended in the future.

- In future work we could extend the framework to also cover a more sophisticated SOP concept called suborigins [39]. This concept is similar to the SOP for `file:` URIs implemented by Mozilla Firefox [40] and addresses a design issue in the SOP raised by Tim Berners-Lee in 2015 [41].

- We could increase the test coverage by adding the `file:` and `data:` protocol. Up to the HTML5.2 standard the latter still inherits the origin of the active document in certain cases which could lead to a SOP bypass as show by Manuel Caballero [42].
- Another approach to follow would be to extend and modify the framework to prove that native JavaScript code and JavaScript code executed with the `javascript:` pseudo protocol are the same, not only regarding the SOP-DOM.
- We could extend the framework to also test the SOP(-DOM) for technologies like Adobe Flash or Silverlight.
- Future changes to HTML might lead to more elements being able to include cross-origin resources. Obviously such elements should be added into the framework to increase test coverage.
- We could automate the testing process. Currently, for every browser, a person is required to start test cases, export the results and finally add them into the overview. A future implementation may automate this tasks and might even evaluate the results directly reporting if there are new differences.

A. Appendix

A.1. Call Stack

```
//<![CDATA[
function sleep(timeout) { /* simple JavaScript sleep method */
    return new Promise(resolve => setTimeout(resolve, timeout));
}

function call(func, args = []) {
    if(!document.callQueue) {
        document.callQueue = Array();
    }
    document.callQueue.push([func, args]);
}

async function depleteQueue() { /* async: allows await to be used */
    if(!document.working || document.working == false) { /* ensure that only one instance of depleteQueue() is running */
        document.working = true;
    }
    else {
        return 0;
    }
    if(!document.free) {
        document.free = true;
    }
    var i = 0;
    while (i < document.callQueue.length) {
        if (document.callQueue[i]) { /* function name */
             var code = document.callQueue[i][0].toString(); /* get source code */
            while (document.free == false) /* as long as a called function is still running wait */
            {
                await sleep(10);
            }
```

```
31                document.free = false; /* We are going to call a
                   ↪ function, block execution -> only one test case
                   ↪ executes at a time */
32                /* implement different ways to execute a script below
                   ↪ */
33                <?php
34                if (isset($_GET['exec']) && $_GET['exec'] === 'js') {
35                    echo 'window.location = "javascript:" + code +
                       ↪ document.callQueue[i][0].name + ".apply(null,
                       ↪ document.callQueue[" + i + "][1]);";';
36                    echo "\n";
37                } else {
38                    echo 'document.callQueue[i][0].apply(null,
                       ↪ document.callQueue[i][1]);';
39                    echo "\n";
40                }
41                ?>
42                var startTime = Date.now();
43                while (document.free == false) /* wait to clean the
                   ↪ queue */
44                {
45                    await sleep(10);
46                    /* Following code can be used to abort overlong
                       ↪ executing (malfunctioning) test cases */
47                    if(Date.now() - startTime > 10000) {
48                        console.log("Aborting: " +
                           ↪ document.callQueue[i][0].name + " due to
                           ↪ too long execution time for: " +
                           ↪ window.location);
49                        document.free = true;
50                    }
51                    /* console.log("blocking: " +
                       ↪ document.callQueue[i][0].name + " for: " +
                       ↪ window.location); */
52                }
53                document.callQueue[i] = undefined; /* clean current
                   ↪ queue entry to prevent double execution */
54            }
55            i++;
56        }
57        document.working = false; /* current instance of depleteQueue
           ↪ finished */
58    }
59    //]]>
```

Bibliography

[1] J. Schwenk, M. Niemietz, and C. Manika, "Same-origin policy: Evaluation in modern browsers," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk

[2] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the incoherencies in web browser access control policies," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 463–478. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.35

[3] E. Z. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp, "Toward principled browser security," in *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 17–17. [Online]. Available: http://www.scs.stanford.edu/~deian/pubs/yang:2013:towards.pdf

[4] R. Baloch, "Bypassing mobile browser security for fun and profit." Black Hat Asia, 2016. [Online]. Available: https://www.blackhat.com/docs/asia-16/materials/asia-16-Baloch-Bypassing-Browser-Security-Policies-For-Fun-And-Profit-wp.pdf

[5] "Same origin policy bypass via getsvgdocument() method." [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=21338

[6] "window.open() method javascript same-origin policy violation." [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=30660

[7] G. S. Kalra, "Exploiting insecure crossdomain.xml to bypass same origin policy (actionscript poc)," Aug. 2013. [Online]. Available: http://gursevkalra.blogspot.de/2013/08/bypassing-same-origin-policy-with-flash.html

[8] W3C, "Same origin policy," https://www.w3.org/Security/wiki/Same_Origin_Policy.

[9] Mozilla, "Same-origin policy," https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[10] M. Zalewski, "Browser security handbook, part 2." [Online]. Available: https://code.google.com/archive/p/browsersec/wikis/Part2.wiki#Same-origin_policy

[11] Microsoft, "Client-side cross-domain security," https://msdn.microsoft.com/en-us/library/cc709423(v=vs.85).aspx.

[12] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Stutor, C. Wilson, and L. Wood, "Document object model (dom) level 1," Tech. Rep., Oct. 1998. [Online]. Available: https://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

[13] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 723–735. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies

[14] M. Zalewski, *Browser security handbook*. Google Code, 2010.

[15] M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*, 1st ed. San Francisco, CA, USA: No Starch Press, 2011.

[16] P. Stuttard, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*, 2nd ed. No Starch Press, 2011.

[17] A. van Kersteren, "Croos-origin resource sharing," Tech. Rep., Jan. 2014. [Online]. Available: https://www.w3.org/TR/2014/REC-cors-20140116/

[18] W. Alcorn, C. Frichot, and M. Orru, *The Browser Hacker's Handbook*, 1st ed. Wiley Publishing, 2014.

[19] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver, "Cookies lack integrity: Real-world implications," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 707–721. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/zheng

[20] A. Bortz, A. Barth, and A.Czeskis, "Origin cook- ies: Session integrity for web applications," 2011. [Online]. Available: https://www.abortz.net/papers/session-integrity.pdf

[21] C. Masone, K.-H. Baek, and S. Smith, *WSKE: Web Server Key Enabled Cookies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 294–306. [Online]. Available: https://doi.org/10.1007/978-3-540-77366-5_28

[22] "Html 5.1," W3C, Tech. Rep. [Online]. Available: https://www.w3.org/TR/html51/

[23] W3C, "Html 5.2," Tech. Rep. [Online]. Available: https://www.w3.org/TR/2017/CR-html52-20170808/

[24] Mozilla, "Navigator.registerprotocolhandler()," https://developer.mozilla.org/de/docs/Web/API/Navigator/registerProtocolHandler.

[25] Google and E. Bidelman, "Registering a custom protocol handler." [Online]. Available: https://developers.google.com/web/updates/2011/06/Registering-a-custom-protocol-handler

[26] W3C, "Css object model (cssom)." [Online]. Available: https://www.w3.org/TR/cssom-1/

[27] "Inconsistent cors implementation regarding css and the link element." [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=775525

[28] M. Johns, S. Lekies, and B. Stock, "Eradicating DNS rebinding with the extended same-origin policy," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 621–636. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/johns

[29] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 58–71. [Online]. Available: https://people.eecs.berkeley.edu/~tygar/papers/Karlof/Dynamic_pharming.pdf

[30] "The multi-principal OS construction of the gazelle web browser," in *Presented as part of the 18th USENIX Security Symposium (USENIX Security 09)*. Montreal, Canada: USENIX, 2009. [Online]. Available: https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/multi-principal-os-construction-gazelle

[31] A. Barth, *The Web Origin Concept*, IETF RFC 6454, Dec. 2011.

[32] "Dom," Tech. Rep. [Online]. Available: https://dom.spec.whatwg.org/

[33] WHATWG, "Fetch living standard," https://fetch.spec.whatwg.org/#cors-preflight-request.

[34] Mozilla, "Access-control-allow-origin," https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin.

[35] Microsoft, "javascript protocol," https://msdn.microsoft.com/en-us/library/aa767736(v=vs.85).aspx.

[36] Mozilla, "Globaleventhandlers.onload." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onload

[37] ECMA, "Ecmascript® 2017 language specification." [Online]. Available: https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[38] Mozilla, "Element.clientheight," https://developer.mozilla.org/de/docs/Web/API/Element/clientHeight.

[39] W3C, "Suborigins," https://w3c.github.io/webappsec-suborigins/.

[40] Mozilla, "Same-origin policy for file: Uris," https://developer.mozilla.org/en-US/docs/Same-origin_policy_for_file:_URIs.

[41] T. Berners-Lee, "Granularity of web sites, trust and the same origin policy," https://www.w3.org/DesignIssues/Security-Origin.html.

[42] M. Caballero, "Sop bypass / uxss – stealing credentials pretty fast (edge)," https://www.brokenbrowser.com/sop-bypass-uxss-stealing-credentials-pretty-fast/.