Same-Origin Policy: Evaluation in Modern Browsers

Jörg Schwenk, Marcus Niemietz, and Christian Mainka
Horst Görtz Institute for IT Security, Chair for Network and Data Security
Ruhr-University Bochum

Abstract

The term *Same-Origin Policy (SOP)* is used to denote a complex set of rules which governs the interaction of different *Web Origins* within a web application. A subset of these SOP rules controls the interaction between the host document and an embedded document, and this subset is the target of our research (SOP-DOM). In contrast to other important concepts like Web Origins (RFC 6454) or the Document Object Model (DOM), there is no formal specification of the SOP-DOM.

In an empirical study, we ran 544 different test cases on each of the 10 major web browsers. We show that in addition to Web Origins, access rights granted by SOP-DOM depend on at least three attributes: the type of the embedding element (EE), the sandbox, and CORS attributes. We also show that due to the lack of a formal specification, different browser behaviors could be detected in approximately 23% of our test cases. The issues discovered in Internet Explorer and Edge are also acknowledged by Microsoft (MSRC Case 32703). We discuss our findings in terms of *read*, *write*, and *execute* rights in different access control models.

1 Introduction

The *Same-Origin Policy (SOP)* is perhaps the most important security mechanism for protecting web applications, and receives high attention from developers and browser vendors.

Complex Set of SOP Rules. Today there is 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 [1] or browser vendors [2] is still incomplete. Our evaluation of related work has shown that the SOP does not have a consistent description – both in the academic and non-academic world (e.g., [15, 16, 5]). Therefore, recurrent browser bugs enabling SOP bypasses are not surprising.

SOP rules can roughly be classified according to the problem areas which they were designed to solve (cf. Table 1). It is impossible to cover all these subsets in a single research paper and even may be impossible to find a "unifying formula" which covers all subsets.[1] However, it is possible to cover single subsets, as previous work on HTTP cookies has shown [12]. Thus, we restricted our attention to the following research questions:

- *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?*

More precisely, we concentrate on a subset of SOP rules according to the following criteria:

- **Web Origins.** We use RFC 6454 as a foundation.
- **Browser Interactions.** We concentrate on the interaction of web objects once they have been loaded.

It is a difficult task to select a test set for SOP-DOM that has constantly evolved over nearly two decades. The SOP-DOM has been adapted several times to include new features (e.g., CORS) and to prevent new attacks. 15 out of 142 HTML elements have a URL attribute and may thus have a different Web Origin [17]. Additionally, sandbox and CORS attributes also modify SOP-DOM.

The Need for Testing. Amongst web security researchers, SOP-DOM is partially common knowledge, but not thoroughly documented. Although this means

[1] For example, the SOP rules for DOM access and HTTP cookies are inconsistent, because their concept of "origin" differs.

SOP Subset	Description	Related Work
DOM access (this paper)	This subset describes if JavaScript code loaded into one "execution context" may access web objects in another "execution context". This includes modifications of the standard behavior by changing the Web Origin, for example, using `document.domain`.	[1], [2], [3], [4], [5] , [6]
Local storage and session storage	This subset defines which locally stored web object ([name,value] pairs) may be accessed from a JavaScript execution context.	[7], [8]
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.	[9], [7], [8], [10]
Pseudo-protocols	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.	[8], [10]
Plugins	Many plugins like Java, Flash, Silverlight, PDF come with their own variants of a SOP.	[11], [8]
Window/Tab	Cross-window communication functions and properties: `window.opener`, `open()` and `showModalDialogue()`.	[8], [10]
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.	[12], [13], [14]

Table 1: Different subsets of SOP rules.

that most researches are familiar with many edge cases in SOP-DOM, especially those relating to attacks and countermeasures, it is likely that some of those edge cases will not be covered in this paper. Additionally, each individual researcher will be unaware of other edge cases, which may include novel vulnerabilities. For example, it is well known that JavaScript code from a different web origin has full read and write access to the host document; nevertheless, recently Lekies et al. [5] pointed our that there is also read access from the host document to JavaScript code, which may constitute a privacy problem.

Additionally, HTML5 has brought greater diversity to seemingly well-known HTML elements. For instance, the term "authority" used in RFC 6454 [18] may not be sufficient any more if we compare the power of SVG images [19] with the following quote from RFC 6454: "an image is passive content and, therefore, carries no authority, meaning the image has no access to the objects and resources available to its origin". Our evaluation shows that this statement is true for all image types if they are embedded via <**`img`**>. This statement does not hold if SVG images are embedded via <**`iframe`**> or <**`object`**>. Novel standards like Cross-Origin Resource Sharing (CORS, [9]) also influence access rights granted by the SOP. To be able to keep the implementation of the SOP consistent through all these extensions, a formal model is needed.

Our Approach. The aim of this paper is to develop a comprehensive testing framework for SOP-DOM (see Figure 1). The SOP restricts access of active content like JavaScript on other components of a web page. We also apply it to CSS code by interpreting the style changes

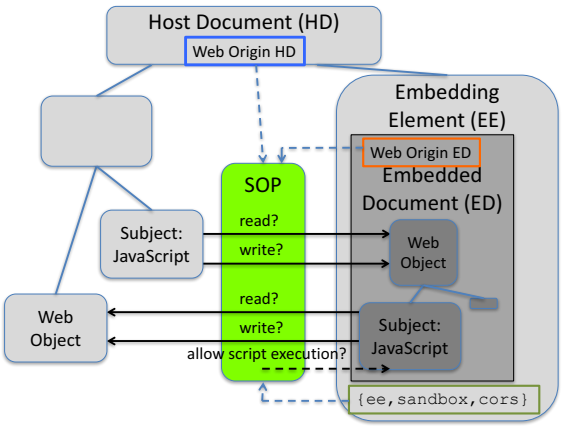


Figure 1: Setup for our test cases for SOP DOM access. The embedding element (`EE`) itself belongs to the host document (`HD`).

imposed by CSS code as write access on certain DOM elements.

We define "comprehensive" by meaning the coverage of all *interesting* edge cases. We thus do not cover all 15 elements with URI attributes but only a selected subset according to importance and interesting properties. Instead, we include "URL-attribute-like" constructions in the <**`canvas`**> element. We also do not restrict the test set to full DOM read or write access (which easily could have been automated to cover more test cases) but instead, also concentrate on the more interesting cases of *partial* read and write access.

Our tests thus cover only a representative sample of SOP-DOM, but this sample was chosen to cover each

known edge case of SOP-DOM. To cover these edge cases, many of the 544 test cases were designed manually. We use these representative test results to discuss if classical access control models like DAC, RBAC and ABAC are applicable to SOP-DOM. We reformulate access restrictions in terms of read, write, and execute rights granted to an embedded document (ED) contained in the HD and vice versa. We thus highlight the importance of the EE in defining the access rules of the SOP.

Testbed. We show the applicability of our test methodology for SOP implementations in current web browsers by providing a testbed at `www.your-sop.com`, where proof-of-concept HTML, JavaScript, and CSS code is given for each test case. Our tool consists of more than 10,000 lines of code covering 544 test cases with five types of ED and ten types of EE. The tests are created in a semi-automatic manner. For each EE to be tested, we automatically load the ED with possible CORS/sandbox attributes successively. We did not choose a fully-automatic test creation because this would lead to an overwhelming number of errors. Combining each EE with all possible attributes would lead to errors; for example, neither `<img data=".">` nor `<object src="...">` are semantically correct. In addition, there is no universal access from HD to ED and vice versa; for example, accessing the SVG ED can be achieved with a dedicated `getSVGDocument()` method.

Limitations. We describe a subset of the SOP for the interaction of web objects that are loaded into the browser. Zalweski describes other contexts such as cookie, local storage, Flash, XMLHttpRequest, Java, Silverlight, and Gears [8]. For each of them a different SOP is used. For example, Zheng et al. [12] have analyzed the SOP for HTTP cookies in-depth; here the SOP takes the path contained in an URI into account, which is an extension of the Web Origin concept. An in-depth discussion of the limitations of our approach can be found in Section 5.

Contributions. We make the following contributions:

- We systematically test edge cases of the SOP that have not been previously documented like the influence of the embedding element, and the CORS and sandbox attributes.
- We provide a testbed where the SOP implementation of a browser can be automatically tested and visualized.
- We used this testbed to extensively evaluate our model in 544 test cases on 10 modern browsers. More than 23% of the test cases revealed different SOP-DOM access rights implemented in at least one of the tested browsers. Our ABAC model provides a systematic way to describe these differences.
- We prove that a better understanding of SOP-DOM is useful by describing a novel CSS-based login oracle attack for IE and Edge, which we found using the ABAC rules for cross-origin access to CSS.
- We critically discuss the applicability of standard access control models like DAC, RBAC, and ABAC to SOP-DOM.

2 Foundations

Document Object Model (DOM). DOM is the standardized application programming interface (API) for scripts running in a browser to interact with the HTML document. It defines the "environment" in which a script operates. The first standard (DOM Level 1) was published in 1998 and the latest published version is DOM Level 3 (2004). The DOM standard is now a "living standard" since it has to be adapted to each new HTML5 feature, resulting the DOM Level 4 to remain in the "work in progress" stage.[2]

A browser's DOM includes more objects and properties than just the pure HTML markup, as shown in Figure 2. These objects can be accessed through a variety of different methods. For example, the iFrame element can be accessed through predefined selector methods like `document.getElementByID("ID1")`. The DOM structure does not necessarily match the markup structure. Although the `<iframe>` element from Figure 2 is a child element of the HTML document, there is no property `document.frames[0]`; instead, there is only `window.frames[0]`.

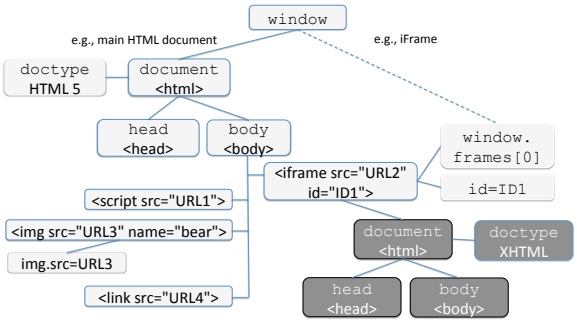


Figure 2: Small extract from the DOM.

[2] `https://dom.spec.whatwg.org/`

To access and modify the DOM, JavaScript code can be used. Each JavaScript script runs in a specific DOM *execution context*. Consider Listing 1 as an example. If this small HTML file is opened in a web browser, first the <iframe> element will be parsed. After that the iFrame's source code from Listing 2 will be loaded and the alert function contained therein will be executed. The <script> element will then be parsed and the (second) alert function will be executed.

```
1 <html><head><title>a.html</title></head>
2 <body><iframe src="b.html" />
3 <script>alert(document.location)</script>
4 </body></html>
```

Listing 1: Code of http://a.org/a.html

The two alert pop-up windows, triggered by the two script elements, will display different URLs because they are acting in different DOMs. The alert window called in Listing 1 will display the URL http://a.org/a.html, whereas the alert window in Listing 2 will display the URL http://a.org/b.html.

```
1 <html><head><title>b.html</title></head>
2 <body><script>alert(document.location)</
    script>
3 </body></html>
```

Listing 2: Code of http://a.org/b.html

Cross-Origin Resource Sharing (CORS). Using XMLHttpRequest, a web page may send arbitrary HTTP requests to any webserver. This is different from just opening an URL or submitting an HTML form since with XMLHttpRequest the web page has full control over all HTTP headers. To restrict such potentially dangerous queries, XMLHttpRequest is restricted by default to the domain from which the calling document was loaded (same-domain). To enable controlled cross-domain requests, the CORS standard [9] was developed. It works as follows: *a)* in a preflight request,[3] the browser sends an origin header (Origin: http://a.com) to the target web service requesting CORS privileges. *b)* the target server may now answer with an error message (access denied) or with a CORS header, such as Access-Control-Allow-Origin: http://a.com, to grant the access. Instead of a domain name, the CORS header may contain a wildcard ($\star$) to grants access from arbitrary domains.[4] Although CORS is designed to relax the Same-Origin Policy (SOP) in a secure manner, there are many cross-origin resources used in the web (e.g., scripts, stylesheets, images, iFrames) that can be loaded without CORS and without XMLHttpRequest. However, in HTML5 some elements (e.g., <img>) may have crossorigin attributes which invoke CORS and subsequently modify the SOP access controls.

[3]The preflight request can be skipped in simple cases

[4]This additionally denies the use credentials such as cookies in a CORS request.

3 Methodology

3.1 SOP-DOM Attributes

The *Same-Origin Policy for DOM Access (SOP-DOM)* controls the access of a *subject* – typically JavaScript code – to a *web object* (e.g., an HTML form). The subject may be located directly in the HD or in an ED. The element that loads the ED is called the EE (cf. Figure 1). Both HD and ED have a Web Origin. The Web Origin of ED is defined by src or similar attributes of EE (e.g, dynsrc, lowsrc, and srcset).

SOP-DOM is often described as a boolean switch which either allows interaction between HD and ED in the same-origin case or blocks access in case of different web origins (e.g., Karlof et al. [15]). In reality, SOP-DOM is more complex; some EEs like <img> block almost all access even in the same-origin case, some EEs like <script> allow full read and write access (in one direction) even in the case of different origins, and some EEs like <iframe> (in the cross-origin case) only grant partial access. Furthermore, access decisions may be influenced by additional attributes like CORS or sandbox.

In our investigations, we have used five values as our *test attributes*, two of which contribute to the definition of Web Origin. These attributes are summarized in Table 2.

Notation. In this paper and in our testbed, we use $\overline{\text{HD}}$ and $\overline{\text{ED}}$ to denote that HD and ED share the same Web Origin, and $\overline{\text{HD}}$ and $\underline{\text{ED}}$ if the origin differs. If cross-origin and same-origin behavior are identical we use HD and ED to save space.

Coverage and Restrictions. The SOP-DOM is very complex, because with each newly considered attribute, the number of test cases may grow by a factor proportional to the number of possible attribute values. Thus, it should be clear that it is nearly impossible to test and describe the whole SOP-DOM in one research paper.

Since 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. Our focus is on ee, where we considered HTML elements with URI attributes and

Attribute	Description	S/O/E	HD/ED
`protocol`	protocol of URL, value of `location.protocol`	S,O	HD+ED
`domain`	domain/hostname of URL, value of `location.hostname`	S,O	HD+ED
`ee`	type of `EE`	S,O	ED
`cors`	value of the CORS attribute of the `ee`, i.e., `ee.crossOrigin`	O	ED
`sandbox`	value of `sandbox`	S,O,E	ED

Table 2: SOP-DOM Attributes. *S* denotes subject attributes, *O* object attributes, and *E* denotes attributes which may also be set independent of the markup (e.g., through a HTTP security policy like Content Security Policy (CSP)).

properties. By systematically analyzing the provided list of the W3C [20] and the WHATWG [21], we picked the representative HTML elements <**script**>, <**img**>, <**canvas**>, <**link**>, <**iframe**>, <**object**>, <**embed**>, and <**link**>. We have also examined CORS (the value of the `crossorigin` attribute) and `sandbox`, as a proof-of-concept, to show that these attributes do have an influence on the SOP-DOM. More limitations of our approach are discussed in Section 5.

3.2 Access Control Test Cases

Web Object Structure. Web objects may have an internal DOM structure, as it is the case with iFrames or SVG images. In this case, we can use standard DOM selector methods to test for read and write access.

Other web objects do not have a DOM structure (e.g., JPEG and PNG images). In this case, we define the type of access for each such web object separately (e.g., single pixel access for JPEG) and use adapted code examples.

Distinguishing Full and Partial Access. In case that the object has an internal DOM structure, we define *full* access if we can access arbitrary parts of the DOM by standard selectors like `getElementbyID()`. We define *partial* access as only being able to read, or only being able to write some specific properties (e.g., `window.top.location`).

If the web object does not have an internal DOM, we always specify exactly what we can read or write. To name one example, single pixels in images or the source code of scripts.

Full Read and Full Write Access. Supposing that JavaScript code has DOM read access, it typically also has write access using some DOM methods (e.g., `innerHTML`). We have tested this by first writing into a particular DOM property, and then by reading the same property to verify whether it contains the newly written value. For full DOM access, we successfully verified that any DOM property which can be read, can also be written. In our proof-of-concept implementation, a script contained in the `ED` tries to read DOM properties from HD and vice versa. To test full DOM access, we inter alia use the code depicted in Listings 3 and 4.

```
<html>
<head>HD from HD.org</head>
<body>
<script>
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.org/ED.mime"></EE>
</body>
</html>
```

Listing 3: Host document (HD) verifying full read access.

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

Listing 4: Embedded Document (ED) for verifying full read access.

Partial Access. Many partial access rules have been added to browser implementations over the years in order to implement new features, or to defend against new attacks. The best-known examples are certainly the DOM properties of an iFrame's top frame that are used to build JavaScript framebusters to defend against UI Redressing [22].

Partial access cannot be tested systematically. Instead, we relied on our knowledge from pentesting, blog posts

of security researchers, and – in some cases – on intuition. Please note that our goal was not to give a full list of partial access rules, but only to document the variety of such rules.

Partial Read: Examples. An example for partial read (and write) access is the pixel-based manipulation of images with the help of CANVAS (e.g., via `context.getImageData`).

Lekies et al. [5] underlined that every script executed within the same web document is able to read global variables created by another script. However, local variables inside a function cannot be accessed unless their values are not explicitly returned by the function. This illustrates clearly that we have partial read access.

As an edge case example for partial read access, CSS in combination with browser features like plain HTML and inactive SVG files can be used to extract some values from the SOP-DOM [23].

Partial Write: Examples. Partially writable are properties like `parent.location.path` and `parent.location.hash`. In the past `location.hash` was used to share data cross-origin. Nowadays, this feature can be replaced by using PostMessage or CORS and write access to `parent.location` can be restricted in iFrames by using the `sandbox` attribute.

Execute. Current sandboxing concepts consider blocking JavaScript execution but not CSS execution. To be consistent with this view, we say that an `EE` grants execute rights to an `ED` when JavaScript code contained in the `ED` can be executed. For example, when EE=<**`iframe sandbox`**>, then the execution of JavaScript is blocked. We verified this by using script execution to send a PostMessageAPI message to HD.

4 Evaluation

We implemented a testbed as a web application which automatically evaluates the SOP implementation of the currently used browsers. Additionally, it displays the results of 10 tested browsers from six different vendors and highlights the differences between them. Our testbed is publicly available at *`www.your-sop.com`*.

4.1 Experiment Setup

We evaluated the following elements with `src` attributes and determined their Alexa 500 rank through an analysis of the Alexa Top–500 start pages. The results are (rank; domains; occurrences): <**`script`**> (3; 460; 12,625), <**`link`**> (8; 453; 5,197), <**`img`**> (11; 439; 24,015), and <**`iframe`**> (21; 261; 1,406). To name an example, the `script`-element was the third most common element listed on 453 out of 500 domains with a total of 12,625 findings. The elements <**`object`**> or <**`embed`**> are not listed under the TOP-30 elements.

Our testbed executes all tests on a single website so that tests can be easily repeated with different browsers. It uses one of the previously mentioned `EE`s and loads an external `ED` via its dedicated attributes. For example, the <**`img`**> elements uses the `src` attribute; however, the <**`object`**> elements uses the `data` attribute. If the element supports CORS, we created a test as follows; we used the three attribute cases, (1.) no `crossorigin` attribute is set, (2.) `crossorigin="use-credentials"`, and (3.) `crossorigin="anonymous"`. For each attribute, we created a test that receives an HTTP response header `Access-Control-Cross-Origin` (1.) set to a specific domain `your-sop.com` or `other-domain.org`, (2.) set to the wildcard `*`, (3.) or not set at all. In addition, the HTTP response header `Use-Credentials` is once set for each test to (1.) to `yes`, to (2.) `no`, (3.) and not set. The immense number of combinations lead to a significant number of test cases if CORS is supported.

Each test loads an external resource (`ED`), first from the same domain (`your-sop.com`), and then from a different one (`other-domain.org`). When retrieved through any browser, the SOP decisions of the currently used browser are presented in different overview tables. Since the exact method to access specific objects from `ED` to `HD` – and vice versa – differs with each test, its source code can be inspected by hovering on the result field in the table on the testbed website (cf. Figure 3).

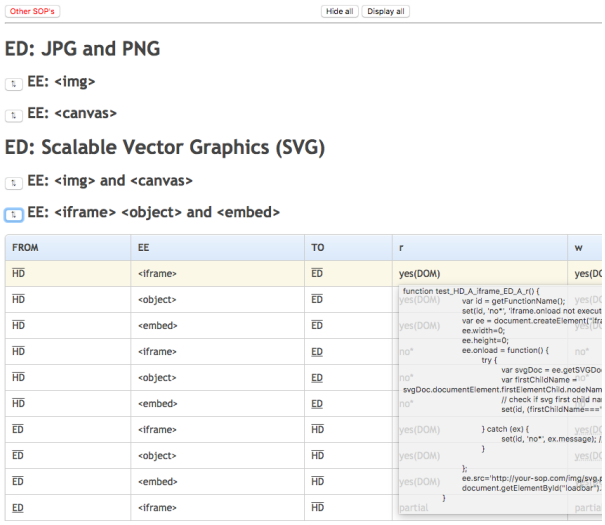


Figure 3: Screenshot of our `your-sop.com` testbed.

Using the testbed, we evaluated the SOP of ten differ-

ent browsers, including Google Chrome, Mozilla Firefox, Internet Explorer, Edge and Safari. We added a feature to export all test results in a JSON file. We then used this feature to add a comparison table of different browser behaviors. It displays all test cases and SOP decisions of all browsers at once or can only highlight the differences. Figure 4 shows a small part of the comparison of different SOP implementations.

4.2 Results

In the following, we describe the general outcome of our testbed. The results are structured by the type of the embedding element (EE).

Images. An <**img**> element acts like a sandboxed iFrame; read and write access is blocked in both directions, even in the same-origin case. Script execution is blocked in the ED; even if the ED is an SVG containing some JavaScript code, the script is not executed. This behavior holds for both the same-origin and cross-origin case.

If we use <**canvas**> as the embedding element EE[5], we can get read access to pixels in JPG, PNG and SVG images if loaded from the same origin. This allows reading out the color of each pixel and it may be critical in some security contexts like JPG- or PNG-based CAPTCHAs. Here, an attacker could use CANVAS to automatically read out the displayed token.[6]

SVG files are basically XML-based vector graphics. Please note, that unlike <**img**>, the <**svg**> element does not support a `src` attribute to load an external SVG file. If embedded into a website with <**img**> or <**canvas**>, they behave as if they were bitmaps; thus, we can only read pixels. It is also possible to include SVGs in EEs like <**iframe**>, <**object**>, and <**embed**>. Then the DOM of the SVG is mounted into the HD and we can access it fully, and additionally read all SVG vector instructions.

Scripts. Cross-origin loaded JavaScript code via <**script src**="..."> is a well-known special case in the SOP; it is treated as if it had been loaded from the same origin. Technically, a script loaded by the `src` attribute is appended to the `document.scripts` array in the HD's DOM, independent of the domain on which the script is hosted. In the <**script**> case, no access restrictions are imposed by the SOP: we have full read-/write access from the ED to the HD, and execution rights from HD to ED.

For the read/write access from the HD (subject) to the ED (JavaScript, object), this is less well-known. It is clear that we cannot change the content of the external file (write), but we can overwrite functions defined in this external file, and thus change the functionality of the loaded code. We are able to read variable values and the source code of defined functions[7]. However, there are some exceptions: we cannot read `var cnt = 2+5;` but we can read the `cnt`'s value `7`. We can also read the complete line of code if it is contained in a function (cf. [5]). Thus, we have *partial* read/write access from the HD to the ED.

Style Sheets. External CSS code can be loaded via the embedding element <**link**>. In the case where the CSS code is loaded from the same origin, we can read the complete source code. If the CSS file is loaded cross-origin, we can only read the source code if proper CORS values are set. An exception is MS IE/Edge, which allows read access in every case (see Section 4.3 for details).

Write access for CSS code is defined by the ability of CSS to change the visual display of a web object. Since this is the desired behavior, write access from the ED to the HD is independent of the web origin.

Frames. For <**iframe**> (without `sandox` attribute) we have full read/write access in both directions in the same-origin case, and partial read/write access in the cross-origin case.

The cross-origin case from $\overline{\text{ED}}$ (subject) to $\underline{\text{HD}}$ (object) is of special interest; we have partial read/write access. Some properties that can be read are: `top.length` (number of frames/iFrames in HD), `top.closed` (boolean value if HD is closed), `top.opener` (reference to opener HD in the event of a popup). Although this is a very limited read access, we have a side-channel allowing us to read some cross-origin information. Especially the first property is noteworthy; it allows to get the number of frames/iFrames that are contained in the HD. We also have partial write access in this case; for example, to the `top.location` property (a property that we can only write, but are unable to read).

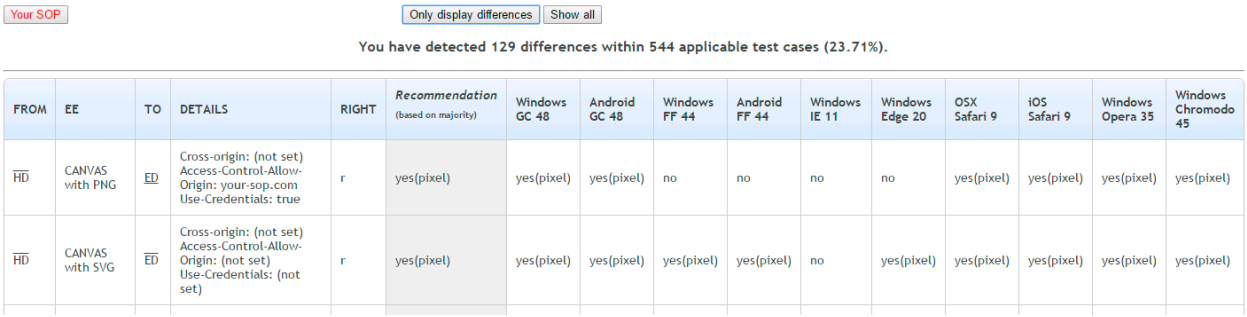
Similar results hold for the other direction (subject HD to object ED) in the cross origin case. In this case, the properties are accessed via the `window.frames[]` array (instead of `top`).

Sandboxed Frames. The origins of the SOP-DOM lie in the necessity of a clear separation of two HTML documents, shown by several attacks over the last ten years

[5] See the example on `https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Pixel_manipulation_with_canvas`

[6] http://ejohn.org/blog/ocr-and-neural-nets-in-javascript/

[7] For example, by using `Object.getOwnPropertyNames(window)`, we can read all properties defined in the `window` object



FROM	EE	TO	DETAILS	RIGHT	*Recommendation* (based on majority)	Windows GC 48	Android GC 48	Windows FF 44	Android FF 44	Windows IE 11	Windows Edge 20	OSX Safari 9	iOS Safari 9	Windows Opera 35	Windows Chromodo 45
HD	CANVAS with PNG	ED	Cross-origin: (not set) Access-Control-Allow-Origin: your-sop.com Use-Credentials: true	r	yes(pixel)	yes(pixel)	yes(pixel)	no	no	no	no	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)
HD	CANVAS with SVG	ED	Cross-origin: (not set) Access-Control-Allow-Origin: (not set) Use-Credentials: (not set)	r	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)	no	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)

Figure 4: Evaluation result by comparing 10 different browsers.

[24, 25, 26]. However, a complete separation between two HTML documents is often not possible; for example, to allow UI redressing countermeasures with JavaScript frame-busters [22].

To allow a better separation between the iFrame `ED` and the `HD`, *sandboxed iframes* were introduced [27]. We limited our evaluation to the attribute values that directly affect our read, write, and execute results: `allow-scripts`, `allow-same-origin`, `allow-top-navigation`.

The `sandbox` attribute is a special case that is discussed in Section 7.

Recommendations for Browser Vendors. From the perspective of a browser vendor, it is interesting to know how the results of our tool can be used to identify bugs and therefore potential vulnerabilities. In our analysis, we have automatically compared each SOP-DOM difference with the behavior of all other browsers. In case that at least one browser grants SOP-DOM access that the other browsers restrict, a browser vendor should have a closer look on this test case. We recommend to adjust the SOP-DOM behavior to the majority of other browser behaviors for reasons of clarity. For each test, our website recommends a result, which is based on the majority of all ten tested browsers (see Figure 4). Because our testbed includes browsers of different vendors (e.g., Apple, Google, Mozilla, Microsoft), we believe that this might be a representative SOP-DOM result.

4.3 Different Browser Behaviors

We implemented 544 test cases and 129 of these cases differ across ten tested browsers (23.71%).[8] We identified three subsets of different browser behaviors.

First, more than 35% of the identified differences could be attributed to <**canvas**> and PNG/SVG. In contrast to the other seven browser tests that allow partial read access with the help CORS from `HD` to `ED` cross-origin, FF, IE, and Edge do not allow read access in the following CORS cases of <**canvas**> with SVG and PNG: `Access-Control-Allow-Origin : your-sop.com` (`ED` sets the domain of `HD`) and `Use-Credentials: true`. Irrespective of CORS, <**canvas**> and SVG have 44 differences that are based on a denied access in IE 11.[9]

Second, over 12% of the test cases show differences between Safari 9 and the other browsers by looking on <**object**> and <**embed**> elements that load SVG files. Safari 9 does not show an SVG if it is loaded by code like <**object data**="image.svg"></**object**>. Therefore, JavaScript code contained in the SVG file cannot be executed. It needs an additional `type` attribute with the value `image/svg+xml` such that JavaScript execution is allowed. Since Safari 10.1 Apple has changed their implementation and both elements behave similar to the other browsers. The attribute `type="image/svg+xml"` is no more required.

Third, over 51% of the test cases show different behaviors because of <**link**>. Nearly all the cases have different CORS implementations. CORS thus shows that a relatively new and complex technology leads to different interpretations of "well-known" web concepts like SOP.

Similarly to Chromium's testbed that have been applied to other browsers to find bugs, our testbed could be used and extended by browser vendors and security researchers to identify browser differences leading to exploits.[10]

4.4 Cross-Origin Login Oracle Attack.

We have detected one browser difference due to IE/Edge, which does not need CORS. In this case, IE/Edge allows us to read CSS rules cross-origin while other browsers do not allow such access.

[8] `http://www.your-sop.com/stats.php`

[9] We have communicated these differences to Microsoft and it seems that they have fixed them in the newest browser versions.

[10] `https://github.com/thomaspatzke/BrowserCrasher`

By using the difference that was detected in case of <link>, we show that dynamically generated CSS files can be abused to attack the user's privacy. In case of CSS code from different origins, IE/Edge behaves differently from GC and FF; it does not set DOM properties like cssRules to null. Therefore, an attacker is always allowed to read the CSS code regardless of its origin. This allows us to build a novel login oracle:

- Suppose a webserver delivers different CSS files, depending on whether the user is logged in or not.
- The attacker's website consists of the EE <link> loading the victim's CSS code (ED).
- Though HD has another origin than ED, the attacker's JavaScript code in HD automatically reads all CSS rules. By comparing the CSS code with CSS code of a logged out user, the attacker can determine the logged in state.

We verified our login oracle with the startpage service start.me (ED); an attacker is clearly able to decide whether a user is logged in or not. This attack is similar to [5]. We have informed the website administrators about this vulnerability. Microsoft (Research Center, MSRC) acknowledged this bug (Case 32703) and the fix will be incorporated into a future version of IE/Edge.

5 Limitations

Even if we restrict our attention to SOP-DOM, the Same-Origin Policy has a very large scope. We have 15 HTML elements with src attributes, and several more with a similar functionality (e.g. <canvas>). There are six different sandbox attributes, and they (e.g., the CORS attribute) may be influenced by HTTP-based security policies like CSP. There are many different ways how to embed a document of a given MIME type into a webpage (e.g., SVG via <img> or <iframe>), and there are many different MIME types with and without a DOM structure to consider. There are pseudoprotocols like data: and about:, which have different Web Origin definitions. There is also a large number of DOM properties which could be tested for partial access.

Covering all interactions within this scope would result in an exponential number of test cases, which cannot be covered in one research paper. For example, Zalweski [28] lists four classes of common URL schemes (e.g., document-fetching and third-party) consisting of different subclasses (e.g., browser specific schemes like vbscript, firefoxurl, and cf). Moreover, it is possible to register self-defined handlers for particular schemes via registerProtocolHandler. In this section, we therefore discuss several technologies that we excluded from our research and give a rationale for these decisions.

Link. One technical limitation of our evaluation framework is that we used the <link> element only to load CSS. We did not consider, for example, HTML imports via <link rel="import"href="data.html">. An interesting novel technology that is highly under development are Service Workers [29]. They can, for example, be loaded using <link rel="serviceworker"href="worker.js">. However, it is currently "an experimental technology" according to Mozilla [30], although they are used by many websites (e.g., Google and Twitter). Our evaluation does not cover Web Workers [31]. This technology allows running a JavaScript in different context; for example, there is no window object reference. For this reason, we excluded it.

SVG. We only covered <svg> as an ED which directly embeds the JavaScript code for testing read/write access. It is also possible, to use <svg> as a HD; for example, an external JavaScript can be loaded by using <svg><script xlink:href=".."></svg>. Our testbed always uses an HTML document as HD.

JavaScript. We only cover a small, but hopefully representative, set of DOM properties. Our testbed only covers the location property, but sub-properties such as location.hash or location.path were not analyzed. The same holds for the window.name property, which is well-known to be writable across origins.

A design decision for our testbed was to be able to easily execute all test simultaneously. Therefore, only one *index.html* is capable to run all 544 tests with only one click by the user. For this reason, we excluded popups and the corresponding window.opener property.

Other Mime Types. Our testbed is limited to HTML, JavaScript, CSS, and SVG. For example, it would be interesting to investigate PDF, which can also include JavaScript code. There are many more *active* MIME types, such as Flash or ActiveX, which should be addressed in further research.

Pseudoprotocols. We excluded pseudo protocols (e.g., about:, chrome:) and Data and JavaScript-URIs from our tests, because in a (possibly outdated) overview, Zalewski [28] already pointed out that there are different Web Origin assignments in different browser implementations. However, extending the testbed to selected pseudoprotocols is future work.

6 Related Work

Different SOP Contexts. Jackson and Barth [32] discussed different SOP contexts, and showed vulnerabilities introduced by the interaction of these contexts. Zheng et al. [12] describe in detail the SOP for HTTP cookies. They also presented bypasses based on subdomains. Session integrity problems resulting from the cookie context SOP are discussed by Bortz et al. [13]. Karlof et al. [15] and Masone et al. [14] describe refined origins for the cookie SOP: they replaced the domain name with a server's X.509 certificate and public keys. Thus, they are able to use different cookies for different servers on the same domain. Singh et al. [7] analyzed in-coherencies in web browser access control policies by showing that there are different definitions of Web Origins; there are web-origins for DOM objects, localStorage, and XMLHttpRequest, as well as other definitions for cookies (domain, path) and the clipboard (user).

SOP Enhancements. Wang et al. [33] proposed their secure browser Gazelle with a multi-principal OS architecture and showed how to implement extended access control policies. Chen et al. [34] analyzed browser domain-isolation bugs and attacks. They proposed "script accenting" as a defense mechanism so that frames cannot communicate if they have different accents.

SOP Bypasses. Ways to bypass SOP restrictions are regularly published in the academic and non-academic areas. Jackson et al. [35] and Johns et al. [36] discuss DNS rebinding attacks (which manipulate Web Origins and thus disable the SOP) and proposed mitigation techniques. Oren and Keromytis [16] used Hybrid Broadcast-Broadband Televisio (HbbTV) to bypass the SOP. In contrast to websites, HbbTV data does not have a origin. This characteristic allows an attacker to inject malicious code of his choice into any website, which are loaded via the HbbTV data stream. Lekies et al. [5] are using dynamically generated JavaScript files to attack the privacy of a victim. Singh et al. [7] describe major access control flaws in browsers. Complicated side-channels have been abused to read DOM properties in [23].

Various non-academic publications describe ways to bypass the SOP. Jain [37] states that Safari v6.0.2 does not have SOP restrictions in case the `file` protocol is used. In 2010, Stone [38] showed that UI redressing can be used to bypass the SOP. Even if the SOP is restricting access on the script level, copy-and-paste as well as drag-and-drop actions are not restricted. In 2012, Heyes [39] showed that the location of a window can be accessed cross-origin in FF; however, this should not be allowed. Three years later, Bentkowski demonstrated with CVE-2015-7188 that FF's $\leq$42 SOP can be bypassed by adding whitespace characters to IP address strings.[11] In 2016, Ormandy [40] showed that Comodo's browser Chromodo disables, at least partially, the SOP and thus Chromodo "actually disables all web security". There are also SOP bypasses via Java applets [41], Adobe Reader [42], Adobe Flash [11], and inter alia Microsoft Silverlight [10].

Formal approaches to Web Security. Yang et al. [6] propose to describe the SOP in terms of Information Flow Control. Akhawe et al. [43] have a much broader scope and describes the backbone of a formal model for the Web itself.

Other Approaches. Crites et al. [44] proposed the abstraction and access control model OMash, as a replacement of SOP. Barth et al. [45] proposed a browser extension system for protecting browsers from extension vulnerabilities. They reused the SOP to isolate extensions from attacks, which needs inter alia access to browser internals and web page data. Chen et al. [46] described an opt-in app isolation mechanism that acts like the user is executing different browsers. Even if the attacker is able to act in the same origin, the users credentials might only be available in a logged-in state which is isolated. Stamm et al. [47] proposed CSP, which is implemented in all modern browsers. In CSP, code injection attacks are mitigated through restrictions imposed on code origins (whitelisting of allowed origins), and through abandoning inline code. Jackson and Wang [48] introduced Subspace as a cross-domain communication primitive allowing communication across domains.

7 Access Control Policies

Since SOP-DOM restricts access of subjects (mainly JavaScript code) to web objects, we think that an appropriate formal model could be found amongst the class of access control policies. Access control policies restrict the access of subjects from a set S (humans, machines or code) to objects from a set O. In the following, we discuss how well the three main classes fit our findings.

SOP-DOM is a global access control policy regulating access between websites throughout the Internet; however, decisions through the SOP-DOM can only be made on that which is locally available. This data includes the web origins of the different subjects and objects, the HTML markup (elements and attributes), and more recently, security policies communicated through HTTP headers like CORS, CSP, X-Frame-Options, and others.

[11] `https://www.mozilla.org/en-US/security/advisories/mfsa2015-122/`

In SOP-DOM, the set O of objects may contain any element or property of the local DOM of the web page. Typically, access rights granted to two objects o_1 and o_2 should only differ if the Web Origins of these two objects differ. The set S of subjects could be defined as $S = O$; however, this would only result in numerous "inactive" subjects which do not need any access rights since they never access any other objects (e.g., text nodes). We therefore restrict the set S to "active" objects, where the definition of "active" still awaits a mathematically precise definition. We include all script objects in S and all CSS code; however, since the discovery of scriptless attacks [23], there may be a need to extend this definition.

7.1 Discretionary Access Control (DAC)

DAC access control is well-known from operating systems (OSs); each user has a login name and the OS decides if this particular user has access to a certain resource (e.g., a data file or network printer). Each resource also has a unique name; therefore, S and O contain the names of users and resources. Another example is email encryption in which read access is granted on the basis of the RFC 822 email addresses of the recipients.

Definition 1 *In DAC, access rights are directly assigned to subjects: the policy set P is a subset of $S \times O$, and subject s has access to object o if $(s,o) \in P$.*

In the WWW, each subject from S and each object from O can be assigned a unique name, which is the URL at which it can be found. Thus, this part would fit in the DAC model. However, there is no global "web operating system" which keeps track of all possible pairs in $S \times O$. Instead SOP-DOM uses only a part of this name in its access decisions, namely the Web Origin.

Some sources trivialize RFC 6454 in the sense that they state that read and write access are only possible if the Web Origins of the subject and object are identical. If this was true, it would be a perfect fit for DAC and a very simple global DAC policy could be formulated as follows:

$$(s,o) \in P \iff origin(s) = origin(o).$$

This however is simply incorrect, since in many cases $(s,o) \in P$ even if $origin(s) \neq origin(o)$, for example, in case a script s was embedded via a `<script>` element, or if s is contained in a sandboxed iFrame with top-level frame access.

Unfortunately, the elegant DAC-based definition of SOP-DOM via web origins does not fit.

7.2 Role-Based Access Control (RBAC)

RBAC is often used in distributed environments as an abstraction to improve the manageability of access control rules. By means of example, the role *system administrator* may be assigned to different subjects over time or even periodically, and this role has many important access rights. Instead of assigning, revoking, and reassigning these access rights periodically to individual subjects, the access rights are assigned to the *role* "system administrator", and this single role is assigned, revoked and reassigned over time.

Definition 2 *In RBAC, subjects are assigned to* roles *from a set R, and access rights are assigned to roles: $P_1 \subseteq S \times R, P_2 \subseteq R \times O$, and s has access to o if there exists a role r such that $(s,r) \in P_1$ and $(r,o) \in P_2$.*

In typical RBAC installations, access rights to individual resources are assigned manually by the system administrator. This is problematic for SOP-DOM, since access policies must be created automatically. We discuss the following variant of RBAC where roles are assigned to both subjects and objects, and access decisions are based on both roles only.

Definition 3 *In enhanced RBAC (eRBAC), subjects are assigned* subject roles *from a set R_S, objects are assigned* object roles *from a set R_O, i.e. $P_S \subseteq S \times R_S, P_O \subseteq O \times R_O$. Access rights are assigned between roles: $P \subseteq R_S \times R_O$. So subject s has access to object o if there exists roles $rs \in R_S$ and $ro \in R_O$ such that $(s,rs) \in P_S, (o,ro) \in P_O$ and $(rs,ro) \in P$.*

Since we have identified the important influence of the embedding element `EE` on the access decisions in SOP-DOM, we may use `EE` to assign a "role" to subjects and objects. So in SOP-DOM, P_S and P_O would be computed locally from the HTML markup and additional security policies, and P would be the global SOP-DOM rules implemented in each browser.

For example, to specify that both external and inline scripts have full cross-origin read and write access rs^{co}_{rw} we may formulate:

$$(s, rs^{co}_{rw}) \in P_S \iff EE(s) = \texttt{<script>} \vee EE(s) = HD. \quad (1)$$

Access to objects is again mainly defined by the embedding element. An image embedded via <**img**> is, for example, inaccessible at all, whereas the same image embedded via <**canvas**> is partially readable. So we could define a role ro^{so}_r with the following equation:

$$(o, ro^{so}_r) \in P_O \iff EE(o) \notin \{ < \texttt{img} >, ... \} \quad (2)$$

Web origins could be taken into account in P by stating that for all values X, $(rs_X^{so}, ro_X^{so}) \in P$ (subject role has same-origin access to object role), $(rs_X^{co}, ro_X^{co}) \in P$ (subject role has cross-origin access to object role), and $(rs_X^{co}, ro_X^{so}) \in P$ (if subject role has cross-origin access to object role, then it also has same-origin access).

This shows that eRBAC seems to be a feasible model, however, the rules to assign roles to subjects and objects could become quite complicated because in addition to the `EE`, we have identified at least two attribute values (*cors* and *sandbox*) which may influence the assignment of such roles. This complexity will be increased if we extend the scope to HTTP security policies such as CSP and pseudo-URIs like `data:`, which are not covered by our current analysis.

7.3 Attribute-Based Access Control

Attribute-Based Access Control (ABAC) [49] is a flexible access control mechanism used in, for example, XACML [50]. It may also be used to implement RBAC: roles can be modeled as *role attributes* assigned to both subject and object. The policy decision in ABAC may depend on other subject, object and environment attributes as well.

Definition 4 *Let $A_i = \{NULL, value_i^1, ..., value_i^{k_i}\}$ be the set of different values of attribute i. Let $\mathscr{SA} = A_1 \times ... \times A_l$, $\mathscr{OA} = A_{l+1} \times ... \times A_m$ and $\mathscr{EA} = A_{m+1} \times ... \times A_n$ be the cartesian products of all subject, object and environment attribute values. Let $\mathscr{R}$ be the set of all access rights. Then an ABAC policy $\mathscr{P}$ is defined as $\mathscr{P} \subseteq \mathscr{SA} \times \mathscr{OA} \times \mathscr{EA} \times \mathscr{R}$.*

Now let $\vec{sa}$ be the array of subject attributes of subject s, $\vec{oa}$ the array of object attributes of object o, and $\vec{ea}$ the actual array of environment attributes. Then subject s has access $r \in \mathscr{R}$ to object o if the array $\vec{a}$, formed by concatenating $\vec{sa}$, $\vec{oa}$, $\vec{ea}$, and r, is contained in $\mathscr{P}$: $\vec{a} \in \mathscr{P}$.

ABAC could be suitable for SOP-DOM because we can model any parameter that influences the access decisions as an attribute. This allows to give a unified treatment to some well-known concepts.

Extended Web Origins. Both subject and object have attributes from which their Web Origin can be computed. In the classical definition of Web Origins in RFC6454 these are `protocol` (`location.protocol`), `domain` (`location.hostname`) and `port` (`location.port`).

- We can, for example, extend this definition to take the legacy `document.domain` declaration into account (see below). We define an additional variable `dd` and assign the value of `document.domain` to it. All these variables are both subject and object variables (cf. Section 7), and are present for both `HD` and `ED` (cf. Table 2).
- The assignment of random Web Origins to sandboxed iFrames can be specified by stating that $origin(o) = \$RAND$ if $sandbox(o) = TRUE$.

Embedding Element. The important role of the embedding element `EE` is modeled as a variable `ee`, applicable to both subject and object, but set only for the embedded document `ED`. The value of `ee` is set to the type of the embedding element. It modifies both same-origin and cross-origin access decisions significantly.

Additional Attributes. Similar to the `ee` attribute, the `cors` and `sandbox` attributes are only defined for the embedded document `ED`. For `cors`, our tests revealed that this attribute modifies access rights to a web object and therefore, it is only an object attribute.

Attributes not fixed by the HTML source code. The ABAC model also defines *environment attributes*, which may not depend on subject or object alone but rather on the execution environment. The only attribute we could qualify to be in $\mathscr{EA}$ during our tests is `sandbox`, since it may be set interactively by using a suitable directive of Content Security Policy.

Extended Web Origin. The ABAC model for SOP-DOM can be presented as the set $\mathscr{P}$ but this does not give any insights into the structure of SOP-DOM. However, four of the seven variables can be combined into a very elegant description of an extended Web Origin. This shows that the ABAC model can also be used to simplify the description of SOP-DOM.

```
1 Read(protocol,domain,port,dd);
2 if dd=NULL or (dd is not a
      superdomain)
3 then wo:=(protocol,domain,port)
4 else wo:=(protocol,dd,NULL)
```

Listing 5: Computation of extended Web Origin.

Listing 5 shows how an extended web origin is computed from the four given ABAC variables. Please note that the `else` branch of this algorithm has been verified by our testbed but different descriptions exist in the literature. In contrast to previous descriptions of the interaction of Web Origins and the `document.domain`

declaration, the novel ABAC based concept of *extended* Web Origin is both simpler ans less error-prone.

7.4 Summary

The requirements on an access control model for SOP-DOM can be formulated as follows: the general rules of SOP-DOM must be expressible without reference to the URL or the HTML context of a web subject or object, and to apply the SOP-DOM rules, URL and HTML context of each web object must be transformed into an abstracted description which then will serve as an input to the general SOP-DOM rules.

This rules out DAC as a model, since DAC rules would simply consist of a large global matrix, where each web object worldwide has a row, and each subject a column.

eRBAC and ABAC both seem promising candidates, since they fit the general requirements. A tentative formalization of the test results presented in this paper in both models could lead to new test cases which could help to decide which of the two approaches, if any, is better suited to formalize SOP-DOM.

8 Conclusions & Future Work

Our analysis highlights the importance to evaluate every single possibility of browser interactions in the SOP-DOM. Different browser data sets can be used to identify inconsistencies across implementations, which can lead to security vulnerabilities. Although edge cases (CORS, sandbox attribute) are mainly responsible for the detected browser behaviors in our evaluation, commonly known cases can also have differences and even vulnerabilities. Consequently, browser vendors have to compare their own implementation with those of other vendors.

Our discussion on access control policies as a model to describe the SOP-DOM helps for a better understanding. Browser implementations can use our insights to describe the SOP-DOM implementation more formally and thus preemptively prevent SOP bypasses. We strongly believe that a more formal SOP-DOM definition will help the scientific as well as the pentesting community to find more severe vulnerabilities. Our test results of the ten tested browsers are available on the testbed website.

Future Work. To extend the coverage, future work may address the following areas: (1.) local storage/session storage or even new data types like Flash or PDF; (2.) different protocols, including pseudo-protocols like `about:` and `data:`; (3.) other elements with URL attributes or properties; (4.) additional HTML attributes.

To generate novel insights into SOP-DOM, the path taken by integrating the `document.domain` declaration could be extended to other attributes like `ee`; for sandboxed iFrames, for example, a random Web Origin should be generated according to the specification. This is however only possible if other `EE`s imposing similar restrictions (e.g., the <**`img`**> element) also use random Web Origins. This remains to be tested.

References

[1] W3C, "Same origin policy," https://www.w3.org/Security/wiki/Same_Origin_Policy, January 2010.

[2] Mozilla, "Same-origin policy," https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, March 2016.

[3] J. Ruderman, "The same origin policy," Online, http://www-archive.mozilla.org/projects/security/components/same-origin.html, 2008.

[4] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. J. Le Hors, G. T. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood, "Document object model (DOM) level 1 specification," World Wide Web Consortium, Recommendation REC-DOM-Level-1-19981001, Oct. 1998.

[5] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *USENIX Security 2014*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 723–735.

[6] E. Z. Yang, D. Stefan, J. C. Mitchell, D. Mazières, P. Marchenko, and B. Karp, "Toward principled browser security," in *HotOS*. USENIX Association, 2013.

[7] 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.

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

[9] A. van Kesteren, "Cross-origin resource sharing," W3C, W3C Recommendation, Jan. 2014, http://www.w3.org/TR/2014/REC-cors-20140116/.

[10] W. Alcorn, C. Frichot, and M. Orrù, *The Browser Hacker's Handbook*. John Wiley & Sons, 2014.

[11] G. S. Kalra, "Exploiting insecure crossdomain.xml to bypass same origin policy (actionscript poc),"

Online, http://gursevkalra.blogspot.de/2013/08/bypassing-same-origin-policy-with-flash.html, August 2013.

[12] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver, "Cookies lack integrity: Real-world implications," in *USENIX Security 2015*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 707–721.

[13] A. Bortz, A. Barth, and A. Czeskis, "Origin cookies: Session integrity for web applications," Online, http://abortz.net/papers/session-integrity.pdf, Web 2.0 Security and Privacy (W2SP), 2011.

[14] C. Masone, K.-H. Baek, and S. Smith, "Wske: Web server key enabled cookies," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, S. Dietrich and R. Dhamija, Eds. Springer Berlin Heidelberg, 2007, vol. 4886, pp. 294–306.

[15] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *ACM CCS 2007*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 58–71.

[16] Y. Oren and A. D. Keromytis, "Attacking the internet using broadcast digital television," *ACM Trans. Inf. Syst. Secur.*, vol. 17, no. 4, pp. 16:1–16:27, Apr. 2015.

[17] M. Smith, "HTML: The markup language (an HTML language reference)," W3C, W3C Note, May 2013, http://www.w3.org/TR/2013/NOTE-html-markup-20130528/.

[18] A. Barth, "The Web Origin Concept," RFC 6454 (Proposed Standard), Internet Engineering Task Force, Dec. 2011.

[19] C. McCormack, J. Watt, D. Schepers, A. Grasso, P. Dengler, J. Ferraiolo, E. Dahlström, D. Jackson, J. Fujisawa, and C. Lilley, "Scalable vector graphics (SVG) 1.1 (second edition)," W3C, W3C Recommendation, Aug. 2011, http://www.w3.org/TR/2011/REC-SVG11-20110816/.

[20] W3C, "Html: The markup language (an html language reference)," https://www.w3.org/TR/2012/WD-html-markup-20121025/elements.html, February 2017.

[21] WHATWG, "The elements of html," https://html.spec.whatwg.org/multipage/semantics.html, February 2017.

[22] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," in *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[23] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: Stealing the pie without touching the sill," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 760–771.

[24] R. Dhamija, J. D. Tygar, and M. Hearst, "Why phishing works," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '06. New York, NY, USA: ACM, 2006, pp. 581–590.

[25] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the android system," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011, pp. 343–352.

[26] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," *Commun. ACM*, vol. 52, no. 6, pp. 83–91, Jun. 2009.

[27] R. Berjon, S. Faulkner, T. Leithead, E. Doyle Navara, E. O'Connor, and S. Pfeiffer, "HTML5 — A vocabulary and associated APIs for HTML and XHTML," World Wide Web Consortium, Recommendation REC-html5-20141028, Oct. 2014.

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

[29] W3C, "Service workers," https://www.w3.org/TR/service-workers/.

[30] Mozilla, "Serviceworker (this is an experimental technology)," https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker.

[31] ——, "Using web workers," https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

[32] C. Jackson and A. Barth, "Beware of finer-grained origins," in *In Web 2.0 Security and Privacy (W2SP 2008)*, 2008. [Online]. Available: http://seclab.stanford.edu/websec/origins/fgo.pdf

[33] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The multi-principal os construction of the gazelle web browser," in

USENIX Security 2009, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 417–432.

[34] S. Chen, D. Ross, and Y.-M. Wang, "An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315248

[35] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from dns rebinding attacks," *ACM Trans. Web*, vol. 3, no. 1, pp. 2:1–2:26, Jan. 2009.

[36] M. Johns, S. Lekies, and B. Stock, "Eradicating dns rebinding with the extended same-origin policy," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 621–636. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534820

[37] J. Jain, "Sop bypassing in safari," Online, http://resources.infosecinstitute.com/bypassing-same-origin-policy-sop-part-2/, Last visited Oct. 2015.

[38] P. Stone, "Next generation clickjacking new attacks against framed web pages," Online, http://www.contextis.com/documents/5/Context-Clickjacking_white_paper.pdf, April 2010.

[39] G. Heyes, "Firefox knows what your friends did last summer," Online, http://www.thespanner.co.uk/2012/10/10/firefox-knows-what-your-friends-did-last-summer/, October 2012.

[40] Ormandy, "Comodo: Comodo "chromodo" browser disables same origin policy, effectively turning off web security." https://code.google.com/p/google-security-research/issues/detail?id=704, Jan. 2016.

[41] N. Poole, "Java applet same-origin policy bypass via http redirect," Online, http://is.gd/MWMaUZ, November 2011.

[42] B. Rios, F. Lanusse, and M. Gentile, "Vulnerability summary for cve-2013-0622," Online, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0622, June 2013.

[43] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song, "Towards a formal foundation of web security," in *CSF*. IEEE Computer Society, 2010, pp. 290–304.

[44] S. Crites, F. Hsu, and H. Chen, "Omash: Enabling secure web mashups via object abstractions," in *ACM CCS 2008*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 99–108.

[45] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting browsers from extension vulnerabilities," in *NDSS 2010*, 2010.

[46] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: Get the security of multiple browsers with just one," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 227–238.

[47] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 921–930.

[48] C. Jackson and H. J. Wang, "Subspace: Secure cross-domain communication for web mashups," in *WWW*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 611–620.

[49] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (abac) definition and considerations," NIST Special Publication 800-162, January 2014.

[50] E. R. (Ed.), "extensible access control markup language (xacml) version 3.0," http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf, January 2013.