JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript

Steven Van Acker(✉) and Andrei Sabelfeld

Chalmers University of Technology, Gothenburg, Sweden
acker@chalmers.se

Abstract. Today's web applications rely on the same-origin policy, the primary security policy of the Web, to isolate their web origin from malicious client-side JavaScript.

When an attacker can somehow breach the same-origin policy and execute JavaScript code inside a web application's origin, he gains full control over all available functionality and data in that web origin.

In the JavaScript sandboxing field, we assume that an attacker has the ability to execute JavaScript code in a web application's origin. The goal of JavaScript sandboxing is to isolate the execution of certain JavaScript code and restrict what functionality and data is available to it.

In this paper we discuss proposed JavaScript sandboxing systems divided into three categories: JavaScript sandboxing through JavaScript subsets and rewriting systems, JavaScript sandboxing using browser modifications and JavaScript sandboxing without browser modifications.

1 Introduction

The Web today is unthinkable without JavaScript. Studies [96] show that close to 90 % of the top 10 million websites of the Web use JavaScript.

JavaScript can turn the Web into a lively, dynamic and interactive end-user experience. For this purpose, today's browsers have an arsenal of powerful JavaScript functionality at their disposal which all becomes available to Web applications running JavaScript. Examples of this powerful functionality include access to audio and video recording devices, real-time communication (RTC) channels than can pierce firewalls, the ability to store data on the client-side, 3D graphics rendering facilities and more.

Giving all this power to unfamiliar web applications is not necessarily a good idea. With great power comes great responsibility, a trait not commonly found in web applications because they often include third-party JavaScript from untrusted sources [70]. In the wrong hands, this powerful JavaScript functionality can be abused to e.g. access and steal sensitive information.

A typical scenario illustrating third-party JavaScript inclusion can be found in online advertising. A recent security-related event in this setting equally illustrates the threat associated with third-party JavaScript inclusions. In July 2015, the website of renowned security expert Troy Hunt experienced [89] a Cross-Site Scripting attack launched through a script used for online advertising.

A. Aldini et al. (Eds.): FOSAD VIII, LNCS 9808, pp. 32–86, 2016.
DOI: 10.1007/978-3-319-43005-8_2

The attack was obvious and visible because the attacker seemingly set out to create a proof-of-concept to display a JavaScript prompt window. However, this attack could have caused a lot more damage while at the same time remain invisible if the attacker has chosen to do so instead.

This scenario is a good case for restricting JavaScript functionality, otherwise known as JavaScript sandboxing. Had the advertisement run in a JavaScript sandbox with restricted functionality, then a successful attack would not be able to abuse the full power of a browser's JavaScript APIs.

In this paper, we discuss the current state-of-the-art research in JavaScript sandboxing on the client-side, and in the browser in particular. JavaScript can be used elsewhere on the client-side, for instance as an embedded scripting in browser extensions [29,66], OpenOffice [9], MongoDB [63], etc. JavaScript can also be used on the server-side, e.g. Node.JS [4], and there are even microcontrollers that understand JavaScript [21,87]. We consider these use cases out of scope and only focus on JavaScript as used in web pages visited by web browsers.

Based on the typical web scenario and attacker model, we divide the JavaScript sandboxing literature in three categories: JavaScript sandboxing through JavaScript subsets and rewriting systems, JavaScript sandboxing using browser modifications and JavaScript sandboxing without browser modifications.

The remainder of this paper is organized as follows. Section 2 draws the context and introduces background material. Section 3 discusses JavaScript sandboxing systems involving JavaScript subsets and rewriting systems. Section 4 discusses browser modifications to achieve JavaScript sandboxing. Section 5 discusses JavaScript sandboxing systems which do not require any browser modifications. Section 6 highlights two well-known JavaScript sandboxing systems and details their usage in the real world. Section 7 concludes this work with a brief discussion of the advantages and disadvantages of the three categories of JavaScript sandboxing systems.

2 Background – Setting the Context

In this section we set the context for the remainder of this paper.

First, we look at a reference browser architecture, the JavaScript language and the different JavaScript APIs available to web developers in Sects. 2.1 to 2.3. We note that a browser is composed of several reusable subsystems such as the JavaScript engine and that the JavaScript engine is disconnected from the rest of the browser, forming a good interception point for enforcing security policies.

Next, we take a brief look at web applications as a combination of web technologies in Sect. 2.4, followed by the same origin policy in Sect. 2.5, a cornerstone of web security, which makes sure that web applications remain separated from each other inside the browser.

A typical web scenario with its actors and interactions, together with the attacker model we use in this paper, is described in Sect. 2.6.

Third party script inclusion is an integral part of web applications today, at the cost of having to trust the third party. This trust is not always deserved,

leading to security problems. The threat posed by third party script inclusions is motivated in Sect. 2.7.

Finally, in Sect. 2.8, we describe the concept of JavaScript sandboxing as a means to restrict available functionality inside the JavaScript environment, and three main ways in which this can be accomplished.

2.1 Browser Architecture

Simply put, a web browser is a computer program used to retrieve content from the Web, interact with it and display it on a screen, either directly or through helper applications. More concretely, a web browser is a complex piece of software comprised of multiple subcomponents, each with its own task, that work together to allow a user to visit the Web.

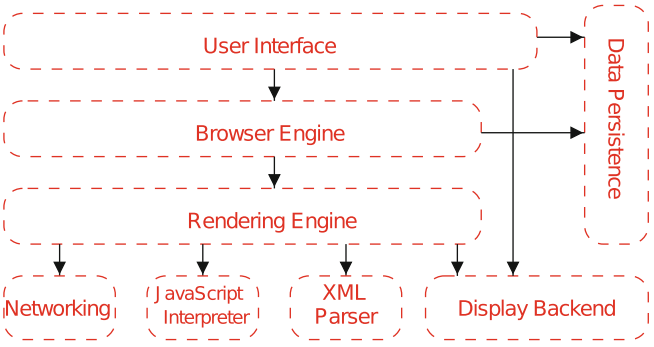


Fig. 1. The eight subsystems of the reference architecture of a web browser, from [31].

The reference architecture of a web browser consists of eight interconnected subsystems [31], shown in Fig. 1:

User Interface. The part of the browser that interacts directly with the user, displaying windows and toolbars.

Browser Engine. Handles *Uniform Resource Identifier* (URI a more generic form of URL) loading, and implements browser actions such as the forward and backward button behavior. The browser engine provides a high-level interface to the rendering engine.

Rendering Engine. The subsystem responsible for displaying content on the screen. It can display HTML and XML, styled with *Cascading Style Sheets* (CSS) and embedding images. It also includes the HTML parser, turning HTML content into the *Document Object Model* (DOM), a structured form more suitable for other components. For the sake of compatibility with older browsers, many HTML parsers also have a *quirks mode* [5] next to a *standards mode*. In standards mode, the HTML parser strictly complies to W3C and IETF standards and rejects any malformed HTML. In quirks mode however,

the HTML parser is more lenient and quietly repairs broken HTML instead of rejecting it.

Networking Subsystem. The part of the browser responsible for communicating with the network over protocols such as HTTP, loading content from other web servers, caching data and converting data between different character sets.

JavaScript Interpreter. Also known as the JavaScript engine, this subsystem parses and executes JavaScript code. JavaScript itself is an object-oriented programming language that can evaluate expressions, but does not define ways to influence the rest of the world. To interact with the outside, such as the other browser components, the user or the network, the JavaScript engine must communicate with other subsystems.

XML Parser. Parses XML documents into a DOM structure. This component is different from the HTML parser and is a generic, reusable component. The HTML parser on the other hand, is optimized for performance and tightly coupled with the rendering engine.

Display Backend. This component provides an interface to the underlying operating system to draw windowing primitives and fonts.

Data Persistence. Stores and retrieves data such as browsing history, bookmarks, cookies and browser settings.

The modular subsystems are often reused between different browser vendors. For instance, the Gecko [64] browser engine is used by Mozilla Firefox, Netscape Navigator, Galeon [1] and others. Google Chrome uses the Blink [11] browser engine, also used by Opera [71] and the Android browser [97]. Microsoft Internet Explorer uses the Trident [54] layout engine, also used by the Maxthon [49] browser. Browser components are not only reused by web browsers. Mozilla Firefox's JavaScript engine, SpiderMonkey [80], is also used in the GNOME3 desktop environment [27], and can be used as a standalone JavaScript interpreter. Google Chrome's JavaScript engine, V8 [28], also powers node.js [4], a server-side JavaScript runtime environment.

Many of these subsystems are used by the browser during routine operations such as loading and rendering a webpage. When a user points a browser to a webpage and the browser has downloaded an HTML document, the *rendering pipeline* is started that will eventually display the webpage and allow the user to interact with it.

The rendering pipeline generally consists of 3 steps: parsing, layouting and rendering:

- During the parsing step, the downloaded HTML document is parsed into a data structure known as the *Document Object Model* (DOM) tree. Each node in this tree comprises an HTML element, with links to the parent element and sub-elements.
- In the layouting step, rectangular representations of the nodes in the DOM are arranged according to the styling rules dictated by the webpages and its *Cascading Style Sheets* (CSS) information.

– Finally, in the rendering step, a graphical representation of each HTML element in the DOM is painted in its respective rectangular representation, and finally drawn onto the user's screen.

This rendering pipeline is a gradual process that is re-iterated while a browser loads all the needed resources.

2.2 JavaScript

In 1995, Netscape management told Brendan Eich to create a programming language to run in the web browser that "looked like Java." He created JavaScript in only 10 days [15]. In addition to browser plugins, JavaScript was another novel feature of Netscape Navigator 2.0 that supported Netscape's vision of the Web as a distributed operating system. In contrast with Java, which was considered a heavyweight object-oriented language and used to create Java applets, JavaScript would be Java's "silly little brother" [3], aimed towards non-professional programmers who would not need to learn and compile Java applets.

```
1 var name = prompt("What is your name?");
2 var year = prompt("What year were you born?");

3 var today = new Date();
4 var age = today.getFullYear() - year;

5 alert("Hello "+name+", you are about "+age+" years young");
```

Listing 1.1: Example JavaScript code prompting the user for name and birthyear, calculating age and displaying it in a pop-up.

Listing 1.1 shows a simple example of JavaScript. When executed, the code will prompt for the user's name and birth-year. It will then calculate the user's age based on the current year and display it with a greeting using a pop-up. This JavaScript example makes use of the `prompt()` function, the `Date` object and the `alert()` function.

When an HTML document is about to be loaded, and before the rendering pipeline starts, the browser initializes an instance of the JavaScript engine and ties it uniquely to the webpage about to be loaded.

The webpage's developer can use JavaScript to interact with this rendering pipeline by including JavaScript in several ways. JavaScript can be executed while the pages is loading, using HTML `<script>` tags. These script tags can cause the browser to load external JavaScript and execute them inside the webpage's JavaScript execution environment. Script tags can also contain inline JavaScript, which will equally be loaded and executed. HTML provides a way to register JavaScript event handlers with HTML elements, which will be called when e.g. an image has loaded, or the user hovers the mousepointer over a hyperlink. In addition, JavaScript can register these event handlers itself by querying and manipulating the DOM tree. Events are not only driven by the user, but can also be driven programmatically. For instance, JavaScript has the ability

to use a built-in timer to execute a piece of JavaScript at a certain point in the future. Likewise, the `XMLHttpRequest` functionality available in the JavaScript engine allows a web developer to retrieve Internet resources in the background, and execute a specified piece of JavaScript code when they are loaded. Lastly, JavaScript has the ability to execute dynamically generated code through the `eval()` function.

2.3 JavaScript APIs

JavaScript's capabilities inside a web page are limited to the APIs that are offered to it. Typical functionality available to JavaScript in a web page includes manipulating the DOM, navigating the browser and accessing resources on remote servers.

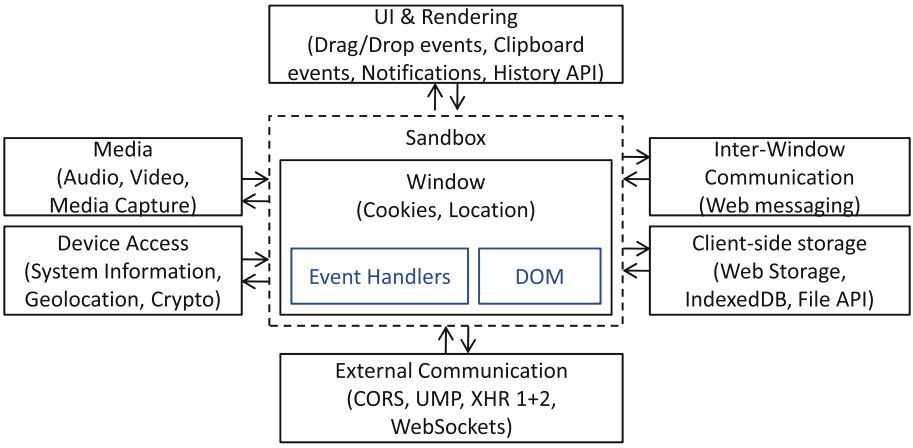


Fig. 2. Synthesized model of the emerging HTML5 APIs, from [91].

In the new HTML 5 and ECMAScript 5 specifications, JavaScript gains access to more and powerful APIs. Figure 2 [17] shows a model of some of these new HTML 5 APIs, which are further explained below.

Inter-frame communication. Facilitates communication between windows (e.g. between mashup components). This includes window navigation, as well as Web Messaging (`postMessage`).

Client-side storage. Enables applications to temporarily or persistently store data. This can be achieved via Web Storage, IndexedDB or the File API.

External communication. Features such as CORS, UMP, XMLHttpRequest level 1 and 2, WebSockets, raw sockets and Web RTC (real-time communication) allow an application to communicate with remote websites.

Device access. Allows the web application to retrieve contextual data (e.g. geolocation) as well as system information such as battery level, CPU information, ambient sensors and high-resolution timers.

Media. Enable a web application to play audio and video fragments, capture audio and video via a microphone or webcam and manage telephone calls through the Web Telephony API.

The UI and rendering. Allow subscription to clipboard and drag-and-drop events, issuing desktop notifications, allow an application to go fullscreen, populating the history via the History API and create new widgets with Web Components API and Shadow DOM.

2.4 Web Applications

A web application combines HTML code, JavaScript and other resources from several web servers, into a functional application that runs in the browser. Unlike typical desktop applications which need to be installed on a computer's hard disk, web applications are accessible through the web browser from anywhere and do not need to be installed.

A key component in today's web application, is JavaScript. JavaScript code in a web application executes in the browser and can communicate with a web server, which typically also executes code for the web application.

Consider a website wishing to display the latest tweets from a Twitter feed. Such a widget can be embedded into a web page, as shown in Fig. 3. Without a client-side programming language such as JavaScript, the web server from which this web page is retrieved, could gather and insert the latest tweets at the moment the web page was requested, and insert them into the web page as HTML-formatted text. When rendered, the visitor would see the latest tweets, but they would not update themselves in the following minutes because the web page is static.

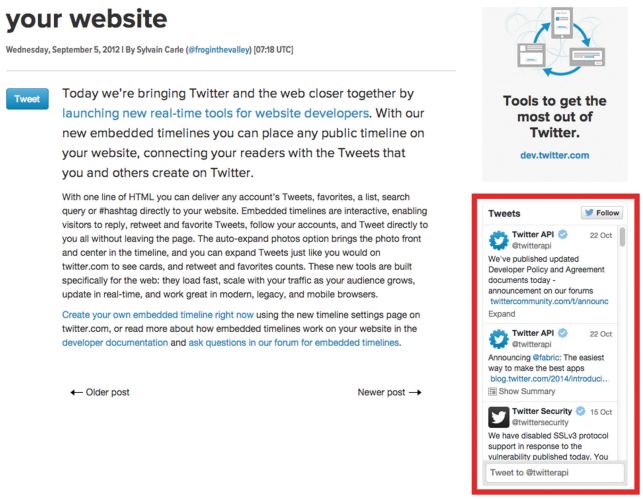


Fig. 3. Example of an embedded live Twitter feed (indicated by the rectangle on the bottom right), from [90].

Another option is to use JavaScript on the client-side. When the web page is requested, the web server can insert JavaScript that regularly requests the latest tweets from the feed and updates the web page to display them. The result is an active web page that always displays the latest information.

This example consists of only one HTML page and requests information from one source. Today's web has many web applications combining a multitude of third-party resources. Examples are Facebook, YouTube, Google Maps and more.

2.5 The Same-Origin Policy

If web applications were allowed complete access to a browser, they would be able to interfere in the operation of other web applications running in the same browser. Given the powerful APIs briefly discussed in the previous section, a web application would be able to access another web application's DOM, local storage and data stored on remote servers.

To prevent this, web applications are executed in their own little universe inside the web browser, without knowledge of each other. The boundaries between these universes are drawn based on the *Same-Origin Policy* (SOP) [92].

When the root HTML document of web application is loaded from a certain URL, the *origin* of that web application is said to be a combination of the scheme, hostname and port-number of that URL. For instance, a web application loaded from https://www.example.com has scheme *https*, hostname *www.example.com* and, in this case implicit, port number *443*. The origin for this web application is thus *(https,www.example.com,443)* or https://www.example.com:443.

The *Same-Origin Policy* (SOP) dictates that any code executing inside this origin only has access to resources from that *same* origin, unless explicitly allowed otherwise by e.g. a Cross-Origin Resource Sharing (CORS) [94] policy. In the previous example, the web application from https://www.example.com:443 cannot retrieve the address book from a webmail application with different origin https://mail.example.com:443 running in the same browser, unless the latter explicitly allows it.

The same-origin policy is part of the foundation of web security and is implemented in every modern browser. In this text we only consider the restrictions imposed by the SOP on the execution of JavaScript.

Insecurely written web applications may allow attackers to breach the same-origin policy by executing their JavaScript code in that web application's origin. Once arbitrary JavaScript code can be injected into a web application, it can take over control and access all available resources in that web application's origin.

Consider a typical webmail application, such as Gmail, allowing an authenticated user to access his emails and contact list. The webmail application offers a user interface in the browser and can send requests to the webmail server to send and retrieve emails, and manipulate the contact list.

An attacker may manage to lure an authenticated user of this webmail application onto a specially crafted website. This website could try to contact the

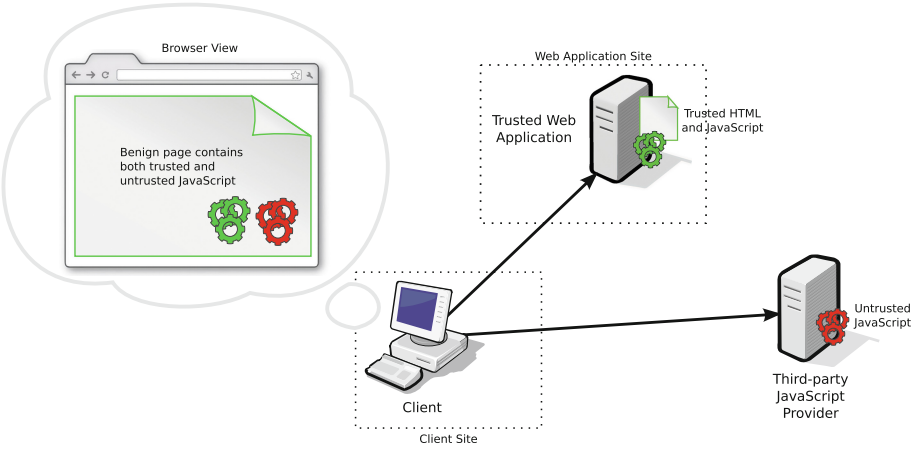


Fig. 4. A typical web application with third-party JavaScript inclusion. The web application running in the browser combines HTML and JavaScript from a trusted source, with JavaScript from an untrusted source.

webmail server to send and retrieve emails and contact information, just as the web application would. However, the webmail application's origin is e.g. https://webmail.com:443, while the attacker's website is https://attacker.com:443. Because of the SOP, JavaScript running on the attacker's website has no access to resources of the webmail's origin.

Now consider what would happen if the webmail application is written insecurely, so that an attacker can execute JavaScript in its origin: https://webmail.com:443. Because the attacker's code runs inside the same origin as the webmail application, it has access to the same resources and can also read and retrieve emails and contact information. Because of the power of JavaScript, an attacker can do much more. Specially crafted JavaScript can compose spam email messages and send them out using the victim's email account, or it could erase the contact list. It could even download all emails in the mailbox and upload them to another server.

An attacker with the ability to execute JavaScript in a web application's origin can take full control of that web application. In the typical web application scenario, untrusted JavaScript can be executed in two ways: by including it legitimately from a third party, or by having it injected through a Cross-Site Scripting vulnerability in the web application or an installed browser plugin or extension.

2.6 The Typical Web Scenario and Attacker Model

When discussing Web security, it is important to keep in mind a typical web application with third-party JavaScript and the actors involved in it. Figure 4 shows such a typical web application where HTML and JavaScript from a trusted

source are combined with JavaScript from an untrusted source. Remember that all JavaScript, trusted or untrusted, running in a web application's origin has access to all available resources.

There are three actors involved in this scenario: The developer of the trusted web application and the server it is hosted on, the developer of the third-party JavaScript and the server it is hosted on, and the client's browser.

Both the client and the trusted web application have a clear motive to keep untrusted JavaScript from accessing the web application's resources. The client will wish to protect his own account and data. The trusted web application has its reputation to consider and will protect a user's account and data as well. Furthermore, the client does not need to steal information from himself and can use any of his browser's functionality without needing to use a remote web application. Likewise, the web application developer owns the origin in which the web application runs. Stealing data from his own users through JavaScript is not necessary.

It may be the case that the client has modified his browser and installed a browser plugin or extension. Such a plugin or extension may be designed to make the interaction with the web application easier or automated, potentially circumventing certain defensive measures put in place by the developer of the web application. In this scenario, the client is still motivated to protect his account and data, but may be exposing himself to additional threats through the installed browser plugins or extensions that form additional attack surface.

The third-party script provider however, does not necessarily share the same desire to protect a user's data. Even with the best of intentions, a third-party script provider may be compromised and serving malicious JavaScript without its knowledge. It may be the case that the script provider has an intrusion-detection system in place that will detect when it is serving malware, but this would be wishful thinking. In the worst case, the third-party script provider is acting maliciously on its own for whatever sinister reason. In any case, the client and trusted web application cannot trust a third-party script provider with their secrets.

The attacker model best associated with this actor is the *gadget attacker* [10]. A gadget attacker is a malicious actor who owns one or more machines on the Internet, but can neither passively not actively intercept network traffic between the client's browser and the trusted web application. Instead, the gadget attacker has the ability to have the trusted web application's developer integrate a gadget chosen by the attacker.

2.7 Third-Party Script Inclusion

Web applications are built from several components that are often included from third-party content providers. JavaScript libraries like jQuery or the Google Maps API are often directly loaded into a web application's JavaScript environment from third-party script providers.

In a large-scale study of the Web in 2012 [70], Nikiforakis et al. found that 88.45 % of the top 10,000 web sites on the Web, include JavaScript from a

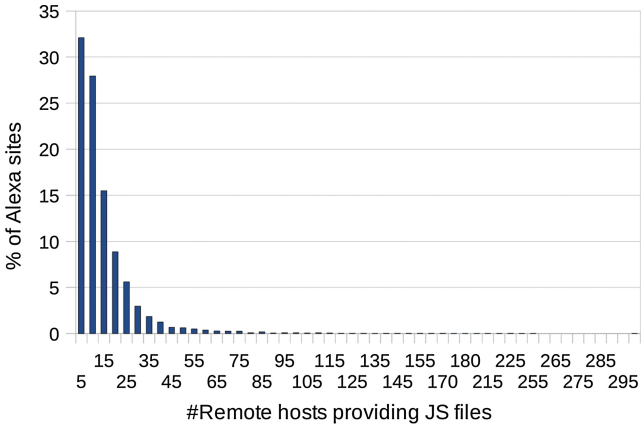


Fig. 5. Relative frequency distribution of the percentage of top Alexa websites and the number of unique remote hosts from which they request JavaScript code, from [70].

third-party script provider. Figure 5 shows the distribution of the number of third-party script providers each web site includes. While about a third include JavaScript from at most 5 remote hosts, there are also web sites that include JavaScript from more than 295 different remote hosts.

Including JavaScript from remote hosts implicitly trusts these hosts not to serve malicious JavaScript. If these third-party script providers are untrustworthy, or if they have been compromised, a web application may end up executing untrusted JavaScript code.

As an example, consider jQuery, a popular multi-purpose JavaScript library used on 60 % of the top million websites on the Web [12]. The host distributing jQuery was compromised in September 2014 [40], giving the attackers the ability to modify the library and possibly infect many websites that include the library directly from http://jquery.com. Fortunately, the attackers did not modify the jQuery library itself, but used the compromised server to spread malware instead. Although the JavaScript library itself was not tampered with, the jQuery compromise indicates the inherent security threat that third-party script inclusions can pose.

2.8 JavaScript Sandbox

The gadget attacker, as defined in Sect. 2.6, has the ability to integrate a malicious gadget into a trusted web application. This allows the attacker to execute any chosen JavaScript code in the JavaScript execution environment of this trusted web application's origin and access its sensitive resources.

Given this attacker model, we cannot stop the attacker from presenting a web application user's browser with chosen JavaScript. In this paper we are not concerned with cross-site scripting or other injection attacks and assume that the

attacker already has the ability to execute JavaScript in the JavaScript environment, no matter through which means this was accomplished. In this scenario, it would be helpful to have a mechanism to restrict the available functionality inside the JavaScript environment, according to the least-privilege principle. The impact of executing (potentially malicious) JavaScript in such an environment would then be limited to the available functionality. Such an environment, in which we can isolate JavaScript and restrict its access to certain resources and functionality, is called a JavaScript sandbox.

From the typical web scenario architecture from Sect. 2.6, keeping in mind our attacker model, there are only two possible locations that can be considered to deploy a JavaScript sandboxing mechanism: the trusted web application and the client's browser. The third-party script provider is considered untrustworthy.

The developer of the web application and the server hosting it, are trusted according to the attacker model. This server then offers a possible location to facilitate JavaScript sandboxing. Before serving the untrusted JavaScript from the third-party script provider to the client, the code can be reviewed and optionally rewritten to make sure it does not abuse the web application's available resources.

The client's browser provides a second location to sandbox JavaScript, because it is also considered trusted. With direct access to the JavaScript execution context, a JavaScript sandboxing system located at the client-side has better means to restrict access to resources and functionality.

JavaScript sandboxing can be achieved by restricting the used JavaScript language to a subset that can then be verified to be safe, or even rewrite the JavaScript code into a version which is safe. Such a solution involves a JavaScript subset and a rewriting mechanism which will be discussed in Sect. 3.

Without restricting or rewriting JavaScript code, JavaScript sandboxing can be achieved by modifying the environment in which JavaScript executes. Such a JavaScript sandboxing mechanism can be implemented by modifying the JavaScript engine in the browser and build in machinery to enforce a certain policy. This type of JavaScript sandboxing which uses a browser modification will be discussed in Sect. 4.

Finally, it is also possible to sandbox JavaScript without any language-level restrictions, rewriting or browser modifications, by repurposing JavaScript functionality to isolate and restrict JavaScript. Such JavaScript sandboxing systems which do not require browser modifications, are discussed in Sect. 5.

2.9 Conclusion

This section introduced important Web technologies required to understand the remainder of this text.

Web browsers are used to browse the Web and consist of many different cooperating subsystems, such as the HTML parser and the JavaScript engine. Web applications are a combination of HTML pages, JavaScript code and other resources retrieved from multiple sources running in the browser. Each web application is isolated and protected in its own origin by the same-origin policy.

Web applications often include JavaScript code from third-party script providers, placing often undeserved trust on third parties, who can then execute unrestricted JavaScript code in the web application's origin.

JavaScript sandboxing can limit the functionality available in a JavaScript environment and we consider three categories which we will discuss in the next sections: JavaScript subsets and rewriting systems in Sect. 3, JavaScript sandboxing using browser modifications in Sect. 4 and JavaScript sandboxing without browser modifications in Sect. 5.

3 JavaScript Subsets and Rewriting

JavaScript is a very flexible and expressive programming language which gives web-developers a powerful tool to build web-applications. However, this same powerful tool is also available to attackers wishing to execute malicious JavaScript code in a website visitor's browser.

Moreover, the powerful nature of JavaScript is problematic because it hinders code verification efforts which could prove safety properties for a given piece of JavaScript code.

Example: `eval()`. Consider for instance the JavaScript fragment in Listing 1.2. When executed in a browser, this code will prompt a user to input a line of text. The one-way hashing algorithm MD5 is then used to compute a hash of this line of text. If the hash matches `"3b022ec21226e862450f2155ef836827"`, the MD5 hash for `"alert('hello')"`, then the line of text is passed to the `eval()` function and executed as JavaScript code.

```
1  var cmd = prompt();

2  // MD5 algorithm computes a one-way hash
3  function md5(m) {
4      // ...
5      return m;
6  }

7  // verifies whether the given input is "alert('hello')"
8  if(md5(cmd) == "3b022ec21226e862450f2155ef836827") {
9      eval(cmd);
10 }
```

Listing 1.2: Example JavaScript calling `eval()` on user input, but only if its MD5 hash matches a given hash.

Given that the MD5 hashing algorithm cannot easily be reversed, it is practically impossible for a code verification tool to automatically determine the effect of this code, prior to its execution. The `eval()` function illustrates a feature of JavaScript which makes code verification difficult because of its dynamic nature. For this reason, `eval()` is considered evil [77] and should be used with the greatest care, or not be used at all.

Example: Strange Semantics and Scoping Rules. As another example, the JavaScript fragment in Listing 1.3 illustrates some strange semantic rules in JavaScript, including the `with` construct. This particular example showcases some non-intuitive scoping rules associated with the scope chain. The scope chain consists of an ordered list of JavaScript objects which are consulted when unqualified names are looked up at runtime.

```
1  var o = {f:2, x:4};

2  console.log("before with: f == " + f);
3  console.log("before with: x == " + x);
4  console.log("before with: \"x\" in window == " + ("x" in window));

5  with(o) {
6      function f() { }
7      console.log("inside with: f == " + f);
8      var x = 3;
9      console.log("inside with: x == " + x);
10 }

11 console.log("after with: o.f == " + o.f);
12 console.log("after with: o.x == " + o.x);
13 console.log("after with: f == " + f);
14 console.log("after with: x == " + x);
15 console.log("after with: \"x\" in window == " + ("x" in window));
```

Listing 1.3: Example JavaScript using the `with` construct to place a new object at the front of the scope chain during the evaluation of the construct's body. This example is adapted from Miller et al. [60].

Before continuing, the reader is advised to read the code and try to predict what it will output. The actual output of the code in this example, is listed in Listing 1.4.

```
1  before with: f == function f() { }
2  before with: x == undefined
3  before with: "x" in window == true
4  inside with: f == 2
5  inside with: x == 3
6  after with: o.f == 2
7  after with: o.x == 3
8  after with: f == function f() { }
9  after with: x == undefined
10 after with: "x" in window == true
```

Listing 1.4: Output of example in Listing 1.3.

From the output, it appears that both `f` and `x` are already defined before they are even declared, but `x` has `undefined` as value. Using `with`, the user-defined object `o` is pushed to the front of the scope chain. The new function `f()` is declared, but the subsequent `console.log()` call seemingly is not aware it. Instead, the value of `f` is retrieved from the first object in the scope chain (`o`), resulting in `2`. Then, a variable `var x` is declared and assigned `3`. The following `console.log()` call is aware of this declaration and outputs the correct value.

Outside the with loop, the object `o` has changed to reflect the new value of `o.x`, but did not record any change to `o.f`.

The strange behavior in this example indicates that variable and function declarations have different semantics in JavaScript. The discrepancy between variable and function declarations can be explained by a process called "variable hoisting." Variable hoisting examines the JavaScript code to be executed and performs all declarations before any code is actually run.

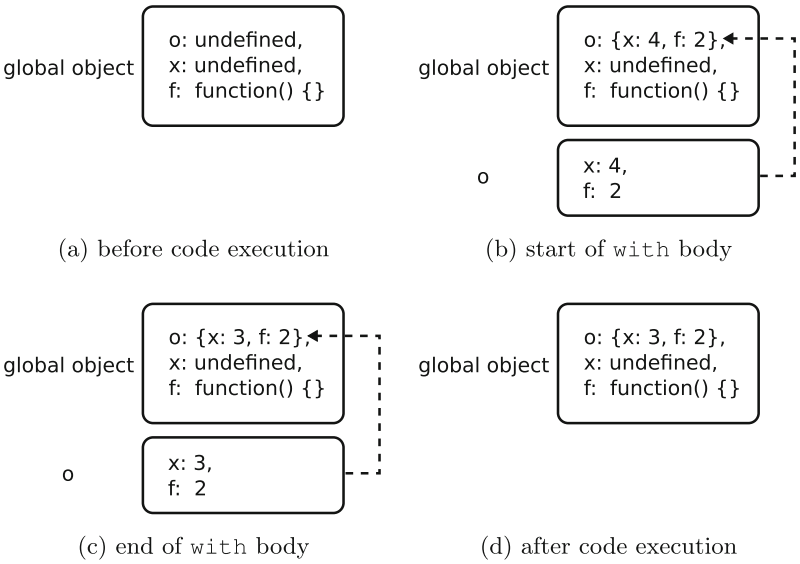


Fig. 6. The scope chain during execution of the example in Listing 1.3. In this depiction, the scope chain grows down so that newly pushed objects are at the bottom.

A graphical representation of the scope chain during the execution of this example is shown in Fig. 6 and can be used as a visual aid during the explanation.

Depicted in Fig. 6a is the result of the variable hoisting before any code is run. The function `f()` and the variable `x` are declared on the global object. While the variable `x` has value `undefined`, the function `f()` is declared and is assigned its value immediately.

Next, the object `o` is pushed to the front of the scope chain. The scope chain right after this push and right before the start of the `with` construct, is shown in Fig. 6b. Any unqualified names are now looked up in the variable `o`.

The third image shown in Fig. 6c, depicts the state of the scope chain at the end of the `with` body. Here, the value of the property `x` of the object `o` has changed to `3` because of the assignment. Also note that the value of `f` has not changed because variable hoisting declares and initializes a function in a single step before the code is run, and so outside of the `with` body.

Finally, in Fig. 6d, the scope chain is restored because the `with` body ended.

The strange scoping rules and semantics of `with` are difficult to reason about for uninitiated programmers. Widely-acknowledged as being a "JavaScript wart" [33], it is often recommended to not use the `with` construct because it may lead to confusing bugs and compatibility issues [68].

JavaScript Subsets: Verification and Rewriting. The goal of JavaScript code verification and rewriting is to inspect JavaScript code before it is executed in a browser, and ensure that it is not harmful.

In the light of the previous examples, it can be desirable to eliminate those constructs from the JavaScript language that hinder code verification efforts or cause confusion in general. At the same time, it is also desirable to maintain as much of the language as possible so that JavaScript is still useful. Such a reduced version of JavaScript, with e.g. `eval()` and `with` construct missing, is called a JavaScript subset.

The usage of a JavaScript subset must be accompanied by a mechanism which verifies that a given piece of code adheres to the subset. A deviation from the subset's specification can be handled in two ways: rejection and rewriting.

Rejection is the simpler of both options, treating a deviation from the subset as a hard error and refusing to execute the given piece of code.

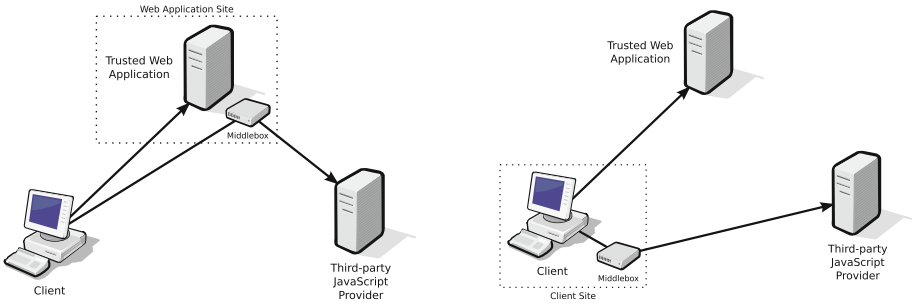
Rewriting is a softer alternative, transforming the deviating piece of code into code which conforms to the subset. Such a rewriting phase can also introduce extra instrumentation in the code to ensure that the code behaves in a safe way at runtime.

Interception in a Middlebox. Both the JavaScript subset verification and rewriting steps necessitate the processing of raw third-party JavaScript code before it reaches the client's browser. These steps are to be performed in a *middlebox*, a network device that sits on the network path between a client and a server. Such a middlebox may consist of a physical device unrelated to either client or server, but it may just as well be collocated with either client or server.

From the attacker model discussed in Sect. 2.6, we can eliminate the third-party script provider's site as a possible location to verify and rewrite JavaScript. We are left with two possible locations for these tasks: the site of the trusted web application and the client's site.

A middlebox at the site of the web application, as shown in Fig. 7a, can equally be implemented as part of a separate network device such as a load-balancer, reverse proxy or firewall, or can be integrated to be part of the web-application.

A middlebox at the client's site, as shown in Fig. 7b, can either be a implemented as a proxy performing the required verification and translation steps, or as a browser plugin or extension, implementing the proxy's behavior as part of the browser.



(a) Middlebox at web application site (b) Middlebox at client site

Fig. 7. Architectural overview of a setup where a middlebox is used for code verification and transformation, at the web application site and at the client site.

ECMAScript 5 Strict Mode. ECMAScript 5 strict mode [65], or JavaScript strict, is a standardized subset of JavaScript with intentionally different semantics than normal JavaScript.

```
1  "use strict";

2  var example = 123;
3  // the following fails because the name is misspelled
4  exmaple = 345;

5  // the following fails because of a duplicate key name
6  var obj = {p:1, p:2};

7  // the following fails because "with" is not allowed
8  with(obj) {
9      alert(p);
10 }
```

Listing 1.5: JavaScript strict mode example.

To use strict mode, a JavaScript developer must only place `"use strict";` at the top of a script of function body, as shown in Listing 1.5. Strict mode will then be enforced for that entire script, or only in the scope of that function. JavaScript strict mode can be mixed with and function together with normal JavaScript.

Strict mode removes silent failures and turns them into hard errors that throw exceptions and halt JavaScript execution. For instance, accidentally creating a global variable by mistyping a variable name, will throw an error. Likewise, overwriting a non-writable global variable like `NaN` or defining an object with a duplicate key, causes strict mode to throw errors.

Strict mode simplifies variable names and allows better JavaScript engine optimization by removing the `with` construct. Through this construct, JavaScript

engine optimizations may be confused about the actual memory location of a variable. In addition, strict mode changes the semantics of `eval()` so that it can no longer create variable in the surrounding scope.

Strict mode also introduces some fixes with regard to security. It is no longer possible to access the global object through the `this` keyword, preventing unforeseen runtime leaks. It is also no longer possible to abuse certain variables to walk the stack or access the `caller` from within a function.

Finally, strict mode forbids the use of some keywords that will be used in future ECMAScript versions, such as `private`, `public`, `protected`, `interface`, ...

Research in the area of JavaScript subsets and rewriting systems includes BrowserShield [76], CoreScript [99], ADsafe [16], Facebook JavaScript [88], Caja [60], Jacaranda [36], Microsoft Live Websandbox [58], Jigsaw [53], Gatekeeper [32], Blancura [25], Dojo Secure [42], ... The remainder of this section discusses a selection of work on JavaScript subsets and rewriting systems.

3.1 BrowserShield

Reis et al. have developed BrowserShield, a dynamic instrumentation system for JavaScript. BrowserShield parses and rewrites HTML and JavaScript in a middlebox, rewriting all function calls, property accesses, constructors and control structures to be relayed through specialized methods of the `bshield` object. A client-side JavaScript library then inserts this `bshield` object, which mediates access to DOM methods and properties according to a policy, into the JavaScript execution environment before any scripts run.

BrowserShield aims at preventing the exploitation of browser vulnerabilities, such as MS04-40 [56], a buffer overflow in the Microsoft Internet Explorer browser caused by overly long `src` and `name` attributes in certain HTML elements. To shield the browser from attacks against these vulnerabilities, BrowserShield rewrites both HTML and JavaScript, transforming them to filter out any detected attacks. BrowserShield does not use a JavaScript subset, because it needs to be able to rewrite any HTML and JavaScript found on the Internet to be effective.

```
1 // original JavaScript code
2 eval("...");

3 // rewritten by BrowserShield
4 bshield.invokeFunc(eval, "...");
```

Listing 1.6: Example JavaScript code rewritten by BrowserShield.

Although sandboxing is not the main goal of BrowserShield, its rewriting mechanism provides all the necessary machinery to accomplish this goal by tuning the policy. For instance, BrowserShield could have a policy in place to mediate access to the sensitive `eval()` function. Listing 1.6 shows the output of BrowserShield's rewriting mechanism on a JavaScript example using the

`eval()` function. After the rewriting step, any call to `eval()` in the original code is relayed through the `bshield` object, which can mediate access at runtime.

A prototype of BrowserShield was implemented as a Microsoft ISA Server 2004 [55] plugin for evaluation. The plugin in this server-side middlebox is responsible for rewriting HTML and script elements, and injecting the BrowserShield client-side JavaScript library which implements the `bshield` object and redirects all JavaScript functionality through it. BrowserShield worked as expected during evaluation. The performance evaluation indicated a maximum slowdown of 136x on micro-benchmarks, and on average 2.7x slowdown on rendering a webpage.

3.2 ADsafe

The ADsafe subset, developed by Douglas Crockford, is a JavaScript subset designed to allow direct placement of advertisements on webpages in a safe way, while enforcing good coding practices. It removes a number of unsafe JavaScript features and does not allow uncontrolled access to unsafe browser components.

Examples of the removed unsafe JavaScript features are: the use of global variables, the use of `this`, `eval()`, `with`, using dangerous object properties like `caller` and `prototype`. ADsafe also does not allow the use of the subscript operator, except when it can be verified that the subscript is numerical, e.g. `a[i]` is not allowed but `a[+i]` is allowed because `+i` will always produce a number. In addition, ADsafe removes all sources of non-determinism such as `Date` and `Math.random()`.

To make use of ADsafe, widgets must be loaded and executed via the `ADSAFE.go()` method. These widgets must adhere to the ADsafe subset, although there is no verification built into ADsafe. Instead, it is recommended to verify subset adherence in any stage of the deployment pipeline with e.g. JSLint [2], a JavaScript code quality verification tool.

ADsafe does not allow JavaScript code to make use of the DOM directly. Instead, ADsafe makes a `dom` object available which provides and mediates access to the DOM.

No performance evaluation has been published about ADsafe by its author, who claim that ADsafe "will not make scripts bigger or slower or alter their behavior" [16]. This claim applies if advertisement scripts are written in the ADsafe subset directly, and not translated from full JavaScript.

Research on ADsafe has revealed several problems and vulnerabilities, which allow leaking the document object [83], launch a XSS attack [25], allow the guest to access properties on the host page's global object [75], prototype poisoning [47] and more.

3.3 Facebook JavaScript

Facebook JavaScript (FBJS) is a subset of JavaScript and part of the Facebook Markup Language (FBML) which was used to publish third-party Facebook

applications on the Facebook servers. FBJS was designed to allow web application developers as much flexibility as possible while at the same time protecting site integrity and the privacy of Facebook's users.

The FBJS subset excludes some of JavaScript's dangerous constructs such as `eval()`, `with`, `_parent_`, `constructor` and `valueOf()`. A preprocessor rewrites FBJS code so that all top-level identifiers in the code are prefixed with an application-specific prefix, thus isolating the code in its own namespace.

```
1 // original code
2 (function() { return this; })();

3 // code rewritten by FBJS
4 (function() { return ref(this); })();
```

Listing 1.7: Example JavaScript code making use of `this` semantics to return the global object and the code rewritten by FBJS to prevent FBJS code from breaking out of its namespace.

Special care is also taken with e.g. the use of `this` and object indexing to retrieve properties, making sure that a Facebook application cannot break out of its namespace. The semantics of i`this` are dependent on the way and location that it is used. A code fragment such as the one listed in Listing 1.7 can return the global object, allowing FBJS code to break out of its namespace. To remedy this problem, the FBJS rewriter encloses all references to `this` with the function `ref()`, e.g. `ref(this)`. This `ref()` function verifies the way in which it is called at runtime, and prevent FBJS code from breaking out of its namespace. Similarly, the FBJS rewriter also encloses object indices such as `property` in `object["property"]` with `idx("property")` to also prevent that `this` is bound to the global object.

Research on FBJS has revealed some vulnerabilities [46,47], which were addressed by the Facebook team.

Maffeis et al. [47] discovered that a specially crafted function can retrieve the current scope object through JavaScript's exception handling mechanism, allowing the `ref()` and `idx()` functions to be redefined. This redefinition in turn allows a FBJS code to break out of its namespace and take over the webpage.

After Facebook fixed the previous issues, Maffeis et al. [46] discovered another vulnerability which allows the global object to be returned on some browsers, by tricking the fixed `idx()` function to return an otherwise hidden property, through a time-of-check-time-of-use vulnerability [62].

3.4 Caja

Google's Caja, short for Capabilities Attenuate JavaScript Authority, is a JavaScript subset and rewriting system using a server-side middlebox. Caja represents an object-capability safe subset of JavaScript, meaning that any code conforming to this subset can only cause effects outside itself if it is given references to other objects. In Caja, objects have no powerful references to other

objects by default and can only be granted new references from the outside. The capability of affecting the outside world is thus reflected by holding a reference to an object in that outside world.

The Caja subset removes some dangerous features from the JavaScript language, such as `with` and `eval()`. Furthermore, Caja does not allow variables or properties with names ending in "_" (double-underscore), while at the same time marking variables and properties with names ending in "_" as private.

```
1 window.alert("hello world");
```

Listing 1.8: Example JavaScript code to be cajoled by Caja.

```
1 var tamedwindow = tame(window);
2 var cajoledcode = function(param) {
3    param.alert("hello world");
4 };
5 cajoledcode(tamedwindow);
```

Listing 1.9: Conceptual cajoled code and tamed window.

Caja's rewriting mechanism, known as the "cajoler," examines the guest code to determine any free variables and wraps the guest code into a function without free variables. Listing 1.8 shows some example code and its cajoled form is shown in Listing 1.9 (the `cajoledcode` variable). In addition, Caja adds inline checks to make sure that Caja's invariants are not broken and that no object references are leaked. The output of the cajoler is cajoled code, which is sent to a client's browser.

On the client-side, objects from the host webpage are "tamed" so that they only expose desired properties before being passed to the cajoled guest code. These tamed objects with carefully exposed properties are the only references that cajoled code obtains to the host page. In this way, all accesses to the DOM can be mediated by taming the global object before passing it to cajoled code. Listing 1.9 shows how the `window` object is tamed and passed to the cajoled form of Listing 1.8.

3.5 Discussion

The JavaScript language makes static code verification difficult, because of its dynamic nature (e.g. `eval()`) and strange semantics (e.g. the `with` construct). JavaScript subsets eliminate some of JavaScript's language constructs so that code may be more easily verified. When required, JavaScript rewriting systems can transform the code so that policies can also be enforced at runtime.

This section discussed four JavaScript subsets and rewriting mechanisms: BrowserShield, ADsafe, Facebook JavaScript and Caja. Some of their features are summarized in Table 1.

Table 1. Comparison between prominent JavaScript sandboxing systems using subsets and rewriting systems.

System	Target application	Rewrites	Uses subset	Removed features	Performance	Known weaknesses
BrowserShield	Preventing browser exploitation	Y	N	n/a	max. 136x slowdown on micro-benchmarks, avg. 2.7x slowdown on user experience	
ADsafe	Advertising	N	Y	`eval()`, `with`, `this`, global vars, ...	no slowdown	[83] [25] [75] [47]
FBJS	Third-party widgets	Y	Y	`eval()`, `with`, ...	no data	[47] [46]
Caja	Third-party widgets	Y	Y	`eval()`, `with`, ...	no data	

It is noteworthy that all three JavaScript subsets remove `with` and `eval()` from the language, which is in line with the standardized JavaScript strict mode subset. The only available performance benchmarks are for BrowserShield, which rewrites code written in full JavaScript, and indicate a heavy performance penalty when rewriting JavaScript in a middlebox. Furthermore, the list of known weaknesses suggest that creating a secure JavaScript subset, although possible, is not an easy task.

JavaScript subsets and code rewriting have been used in real world web applications and have proved to be effective in restricting available functionality to selected pieces of JavaScript code. However, restricting the integration of third-party JavaScript code which conforms to a specific JavaScript subset, puts limitations on third-party JavaScript library developers which they are unlikely to follow without incentive. Even if these developers are willing to limit themselves to a JavaScript subset, they would need to create a version of their code for every subset that they need to conform too. For instance, the jQuery developers would need to create a specific version for use with FBJS, Caja, ADsafe etc. This is an unrealistic expectation.

The standardization of a JavaScript subset, such as e.g. strict mode, helps eliminate this disadvantage for third-party JavaScript providers. But even with a standardized JavaScript subset to aid with code verification, this verification step itself must still happen in a middlebox located at either the server-side or the client-side.

Opting for a middlebox on a server-side has the disadvantage that it changes the architecture of the Internet. From the browser's perspective, JavaScript code would need to be requested from the middlebox instead of directly downloading it from the third-party script provider. Although this poses no problem for generic JavaScript libraries such as jQuery, it does pose a problem for JavaScript code which is generated dynamically depending on the user's credentials, as is the case with e.g. JSONP. In the latter case the third-party script provider might require session information to prove a user's identity, which will not be provided by the browser when requesting said script from a server-side middlebox.

A client-side middlebox on the other hand, does not suffer from this particular problem because it has the option of letting the browser connect to it transparently, e.g. in case of a web proxy. With a client-side middlebox, the web application developers lose control over the rewriting process. Users of the web application should setup the middlebox on the client-side in order to make use of this web application. But requiring users to install a middlebox next to their browser for a single web application, hurts usability and puts a burden on users which they might not like to carry.

From a usability viewpoint, it makes more sense to require only a single middlebox which can be reused for multiple web applications and to integrate this client-side middlebox into the browser somehow.

4 JavaScript Sandboxing Using Browser Modifications

The previous section showed that JavaScript contains several language constructs that cannot easily be verified to be harmless before executing JavaScript code. Instead of verifying the code beforehand, another approach is to control the execution of JavaScript at runtime and monitor the effect of the executing JavaScript to make sure no harm is done.

In a typical modular browser architecture of a browser, as explained in Sect. 2.1, the JavaScript environment is disconnected from other browser components. These other components, such as the DOM, the network layer, the rendering pipeline or HTML parser are not directly accessible to JavaScript code running in the JavaScript environment. Without these components, JavaScript is effectively side-effect free and is unable to affect the outside world.

The connection layer between the JavaScript engine and the different browser components, is an excellent location to mediate access to the powerful functionality that these components can provide. In order to enforce a policy at this location, the browser must be modified with a mechanism that can intercept, modify and block messages between the JavaScript engine and the different components.

```
1 var x = ...;
2 window.setTimeout(x, 1000);
```

Listing 1.10: Example JavaScript calling `setTimeout()` with unknown input.

Example: Allowing only Function Object Parameters for `setTimeout()`. Consider the example in Listing 1.10. In this example, the DOM API function `setTimeout()` is called with a parameter `x`. The specification for the `setTimeout()` function in the Web application API standard [98] lists two versions: a version where `x` must be a `Function` object, and a version that allows it to be a `String`. Passing a string to the `setTimeout()` function is regarded as a bad coding practice and considered as evil as using `eval()` [41]. Because of the inherent difficulty in verifying JavaScript code before runtime, it can be desirable to enforce a policy at runtime which rejects calls to `setTimeout()` when a string is passed as an argument.

The `setTimeout()` function is provided by a browser component which implements timer functionality. To access this function, the JavaScript engine must send a message to this component to invoke the timer functionality, as shown in Fig. 8. At this point, a browser modified with a suitable policy enforcement mechanism can intercept the message, and reject it if the given parameter is not a `Function` object.

Forms of Browser Modifications. Browser modifications can take many forms, but they can generally be split into three groups: browser plugins, browser extensions and browser core modifications.

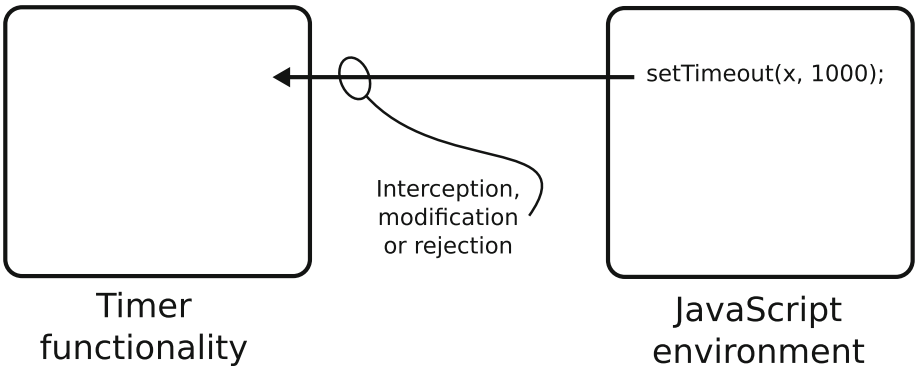


Fig. 8. Executing the `setTimeout()` function will send a message from the JavaScript environment to the component implementing timer functionality, which can be intercepted, modified or rejected by a policy enforcement mechanism in a modified browser.

Browser plugins and browser extensions can add extra functionality to the browser that can be used to enforce a JavaScript sandboxing technique. They are however limited in the modifications they can make in the browser environment.

For more advanced modifications to the browser, such as e.g. the JavaScript engine or the HTML parser, it is typically the case that neither plugins nor extensions are suitable. Therefor, modifying the browser core itself is required.

Research on JavaScript sandboxing through some form of browser modification, includes BEEP [38], ConScript [51], WebJail [91], Contego [45], AdSentry [19], JCShadow [72], Escudo [37], JAMScript [39], ...

4.1 Browser-Enforced Embedded Policies (BEEP)

Jim et al. introduce Browser-Enforced Embedded Policies, a browser modification that introduces a callback mechanism, called every time JavaScript is about to be executed. The callback mechanism provides a hook named `afterParseHook` inside the JavaScript environment, which can be overridden by the web developer.

Every time a piece of JavaScript is to be executed, the browser calls the `afterParseHook` callback to determine whether the piece of JavaScript is allowed to execute or not. To be effective, BEEP must be the first JavaScript code to load in the JavaScript environment, in order to set up the `afterParseHook` callback.

```
if (window.JSSecurity) {
  JSSecurity.afterParseHook =
    function(code, elt) {
      if (whitelist[SHA1(code)]) return true;
      else return false;
    };
  whitelist = new Object();
  whitelist["478zB3KkS+UnP2xz8x62ugOxvd4="] = 1;
  whitelist["AO0q/aTVjJ7EWQIsGVeKfdg4Gdo="] = 1;
  ... etc. ...
}
```

Listing 1.11: Example whitelist policy implemented in BEEP's `afterParseHook` function, from [38].

The authors experimented with two types of policies: whitelisting and DOM sandboxing.

In the whitelisting policy approach, illustrated in Listing 1.11, the **`afterParseHook`** callback function receives the script to be executed, and hashes is with the SHA-1 hashing algorithm. This hash is then compared with a list of hashes for allowed scripts. If the hash is found among this whitelist, the **`afterParseHook`** callback returns **`true`** and the script is executed.

```
<div class="noexecute">
  <!-- possibly-malicious content starts here -->
  <script>
      alert("hello world");
  </script>
  <!-- possibly-malicious content ends here -->
</div>
```

Listing 1.12: Example HTML with the `noexecute` attribute to be used with BEEP's DOM sandboxing policy.

```
<div class="noexecute">
  <!-- possibly-malicious content starts here -->
  </div><script>
      alert("hello world");
  </script><div>
  <!-- possibly-malicious content ends here -->
</div>
```

Listing 1.13: A node-splitting attack against the example in Listing 1.12. Notice how the enclosing `<div>` element with `noexecute` attribute is closed by an attacker-injected closing `<div>` element.

In the DOM sandboxing policy approach, illustrated in Listing 1.12, HTML elements in the web page are clearly marked with a **`noexecute`** attribute if they

can potentially contain untrusted content such as third-party advertising. When a script is about to be executed, the `afterParseHook` callback function receives both the script and the DOM element from which the execution request came. The `afterParseHook` callback function then walks the DOM tree, starting from the given DOM element and following the references to parent nodes. For each DOM node found in this walk, the callback function checks for the presence of a `noexecute` attribute. If such an attribute is found, the `afterParseHook` callback function returns `false`, rejecting script execution.

The authors report two problems with this last approach. First, in an attack to which the authors refer to as "node-splitting," an attacker may write HTML code into the webpage, allowing him to break out of the enclosing DOM element on which a `noexecute` attribute is placed. Shown in Listing 1.13, an attacker could easily break out of the DOM sandboxing policy by closing and opening the enclosing `<div>` tag which has the `noexecute` attribute set, hereby escaping its associated policy of rejecting untrusted scripts. Second, an attacker can introduce an HTML frame, which creates a child document. The `afterParseHook` callback function inside this child document would not be easily able to walk up the parent's DOM tree to check for `noexecute` attributes.

BEEP was implemented in the Konqueror and Safari browsers, and partially in Opera and Firefox. Performance evaluation indicates an average of 8.3 % and 25.7 % overhead on the loadtime of typical webpages for a whitelist policy and DOM sandboxing policy respectively.

4.2 ConScript

Meyerovich et al. present ConScript, a client-side advice implementation for Microsoft Internet Explorer 8. ConScript allows a web developer to wrap a function with an advice function using *around advice*. The advice function is registered in the JavaScript engine as *deep advice* so that it cannot be altered by an attacker.

As with BEEP, ConScript's policy enforcement mechanism must be configured before any untrusted code gains access to the JavaScript execution environment. ConScript introduces a new attribute `policy` to the HTML `<script>` tag, in which a web developer can store a policy to be enforced in the current JavaScript environment. When the web page is loaded, ConScript parses this `policy` attribute and registers the contained policy.

Unlike shallow advice, which is within reach of attackers and must be secured in order to prevent tampering by an attacker, ConScript registers the advice function as "deep advice" inside the browser core, out of reach of any potential attacker.

```
<head>
  <script policy='
    let httpOnly: K -> K = function(_ : K) {
      curse(); throw "HTTP-only cookies"; };
    around(getField(document, "cookie"), httpOnly);
    around(setField(document, "cookie"), httpOnly);
    '>
  </script>
</head>
```

Listing 1.14: Example HttpOnly cookie policy defined on a script element using ConScript, adapted from ConScript [51].

Listing 1.14 shows a ConScript policy being defined in the head of a web page. The policy in this particular example enforces the usage of "HttpOnly" [57] cookies, a version of HTTP cookies which cannot be accessed by JavaScript. To achieve this goal, the policy defines a function `HttpOnly` which simply throws an exception, and registers this function as "around" advice on the getter and setter of the `cookie` property of the `document` object, from which regular cookies are accessible in JavaScript.

Using *around advice* as an advice function allows a policy writer full freedom to block or allow a call to an advised function, possibly basing the decision on arguments passed to the advised function at runtime.

ConScript uses a ML-like subset of JavaScript with labeled types and formal inference rules as its policy language, which can be statically verified for common security holes. To showcase the power of ConScript and its policy language, the authors define 17 example policies addressing a variety of observed bugs and anti-patterns, such as: disallowing inline scripts, restricting XMLHttpRequests to encrypted connections, disallowing cookies to be leaked through hyperlinks, limiting popups and more.

ConScript was implemented in Microsoft Internet Explorer 8 and its performance evaluated. On average, ConScript introduces a slowdown during micro-benchmarks of 3.42x and 1.24x after optimizations. The macro-benchmarks are reported to have negligible overhead.

4.3 WebJail

Van Acker et al. propose WebJail, a JavaScript sandboxing mechanism which uses deep advice functions like ConScript.

In WebJail, HTML iframe elements are used as the basis for a sandbox. A new `policy` attribute for an iframe element allows a web developer to specify the URL of a WebJail policy, separating concerns between web developers and policy makers.

The authors argue that an expressive policy language such as ConScript's can cause confusion with the integrators who need to write the policy, thus slowing the adoption rate of a sandboxing mechanism. In addition, they warn for a scenario dubbed "inverse sandbox," in which the policy language is so expressive that an attacker may use it to attack a target web application by

sandboxing it with a well-crafted policy. For instance, if the policy language is the JavaScript language, an attacker may define a policy on an iframe to intercept any cookie-access and transmit these cookies to an attacker-controlled host. A target web-application could then be loaded into this iframe and would, upon accessing its own cookies, trigger the policy mechanism which leaks the cookies to the attacker.

```
{
  "framecomm": "yes",
  "extcomm": ["google.com", "youtube.com"],
  "device": "no"
}
```

Listing 1.15: Example WebJail policy allowing inter-frame communication, external communication to Google and YouTube, but disallowing access to the Device API, from [91].

To avoid this scenario, WebJail abstracts away from an overly expressive policy language and defines its own secure composition policy language. Based on an analysis of sensitive JavaScript APIs in the HTML5 specifications, the authors divided the APIs into nine categories. The policy consists of a file written in JSON, describing access rights for each of these categories. Access to a category of sensitive JavaScript APIs in WebJail can be granted or rejected with `"yes"` or `"no"`, or determined based on a whitelist of allowed parameters. Listing 1.15 shows an example WebJail policy which allows inter-frame communication (`"framecomm": "yes"`), external communication to Google and YouTube (`"extcomm": ["google.com", "youtube.com"]`), but disallowing access to the Device API (`"device": "no"`).

WebJail's architecture, depicted in Fig. 9 consists of three layers to process an integrator's policy and turn it into deep advice. The policy layer reads an iframe's policy and combines with the policies of any enclosing iframes. Policy composition is an essential step to ensure that an attacker cannot easily escape the sandbox by creating a child document without a policy defined on it. The advice construction layer processes the composed policy and creates advice functions for all functions in the specified JavaScript APIs. Finally, the deep aspect weaving layer combines the advice functions with the API functions, turning them into deep advice and locking them safely inside the JavaScript engine.

WebJail was implemented in Mozilla Firefox 4.0b10pre for evaluation. The performance evaluation indicated an average of between 6.4 % and 27 % for micro-benchmarks and an average of 6 ms loadtime overhead for macro-benchmarks.

4.4 Contego

Luo et al. design and implement Contego, a capability-based access control system for browsers.

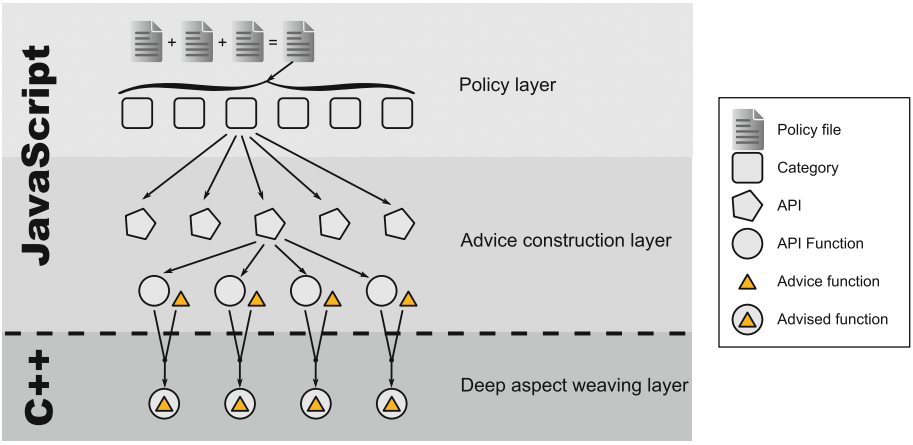


Fig. 9. The WebJail architecture consists of three layers: the policy layer, the advice construction layer and the deep aspect weaving layer, from [91].

In a capability-based access control model, the ability of a principal to perform an action is called a capability. Without the required capability, the principal cannot perform the associated action.

Contego's authors identified a list of capabilities in browsers, among which: performing Ajax requests, using cookies, making HTTP GET requests, clicking on hyperlinks, ... They list three types of actions that can be associated with those capabilities, based on where they originate: HTML-induced actions, JavaScript-induced actions and event-driven actions.

```
<div cap="110001111"> ... </div>
<!--
  Capability bitstring:
    1 AJAX POST request allowed
    1 AJAX GET request allowed
    0 Cookie setting not allowed
    0 Cookie getting not allowed
    0 Cookie using not allowed
    1 HTTP GET request allowed
    1 HTTP POST request allowed
    1 Hyperlink click allowed
    1 Button submit click allowed
-->
```

Listing 1.16: Example usage of Contego and its capability bitstring, from [45].

Contego allows a web developer to assign capabilities to `<div>` elements in the DOM tree, by assigning a bit-string to the `cap` attribute. Each bit in the bit-string indicates whether a certain capability should be enabled (`"1"`) or disabled (`"0"`) for all DOM elements enclosed by the `<div>` element on which the capabilities apply. The meaning of each bit in the bit-string is shown in Listing 1.16, which also shows an example policy disabling access to cookies.

The authors warn about a node-splitting attack when an attacker is allowed to insert content into a `<div>` element. Just as with BEEP's DOM sandboxing policy, care should be taken to avoid that an attacker can insert a closing tag and escape the policy. In addition, Contego has measures in place to ensure that an attacker cannot override capability restrictions by e.g. setting a new `cap` attribute either in HTML or in JavaScript. Cases where principals with different capabilities interact are handled by restricting the actions to the conjunction of the capability sets.

To implement Contego in the Google Chrome browser, the authors extended the browser with two new components: the binding system and the enforcement system. The binding system assigns and tracks individual principal's capabilities within a webpage. The enforcement system then uses the information from the binding system to allow or deny actions at runtime.

The performance evaluation shows an average overhead of about 3 % on macro-benchmarks.

4.5 AdSentry

Dong et al. propose AdSentry, a confinement solution for JavaScript-based advertisements, which executes the advertisements in a special-purpose JavaScript engine.

An architectural overview of AdSentry is shown in Fig. 10. Next to the regular JavaScript engine, AdSentry implements an additional JavaScript engine, called the shadow JavaScript engine, as a browser plugin. The browser plugin is built on top of the Native Client (NaCl) [30] sandbox, which protects the browser

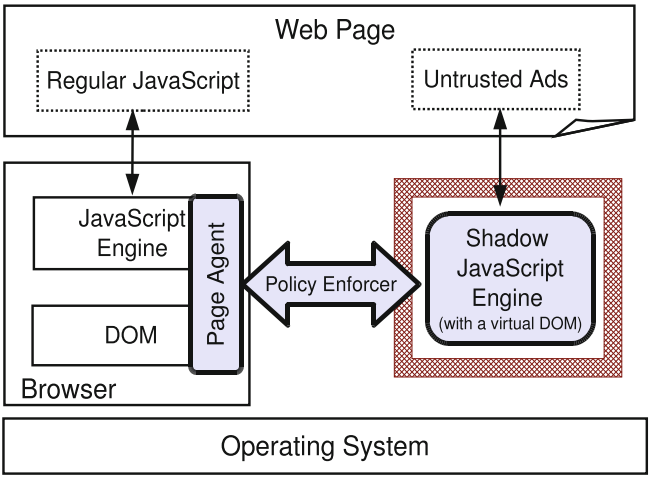


Fig. 10. The AdSentry architecture: advertisements are executed in a shadow JavaScript engine which communicates with the Page Agent via the policy enforcer, from [19].

and the rest of the operating system from drive-by-download attacks occurring inside the sandbox.

Advertisements can either be explicitly marked for use with AdSentry, or they can be automatically detected by AdBlock Plus. When an advertisement is detected in a webpage, AdSentry assigns it a unique identifier and communicates with the shadow JavaScript engine to request that the code be executed there. The shadow JavaScript engine then creates a new JavaScript context with its own global object and virtual DOM and executes the advertisement.

```
1  msg ::= command data
2  command ::= script | callFunc | getProp
3                | setProp | return
4  data ::= <text>
```

Listing 1.17: Format of the communication protocol used between AdSentry's shadow JavaScript engine and the Page Agent, from [19].

The virtual DOM inside the shadow JavaScript context has no access to the real webpage on which the advertisement is supposed to be rendered. Instead, the methods of the virtual DOM are stubs which trigger the shadow JavaScript engine to communicate with a Page Agent in the real JavaScript engine, requesting access on behalf of the advertisement. The communication between the Page Agent and the shadow JavaScript engine is facilitated with a data exchange protocol, shown in Listing 1.17. This communication channel is also where AdSentry's enforcement mechanism operates, granting or blocking access to the real webpage's DOM according to a user-specified policy. No information is given on how this policy can be specified.

AdSentry was implemented in Google Chrome, and uses a standalone version of SpiderMonkey, Mozilla's JavaScript engine, as the shadow JavaScript engine. The performance evaluation indicates an average overhead of 590x on micro-benchmarks when traversing the boundary between the shadow JavaScript engine and the Page Agent, and an around 3 % to 5 % overall loadtime overhead on macro-benchmarks.

4.6 Discussion

This section discussed five browser modifications that aim to isolate and restrict JavaScript code in the web browser: BEEP, ConScript, WebJail, Contego and AdSentry. Some of their features are summarized in Table 2.

JavaScript sandboxing through a browser modification allows the integration of third-party scripts written in the full JavaScript language. Web applications can be built with a much richer set of JavaScript libraries, since those JavaScript libraries are not confined to a subset of JavaScript.

In addition, a browser modification can control the execution of JavaScript inside the browser, allowing the construction of efficient custom-built machinery to enforce a sandboxing policy, ensuring low overhead.

Table 2. Comparison between prominent JavaScript sandboxing systems using a browser modification.

System	Target application	Isolation unit	Restricts	Policy expressiveness	Deployment	Browser	Performance	Known weaknesses
BEEP	restrict scripts	entire JS environment	execution of JS scripts	full JavaScript to indicate "accept" or "reject"	`afterParseHook` implementation by integrator	Konqueror, Safari, partially Opera, partially Firefox	8.3 % to 25.7 % macro	node-splitting
ConScript	sandboxing	entire JS environment	??? anything	high: own JS subset	`policy` attribute on script element	MSIE	1.24x to 3.42x micro, negligible macro	
WebJail	sandboxing	entire JS environment + subframes	access to sensitive APIs	yes/no/whitelist	`policy` attribute on iframe element	Firefox	6.4 % to 27 % micro, 6 ms macro	
Contego	restrict capabilities	`<div>` element	capabilities	bitstring	`cap` attribute on div element	Chrome	3 % macro	
AdSentry	advertisement	shadow JavaScript engine	access to the DOM	???	???	Chrome	590x micro, 3 % to 5 % macro	

However, modified browsers pose a problem with regard to dissemination of the software and compatibility with browsers and browser versions. End-users must take extra steps in order to enjoy the protection of this type of JavaScript sandboxing systems.

Because end-users do not all use the same browser, it becomes impossible to assure that all end-users can keep using their own favorite browser. In the most fortunate case, the developers of this browser core modification may find a way to port their sandboxing system to all browsers. Even if this is the case, a browser core modification is a fork in a browser's code base and must be maintained to keep up with changes in the main code base, which can be a significant time investment.

Likewise, a browser plugin or extension implementing a certain JavaScript sandboxing system, must also be created for all browser vendors and versions, to enable a wide range of users to make use of it. Such a plugin or extension must equally be maintained for future releases of browsers, which can also require a significant time investment.

All in all, modifying a browser through a fork of browser code, a browser plugin or a browser extension in order to implement a JavaScript sandboxing system, is acceptable for a prototype, but proves difficult in a production environment.

An alternative approach is to convince major browser vendors to implement the browser modification as part of their main code base, or even better, pass it through the standardization process so that all browser vendors will implement it. This approach will ensure that the sandboxing technique ends up in a user's favorite browser automatically and that the code base is maintained by the browser vendors themselves.

Unfortunately, getting a proposal accepted by the standardization committees is not a straightforward task, partly because no solution is widely accepted as being "The Solution."

In recent years, the standardization process has yielded new and powerful functionality that could be used to build a JavaScript sandboxing system. Through this approach, a JavaScript sandboxing system would not need any browser modification at all and work out of the box on all browsers that support the latest Web standards.

5 JavaScript Sandboxing Without Browser Modifications

The previous section showed that a sandboxing mechanism implemented as a browser modification, can be used to restrict JavaScript functionality available to untrusted code at runtime. A browser modification is useful for proof-of-concept evaluation of a sandboxing mechanism, but proves problematic in a production environment. Not only must a browser modification be maintained with new releases of the browser on which it is based, but end-users must also be convinced to install the modified browser, plugin or extension.

Given the powerful nature of JavaScript, it is possible to isolate and restrict untrusted JavaScript code at runtime, without the need for a browser modification. This approach is challenging because the enforcement mechanism will execute in the same execution environment as the untrusted code it is trying to restrict. Special care must be taken to ensure that the untrusted code cannot interfere with the enforcement mechanism, and this without any added functionality to protect itself from the untrusted code.

Isolation Unit and Communication Channel. Following the same rationale as in the previous section, a good approach is to create an isolated unit (or sandbox) which is completely cut off from any sensitive functionality, reducing it to a side-effect free execution environment. Figure 11 sketches the relationship between a sandbox and the real JavaScript environment.

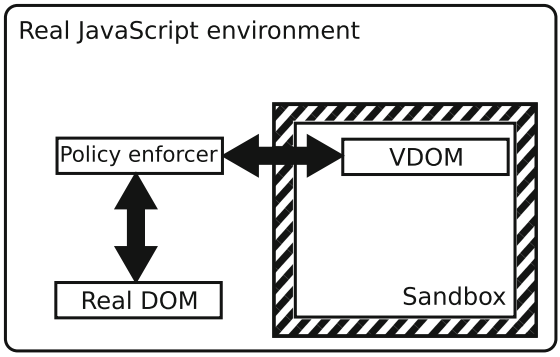


Fig. 11. Relationship between the real JavaScript environment and a sandbox. The sandbox can only interact with a Virtual DOM, which forwards it via the policy enforcer to the real DOM.

Any untrusted code executed in the sandbox, will not be able to affect the outside world, except through a virtual DOM introduced into this sandbox. To access the outside world, the isolated code must make use of the virtual DOM, which will forward the access request over a communication channel to an enforcement mechanism. If the access is allowed, the enforcement mechanism again forwards the access request to the real JavaScript environment.

New and Powerful ECMAScript 5 Functionality. The rise of Web 2.0 resulted in the standardization of ECMAScript 5, which brought new and powerful functionality to mainstream browsers. This new functionality can help with the isolation and restriction of untrusted JavaScript code.

An example of such functionality is the WebWorker API, or WebWorkers [93]. WebWorkers allow web developers to spawn background workers to run in parallel with a web page. These workers are intended to perform long-running com-

putational tasks in the background, while keeping web pages responsive to user interaction.

WebWorkers have a very restricted API available to them, which only allows them to do very basic tasks such as set timers, perform XMLHttpRequests or communicate through `postMessage()`. In particular, WebWorkers have no access to the DOM. Communication between WebWorkers and a web page is achieved through the postMessage API.

Having new ECMAScript 5 functionality in place in browsers today, opens new options for JavaScript sandboxing mechanisms which previously required browser modifications or code verification/transformation in a separate middlebox.

For instance, because WebWorkers restrict JavaScript code from accessing the DOM and other sensitive JavaScript functionality, they can be used as the isolation unit for a JavaScript sandboxing mechanism. TreeHouse, discussed farther in this section, uses WebWorkers as its isolation unit.

Research on JavaScript sandboxing without browser modification includes Self-protecting JavaScript [48,74], AdJail [84], Object Views [50], JSand [6], TreeHouse [35], Privilege-separated JavaScript [7], SafeScript [85], Pivot [52], IceShield [34], SafeJS [14], Two-tier sandbox [73], Virtual Browser [13], ... A selection of this work is discussed in the following sections.

5.1 Self-Protecting JavaScript

Phung et al. propose a solution where DOM API functions are replaced by wrappers which can optionally call the original functions, to which the wrapper has unique access. The wrappers can be used to enforce a policy and, with the ability to store state inside the wrapper function's scope, allow the enforcement of very expressive policies. Access to sensitive DOM properties can also be limited by defining a getter and setter method on them which implements a restricting policy.

```
var wrapper = (function (original) {
    // counter keeps state across
    // multiple function calls
    var counter = 0;

    // create and return the wrapper
    return function(m) {
        if(counter < 2) {
            original(m);
            counter++;
        }
    }
})(window.alert);

window.alert = wrapper;
```

Listing 1.18: Simplified version of Self-protecting JavaScript's creation of a wrapper around the `alert()` function, allowing it to be called maximum twice.

An example of how a DOM function is replaced with a wrapper, is shown in Listing 1.18. In this example, a wrapper for the function `alert()` is created with a built-in policy to only allow the function to be called twice. A reference to the original native implementation of `alert()` is kept inside the wrapper's scope chain, making it only accessible by the wrapper itself. Finally, the original `alert()` function is replaced by the wrapper.

It is vital that the wrappers are created and put in place of the original DOM functions before any other JavaScript runs inside the JavaScript environment, to achieve full mediation. If any untrusted JavaScript code is run before the wrappers are in place, an attacker may keep copies of the original DOM functions around, thus bypassing any policies that are placed on them later.

The authors warn that references to DOM functions can also be retrieved through the `contentWindow` property of newly created child documents. To prevent this, access to the `contentWindow` property is denied.

A bug in the `delete` operator of older Firefox browsers also allows overwritten DOM functions to be restored to references to their original native implementations, by simply deleting the wrappers.

A performance evaluation of Self-protecting JavaScript revealed a average of 6.33x slowdown on micro-benchmarks, and a 5.37 % average overhead for macro-benchmarks.

Magazinius et al. [48] analyzed Self-protecting JavaScript and uncovered several weaknesses and vulnerabilities that allow the sandboxing mechanism to be bypassed by an attacker.

They note that the original implementation does not remove all references to DOM functions from the JavaScript environment, leaving them open to abuse from attackers. The `alert()` function for instance, has several aliases (such as `window.__proto__.alert()`), which must all be replaced with a wrapper for Self-protecting JavaScript to be effective.

Equally, simply denying access to the `contentWindow` property is not sufficient to prevent references to DOM functions from being retrieved from child documents. These references can also be access from child documents through the `frames` property of the `window` object, or from the parent document through the `parent` property of the `window` object.

They also point out that Self-protecting JavaScript is vulnerable to several types of prototype poisoning attacks, allowing an attacker to get access to the original, unwrapped DOM functions as well as the internal state of a policy wrapper.

Lastly, they remind that an attacker could abuse the caller chain during a wrapper's execution, by gaining access to the non-standard `caller` property available in functions, allowing an attacker to gain access to the unwrapped DOM functions.

Finally, Magazinius et al. offer solutions to remedy these vulnerabilities by making sure any functions and objects used inside a wrapper are disconnected from the prototype chain to prevent prototype poisoning, and coercing

parameters of functions inside wrappers to their expected types in order to further reduce the attack surface.

5.2 AdJail

Ter Louw et al. propose AdJail, an advertising framework which enforces JavaScript sandboxing on advertisements.

AdJail allows a web developer to restrict what parts of the web page an advertisement has access to, by marking HTML elements in that web page with the `policy` attribute. This `policy` attribute contains the AdJail policy that is in effect for a certain HTML element and its sub-elements.

The AdJail policy language allows the specification of what HTML elements can be read or written to, and whether that access extends to its sub-elements. The web developer can also define a policy to enable or disable images, Flash or iframes, restrict the size of an advertisement to a certain height and width and allow clicked hyperlinks to open web pages in a new window.

By default, an advertisement is positioned in the "default ad zone," an HTML `<div>` element that aids the web developer in positioning the advertisement in the web page. The default policy is set to "deny all."

An overview of AdJail is shown in Fig. 12. The advertisement is executed in a "shadow page," which is a hidden iframe with a different origin, so that it is isolated from the real web page. Those parts of the real web page's DOM that are marked as readable by the advertisement, are replicated inside the shadow page before the advertisement executes.

Changes made by the advertisement inside the shadow page, are detected by hooking into the DOM of the shadow page, and communicated to the real page through a tunnel script. The changes are replicated on the real page if

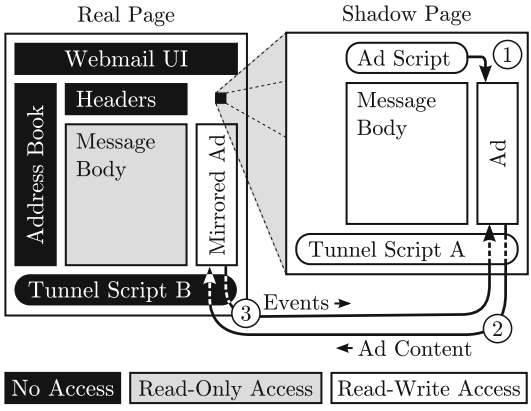


Fig. 12. Overview of AdJail, showing the real page, the shadow page and the tunnel scripts through which they communicate and on which the policy is enforced, from [84].

allowed by the policy. Likewise, events generated by the user on the real page, are communicated to the shadow page so that the advertisement can react to them.

Because AdJail is aimed at sandboxing advertisements, special care must be taken to ensure that the advertisement provider's revenue stream is not tampered with. In particular, AdJail takes special precautions to ensure that content is only downloaded once, to avoid duplicate registration of "ad impressions" on the advertisement network. Furthermore, AdJail leverages techniques used by BLUEPRINT [86] to ensure that an advertisement does not inject scripts into the real webpage.

Performance benchmarks indicate that AdJail has an average overhead of 29.7 % on ad rendering, increasing the rendering time from an average of 374 ms to 532 ms. Further analysis showed that AdJail has an average overhead of 25 % on the entire page loadtime, increasing it from 489 ms to 652 ms.

5.3 Object Views

Meyerovich et al. introduce Object Views, a fine grained access control mechanism over shared JavaScript objects.

An "Object View" is a wrapper around an object that only exposes a subset of the wrapped object's properties to the outside world. The wrapper consists of a proxy between the wrapped object and the outside world, and a policy that determines what properties should be made available through the proxy.

```
1  var wrapper = ...;
2  var obj = { prop: 123, func: function() {
3      alert("hello world");
4      }
5  };

6  defineSetter(wrapper, "prop",
7      function(x) {
8          obj.prop = x;
9      });

10 defineGetter(wrapper, "prop",
11     function() {
12         return obj.prop;
13     });

14 wrapper.func = function() {
15     obj.func(arguments);
16 };

17 alert(wrapper.prop); // displays 123
18 wrapper.prop = 456;  // sets obj.prop to 456
19 wrapper.func()       // displays "hello world"
```

Listing 1.19: Pseudo-code showing how an Object View around an object `obj` can be used to intercept reading and writing a property, and intercepting a function call, from [50].

Sketched in Listing 1.19, an Object View contains a getter and setter method for each property on the wrapped object, and a proxy function for each

function object. Writing a value to a property on an Object View, triggers the setter function which may eventually write the value to the wrapped object's respective property. The getter function works in a similar way for reading properties. Using a property of an Object View as a function and calling it, triggers the proxy function. Object Views are applied recursively to a proxy function's return value.

Creating two Object Views that wrap the same object, poses a problem with regard to reference equality. Although comparing the underlying objects of both object views would result in an equality, this would not be the case for the two wrapping Object Views. This inconsistent view can be prevented by only wrapping an object with an Object View once, and returning that same Object View every time a new Object View for the underlying object is requested.

Object views offer a basis for fine-grained access control through an aspect system. Each getter, setter and proxy function on an Object View can be combined with an "around" advice function, allowing the enforcement of an expressive policy.

Because of its size and complexity, manually wrapping the entire DOM with object views would be a difficult and error-prone process. Instead, the authors advocate a declarative policy system which is translated into advice for the Object Views.

```
{
    "selector": "(//*[@class=âĂŸexampleâĂŹ]) | (//*[@class=âĂŸexampleâĂŹ]//*)",
    "enabled": true,
    "defaultFieldActions": {read: permit},
    "fields": {shake: {methCall: permit}}
}
```

Listing 1.20: A declarative policy rule specifying that a DOM element of class `example` and its subtree are read-only. If a method `shake()` exists, it may be read and invoked as a method.

The declarative policy is specified by a set of rules consisting of an XPath [95] selector to specify a set of DOM nodes and an Enabled flag to indicate that the selected nodes may be accessed. Optionally, each rule can be extended with default and specific rules for each field of a DOM element. An example rule, shown in Listing 1.20, specifies that all DOM elements of class **`example`** and its subtree can be accessed (**`enabled = true`**) and is by default read-only (**`defaultFieldActions`**). A specific rule for a field called **`shake`** allows that field to be read and invoked as a method.

The authors discuss using Object Views in two scenarios: a scenario where JavaScript is rewritten[1] to make use of Object Views for same-origin usage, and a scenario where Object Views are used in cross-origin communication between frames.

[1] This work could also be listed under Sect. 3, but since the published paper mostly focuses on the cross-origin communication which does not require browser modifications, it is listed in this section instead.

In the latter scenario, each frame provides an Object View around its enclosed document to only expose the view required by the other. Communication between the frames is handled by marshaling requests for the other side to a string and transmitting it with `postMessage()`. Because each Object View has its own built-in policy, the communication channel does not need to enforce a separate policy.

The performance of Object Views was evaluated on a scenario where several objects are wrapped in a view, but where the communication between Object Views is not marshaled and transmitted with `postMessage()`. For this scenario, the average overhead is between 15 % and 236 % on micro-benchmarks.

5.4 JSand

Agten et al. propose JSand, a JavaScript sandboxing mechanism based on Secure ECMAScript (SES).

Secure ECMAScript (SES) is a subset of ECMAScript 5 strict which forms a true object-capability language, guaranteeing that references to objects can only be obtained if they were explicitly passed to an object-capability environment.

Without a reference to the DOM, JavaScript code running in a SES environment cannot affect the outside world. JSand wraps the global object using the Proxy API [20] and passes a reference to this proxied global object to the SES environment. Any access to the global object from inside the SES environment, will traverse the proxy wrapper on which a policy can be enforced.

Without additional care, JavaScript inside the SES environment with access to this proxied global object, can invoke methods that return unwrapped JavaScript objects. Such an oversight can cause a reference to the real JavaScript to leak into the SES environment, making JSand ineffective. To avoid this, JSand wraps return values recursively, according to the Membrane Pattern [61]. In addition, JSand preserves pointer equality between wrappers around the same objects, by storing created wrappers in a cache and returning an existing wrapper if one already exists.

Using the Membrane pattern, any access to the outside world from inside the SES environment, can be intercepted and subjected to a policy enforcement mechanism. The authors do not specify a specific policy implementation, but point out that JSand's architecture allows for expressive fine-grained and stateful policies.

There are two important incompatibilities between the SES subset and ECMAScript 5 code, which makes legacy JavaScript incompatible with JSand.

The first is the mirroring of global variables with properties on the global object and vice versa. When a global variable is created under ECMAScript 5, a property with the same name is created on the global object. Similarly, a property created on the global object results in the creation of a global variable of the same name. This ECMAScript 5 behavior is not present in SES and can cause legacy scripts who depend on that behavior, to break.

Second, because SES is a subset of ECMAScript 5 strict, it does not support the `with` construct, does not bind `this` to the global object in a function call and

does not create new variables during `eval()` invocations. Legacy scripts making use of this behavior will also break in SES.

To be backwards compatible with legacy JavaScript that does not conform to SES, JSand applies a client-side JavaScript rewriting step where needed before sandboxing the guest JavaScript code. The UglifyJS [59] JavaScript parser is used to parse JavaScript into an Abstract Syntax Tree (AST). This tree is then inspected and modified for legacy ECMAScript 5 constructs that will break in SES. In particular, JSand rewrites guest code so that the mirroring of global variables and properties of the global object in ECMAScript 5, is replicated explicitly. JSand also finds all occurrences to the `this` keyword and replaces it with an expression that replaces it with `window` if its value is undefined, thus also replicating ECMAScript 5 behavior.

JSand's performance evaluation indicates an average 9x slowdown for function-calls than traverse the membrane wrapper, resulting in an average of 31.2 % overhead in user experience when interacting with a realistic web application. The load-time of a web application is increased on average by 365 % for legacy web applications using ECMAScript 5 code which requires the rewriting step. The authors expect that this rewriting step will not be needed in the future, so that the average load-time overhead will drop to 203 %.

5.5 TreeHouse

Ingram et al. propose TreeHouse, a JavaScript sandboxing mechanism built on WebWorkers. As explained previously, WebWorkers are parallel JavaScript execution environments without a usual DOM, which can only communicate through `postMessage()`.

An overview of TreeHouse's architecture is shown in Fig. 13. TreeHouse loads guest JavaScript code into a WebWorker to isolate it from the rest of a web page. WebWorkers do not have a regular DOM, so TreeHouse installs a broker with a virtual DOM inside the WebWorker that emulates the DOM of a real webpage.

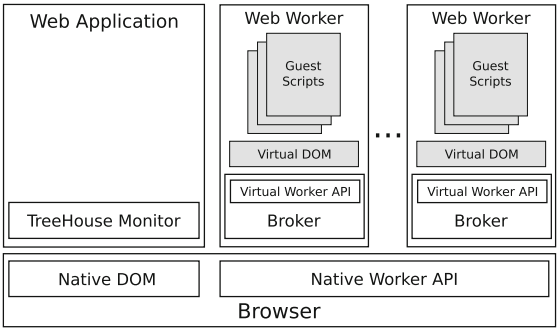


Fig. 13. TreeHouse architectural overview. Sandboxes consist of WebWorkers with a virtual DOM. Access to this virtual DOM is mediated by broker according to a policy. If access is allowed, the request is forwarded to the real page's monitor, from [35].

When this virtual DOM is accessed, the broker first consults the policy to determine whether access is allowed. If access is allowed, the broker then forwards the access request to the real page's "TreeHouse Monitor" using `postMessage()`, which handles the access to the real page's DOM.

TreeHouse offers two deployment options to web developers wishing to use its sandboxing mechanism. One option is to create a sandbox with a policy and load JavaScript in it manually using the TreeHouse API. Another option, is more user-friendly and allows a web developer to specify guest code to be sandboxed, in actual `<script>` elements. These `<script>` elements should have their `type` attribute set to `"text/x-treehouse-javascript"` to prevent them from being executed by the JavaScript engine in the host page. The special script type is also automatically detected by the TreeHouse Monitor, which will create sandboxes and load the script inside them.

```
<script src="tetris.js"
  type="text/x-treehouse-javascript"
  data-treehouse-sandbox-name="worker1"
  data-treehouse-sandbox-children="#tetris">
</script>
<script src="tetris-policy.js"
  type="text/x-treehouse-javascript"
  data-treehouse-sandbox-name="worker1"
  data-treehouse-sandbox-policy>
</script>
```

Listing 1.21: TreeHouse integration in a web page. Guest code is loaded into `<script>` tags with type `"text/x-treehouse-javascript"` so that they are automatically sandboxed. The policy is also specified in a `<script>` element marked with a `"data-treehouse-sandbox-policy"` attribute, from [35].

An example use of TreeHouse is shown in Listing 1.21. Here, the first `<script>` element shows how a sandbox is created called `"worker1"`, with access to the DOM element with id `"#tetris"` and its subtree. The `"tetris.js"` script is then loaded inside the sandbox and executed. The second `<script>` tag references the sandbox `"worker1"` and indicates through the `"data-treehouse-sandbox-policy"` attribute that the script `"tetris-policy.js"` should be interpreted as a policy instead of guest JavaScript code.

A TreeHouse policy consists of a mapping between DOM elements and rules. There are three types of rules: a rule can be expressed by a boolean, a function returning a boolean, or a regular expression. If the rule has a boolean value of `true`, access to the associated DOM element is allowed. If the rule is a function, that function is invoked at policy enforcement time by the broker, and access is allowed if the return value is `true`. Finally, if the rule is a regular expression, it refers to a property. If the regular expression matches a property's name, then the guest code is allowed to set a value to that property.

Because WebWorkers are concurrent by design, they present a problem when multiple TreeHouse sandboxes try to access to same DOM element in a real page. Such simultaneous access would cause a race condition and result in undefined

behavior. To prevent such a race condition, TreeHouse allows a DOM element to only be accessed by one sandbox.

Another concurrency problem arises when the guest code makes use of a synchronous method such as `window.alert()`. The guest code will expect the JavaScript execution to block, waiting for the end-user to click away the pop-up window. In reality, TreeHouse's communication channel between the host page and the WebWorkers is asynchronous because `postMessage()` is asynchronous. When calling `window.alert()` in the guest code, the broker would send an asynchronous message to the host page, and let code execution in the sandbox resume immediately. This conflicts with the guest code's expected behavior. The authors chose not to handle this case and raise a runtime exception when guest code calls synchronous methods.

The performance benchmarks for TreeHouse show an average slowdown of 15x to 176x for macro-benchmarks, and an average of 7x to 8000x slowdown on micro-benchmarks for method invocations on the DOM.

5.6 SafeScript

Ter Louw et al. propose SafeScript, a client-side JavaScript transformation technique to isolate JavaScript code in namespaces.

SafeScript makes use of Narcissus [67], a JavaScript meta-interpreter, to rewrite JavaScript code on the client-side and instrument the code so that it can interpose on the property resolution mechanism. Narcissus is a full JavaScript interpreter and can correctly handle all of JavaScript's strange semantics, its scoping, prototype chains and thus also the property resolution mechanism.

Through this rewriting step, SafeScript can separate JavaScript code in namespaces by manipulating the property resolution mechanism for each sandboxed script so that it ultimately resolves to its own isolated global object. Because property resolution is under SafeScript's control, it can effectively mediate access to the real DOM when sandboxed JavaScript guest code requests access to it.

```
1  <!-- the transformation tool -->
2  <script src='rewriter.js'></script>
3  <!-- an APIâĂŹs implementation -->
4  <script src='interface0.js'></script>

5  <script>
6  var namespace0 = $_sm[0]();
7  var script0_code = load_script('http://3rd.com/main.js');
8  exec_script(transform(script0_code, namespace0));
9  </script>
```

Listing 1.22: SafeScript used on a webpage. After loading the rewriter and API implementation, a namespace is created and the guest code is loaded. Afterwards, the guest code is transformed so that the property resolution mechanism is locked to the created namespace, and the transformed code is executed, from [85].

Listing 1.22 shows how SafeScript can be used to sandbox a given JavaScript. In this example, the `"rewriter.js"` script contains SafeScript's transformation code and `"interface0.js"` contains an API implementation for a "namespace 0." After creating the namespace with `$_sm[0]()`, the guest code is loaded from a third-party host, transformed so that the property resolution mechanism is locked to "namespace 0", and then executed.

SafeScript ensures that any dynamically generated JavaScript code is also transformed and isolated in a namespace. In order to do so, SafeScript traps methods such as `eval()`, `setTimeout()`, which can inject JavaScript code into the execution environment directly. To capture JavaScript code that is indirectly injected, SafeScript monitors methods such as `document.write()` and properties like `innerHTML`. HTML written through these injection points must first be parsed and have its JavaScript code extracted before it can be transformed by SafeScript.

Despite its many optimizations, SafeScript's performance benchmarks indicate an average slowdown of 6.43x on basic operations such a variable incrementation, because SafeScript rewrites every variable statement. The macrobenchmark reveals an average slowdown of 64x.

5.7 Discussion

This section discussed six JavaScript sandboxing mechanisms that do not require any browser modifications: Self-protecting JavaScript, AdJail, Object Views, JSand, TreeHouse and SafeScript. Some of their features are summarized in Table 3.

Besides Self-protecting JavaScript, which protects all access-routes to the DOM API through enumeration, all solutions isolate untrusted JavaScript in an isolation unit. The isolated JavaScript cannot access the DOM directly, but must communicate with the real web page and request access, which is then mediated by a policy enforcement mechanism.

JavaScript sandboxing systems that do not require browser modifications leverage existing standardized powerful functionality that is available in browsers today. The advantage of this approach is that standardized functionality is, or in the near future will be, available in all browsers and thus the sandbox works out of the box for all Internet users.

Much of the new browser functionality incorporated in the previously discussed JavaScript sandboxing systems, was not designed for sandboxing and may not perform well enough for a seamless user experience.

In the future that may change, because browser vendors optimize their code for speed to compete with other browser vendors. When new browser functionality becomes more popular, it will undoubtedly also be optimized for speed, automatically increasing the performance of the JavaScript sandboxing systems making use of it.

Web standards keep evolving, so that we can expect more advanced browser functionality in the future. This new functionality can then be used to design and implement yet more powerful JavaScript sandboxing systems. Ideally, this new

Table 3. Comparison between prominent JavaScript sandboxing systems not requiring browser modifications.

System	Target application	Isolation unit	Communication	Policy expressiveness	Deployment	Performance	Known weaknesses
Self-protecting JavaScript	sandboxing	JavaScript environment	n/a	high	library	6.33x micro, 5.37 % macro	[48]
AdJail	advertisements	shadow page	`postMessage`	read/write elements + enable/disable images/other	`policy` attribute	25 % macro	
Object Views	sandboxing	iframe	`postMessage`	get/set/call	declarative policy with XPath	15 % to 236 % micro	
JSand	sandboxing	SES environment	Membrane/Proxy API	high	VDOM implementation	9x micro, 203 % to 365 % macro on loadtime, 31.2 % macro on user experience	
TreeHouse	sandboxing	WebWorker	`postMessage`	high	script elements with custom type	7x to 8000x micro, 15x to 176x macro	
SafeScript	sandboxing	namespace	VDOM implementation	high?	VDOM implementation	6.43x micro, 64x macro	

functionality will also bring APIs dedicated to JavaScript sandboxing, providing purpose-built mechanisms to isolate code in a sandbox and communicate with that sandbox.

When such specialized JavaScript APIs are adopted and implemented, future JavaScript sandboxing mechanisms will no longer need to rely on repurposed functionality, making them simpler and faster.

6 In Practice – Application Examples

Previous sections discussed several JavaScript sandboxing mechanisms that work well in theory. In reality, JavaScript sandboxing solutions have apparently not seen wide-spread adoption.

The reasons for this low adoption rate are not clear. Perhaps JavaScript sandboxing has not attained enough critical mass to be "obviously" needed by web developers. Maybe web developers are waiting for a one-size-fits-all solution, are not confident enough that the JavaScript sandboxing mechanisms work as securely as advertised, or are scared away because the sandboxes are too difficult to deploy.

In this section, we highlight two technologies that have emerged from the JavaScript sandboxing research and have been used in production systems.

6.1 Facebook JavaScript

Facebook Platform launched in May 2007 [24] as a framework to allow Facebook application developers to deeply integrate with Facebook and interact with core Facebook features. Facebook application developers could use Facebook Markup Language (FBML) to customize the look and feel of their applications as rendered on Facebook. This application frontend written in FBML was hosted by Facebook itself and consisted of HTML, CSS and Facebook JavaScript (FBJS), a subset of JavaScript discussed in Sect. 3.

Facebook JavaScript allowed Facebook application developers to include JavaScript in their application, but in a controlled environment. Because applications written in FBML are hosted by Facebook, they execute inside Facebook's Web origin. If Facebook had allowed the applications to make use of a fully functional JavaScript environment, they could have easily exfiltrated Facebook session information and compromise the Facebook accounts and privacy of all users of that Facebook application. Unlike other platforms who isolate with iframes, Facebook has opted to sandbox third-party applications by rewriting HTML, CSS and JavaScript code using a middlebox located on Facebook's site.

Roughly one year after its introduction, in July 2008, FBML was used by about 33,000 applications [8] built by about 400,000 developers [22].

According to some developers, FBML was "increasingly less reliable, which leads to confusion and frustration" and "FBML always seemed like a pretty buggy and unsustainable approach to Facebook coding." Because Facebook

restricted applications to FBML, developers felt they had to stray away from standard Web coding practices.

In addition to practicality and usability issues, FBML suffered from several security problems. As discussed in Sect. 3, at least two vulnerabilities [46,47] were discovered in FBJS, which allowed attackers to break out of FBJS's JavaScript sandbox and thus escape isolation into Facebook's Web origin.

In August 2010, Facebook announced the deprecation of FBML in favor of iframe isolation for its applications [69], stating that this would eliminate the technical difference between developing an application on and off Facebook.

In December 2010, Facebook announced that new FBML applications would still be allowed until Q1 2011 because the implementation of the iframe isolation was not yet finished [79].

In January 2011, Several old and infrequently used FBML tags and API methods were eliminated [26].

In March 2011, Facebook stopped accepting new FBML applications but still allowed existing FBML applications to be updated. Switching to iframes instead of FBML was recommended [23].

In January 2012, Facebook discontinued support for FBML by no longer fixing bugs for FBML. Security and privacy-related bugs were still being addressed.

In June 2012, an "FBML Removal" migration appeared for all apps, enabled by default. This migration tool allowed application developers to disable the migration, extending their usage of FBML for another month.

In July 2012, Facebook also removed the "FBML Removal" migration tool and the FBML endpoints.

In December 2012, Facebook removed the "Static FBML Page app," which could no longer render FBML but still had the ability to display HTML, finalizing the complete removal of FBML from Facebook.

Facebook now isolates applications in iframes, requiring the webpages to be hosted outside of Facebook. The applications can make use of Facebook's Graph API to interact with the social graph.

6.2 Caja

As discussed in Sect. 3, Google's Caja rewrites HTML, CSS and JavaScript on the server-side to secure JavaScript in applications on the client-side. Caja was developed with ECMAScript 3 as a starting point. ES3 is a "very leaky language" [18] with numerous strange scoping rules, making it a nightmare to secure. The lessons learned from working on Caja were applied to the design of ECMAScript 5, making it a version of JavaScript which, as opposed to ES3, is fairly easy to secure through its "strict mode". Contained in ECMAScript 5 is a subset called Secure ECMAScript (SES), which is object-capability safe.

Caja is used, or has been used in several Google products [18,81,82] to allow embedding of third-party JavaScript: Google Labs (retired in 2011), iGoogle (retired in 2012), Orkut (retired in 2014), Google Sites, Google Apps Script, Blogspot, ...

Yahoo used Caja in its Yahoo! Application Platform [78], MySpace for its MySpace Developer Platform [44] and PayPal for PayPal Apps [43]. All three stopped using Caja (although PayPal's case is unconfirmed), but it is unclear why. A popular opinion seems to be that Caja is too restrictive for developers, who expect to be able to use full JavaScript.

Because Caja is an open-source project, it can be freely used and modified by others. Besides the very visible use-cases, Caja can also be used by less prominent websites. Apache Shindig is a container for the OpenSocial specification, which defines a component hosting environment and a set of common APIs for web-based applications. Shindig uses Caja for JavaScript rewriting, which means that less prominent web applications which make use of Shindig may also be using Caja in the background.

To this day, Caja is still actively developed. Used by Google itself and with its developers involved in workgroups on Web standards and the ECMAScript committee, the work on Caja has contributed to the development of the Web and will probably not go away anytime soon.

7 Conclusion

This work gave an overview of the JavaScript sandboxing research field and the different approaches taken to isolate and restrict JavaScript to a chosen set of resources and functionality.

The JavaScript sandboxing research can be divided into three categories: JavaScript subsets and rewriting systems, JavaScript sandboxing through browser modifications and JavaScript sandboxing without browser modifications.

JavaScript subsets and rewriting systems can restrict untrusted JavaScript if it adheres to a JavaScript subset, but a middlebox needs to verify that this is the case, possibly rewriting the code. These middleboxes break the architecture of the Web when implemented on the server-side, and put an extra burden on the user if implemented on the client-side.

Browser modifications are powerful and can sandbox JavaScript efficiently, because of their prime access to the JavaScript execution environment. Unfortunately, the software modifications are difficult to distribute and maintain in the long run unless they are adopted by mainstream browser vendors.

JavaScript sandboxing mechanisms without browser modifications leverage existing browser functionality to isolate and restrict JavaScript. This approach can be slower but requires no redistribution and maintenance of browser code. When implemented correctly, it automatically works on all modern browsers.

Acknowledgments. This work was funded by the European Community under the ProSecuToR and WebSand projects, the Swedish research agencies SSF and VR.

References

1. Galeon. http://galeon.sourceforge.net/
2. JSLint, The JavaScript Code Quality Tool. http://www.jslint.com/
3. Netscape 2.0 reviewed. http://www.antipope.org/charlie/old/journo/netscape.html
4. node.js. http://nodejs.org/
5. QuirksMode - for all your browser quirks. http://www.quirksmode.org/
6. Agten, P., Van Acker, S., Brondsema, Y., Phung, P.H., Desmet, L., Piessens, F.: JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 1–10. ACM (2012)
7. Akhawe, D., Saxena, P., Song, D.: Privilege separation in HTML5 applications. In: Kohno, T. (ed.) Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8–10, 2012, pp. 429–444. USENIX Association (2012). https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/akhawe
8. Ustinova, A.: Developers compete at Facebook conference, 23 July 2008. http://www.sfgate.com/business/article/Developers-compete-at-Facebook-conference-3203144.php
9. Apache OpenOffice: Writing Office Scripts in JavaScript. https://www.openoffice.org/framework/scripting/release-0.2/javascript-devguide.html
10. Barth, A., Jackson, C., Mitchell, J.C.: Securing frame communication in browsers. Commun. ACM **52**(6), 83–91 (2009). http://doi.acm.org/10.1145/1516046.1516066
11. Blink: Blink. http://www.chromium.org/blink
12. BuiltWith: jQuery Usage Statistics. http://trends.builtwith.com/javascript/jQuery
13. Cao, Y., Li, Z., Rastogi, V., Chen, Y., Wen, X.: Virtual browser: a virtualized browser to sandbox third-party JavaScripts with enhanced security. In: Youm, H.Y., Won, Y. (eds.) 7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS 2012, Seoul, Korea, May 2–4, 2012, pp. 8–9. ACM (2012). http://doi.acm.org/10.1145/2414456.2414460
14. Cassou, D., Ducasse, S., Petton, N.: SafeJS: Hermetic Sandboxing for JavaScript (2013)
15. Charles Severance: JavaScript: Designing a Language in 10 Days. http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.html
16. Crockford, D.: ADsafe - making JavaScript safe for advertising. http://adsafe.org/
17. De Ryck, P., Desmet, L., Philippaerts, P., Piessens, F.: A security analysis of next generation web standards. Technical report. In: Hogben, G., Dekker, M. (eds.) European Network and Information Security Agency (ENISA), July 2011. https://lirias.kuleuven.be/handle/123456789/317385
18. Dio Synodinos: ECMAScript 5, Caja and Retrofitting Security, with Mark S. Miller. http://www.infoq.com/interviews/ecmascript-5-caja-retrofitting-security
19. Dong, X., Tran, M., Liang, Z., Jiang, X.: AdSentry: comprehensive and flexible confinement of javascript-based advertisements. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC 2011, pp. 297–306. ACM, New York (2011). http://doi.acm.org/10.1145/2076732.2076774
20. ECMAScript: Harmony Direct Proxies. http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies

21. Espruino: Espruino - JavaScript for Microcontrollers. http://www.espruino.com/
22. Facebook: Facebook Expands Power of Platform Across the Web and Around the World, 23 July 2008. http://newsroom.fb.com/news/2008/07/facebook-expands-power-of-platform-across-the-web-and-around-the-world/
23. Facebook: Facebook Platform Migrations (Older). https://developers.facebook.com/docs/apps/migrations/completed-changes
24. Facebook: Facebook Unveils Platform for Developers of Social Applications, 24 May 2007. http://newsroom.fb.com/news/2007/05/facebook-unveils-platform-for-developers-of-social-applications/
25. Finifter, M., Weinberger, J., Barth, A.: Preventing capability leaks in secure javascript subsets. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010. The Internet Society (2010). http://www.isoc.org/isoc/conferences/ndss/10/pdf/21.pdf
26. Fran Larkin: Platform Updates: Change Log, Third Party IDs and More, 18 December 2010. https://developers.facebook.com/blog/post/441
27. GNOME: Gjs: JavaScript Bindings for GNOME. https://wiki.gnome.org/action/show/Projects/Gjs?action=show&redirect=Gjs
28. Google: V8 JavaScript Engine. https://code.google.com/p/v8/
29. Google Chrome Developers: Chrome - What are extensions? https://developer.chrome.com/extensions
30. Google Chrome Developers: Native Client. https://developer.chrome.com/native-client
31. Grosskurth, A., Godfrey, M.W.: A case study in architectural analysis: The evolution of the modern web browser. EMSE (2007)
32. Guarnieri, S., Livshits, V.B.: GATEKEEPER: mostly static enforcement of security and reliability policies for javascript code. In: Monrose, F. (ed.) 18th USENIX Security Symposium, Montreal, Canada, August 10–14, 2009, Proceedings, pp. 151–168. USENIX Association (2009). http://www.usenix.org/events/sec09/tech/full_papers/guarnieri.pdf
33. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of javascript. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 126–150. Springer, Heidelberg (2010). http://dx.doi.org/10.1007/978-3-642-14107-2_7
34. Heiderich, M., Frosch, T., Holz, T.: IceShield: detection and mitigation of malicious websites with a frozen DOM. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 281–300. Springer, Heidelberg (2011). http://dx.doi.org/10.1007/978-3-642-23644-0_15
35. Ingram, L., Walfish, M.: Treehouse: javascript sandboxes to help web developers help themselves. In: Heiser, G., Hsieh, W.C. (eds.) 2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13–15, 2012, pp. 153–164. USENIX Association (2012). https://www.usenix.org/conference/atc12/technical-sessions/presentation/ingram
36. Jacaranda: Jacaranda. http://jacaranda.org
37. Jayaraman, K., Du, W., Rajagopalan, B., Chapin, S.J.: ESCUDO: a fine-grained protection model for web browsers. In: 2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21–25, 2010, pp. 231–240. IEEE Computer Society (2010). http://doi.ieeecomputersociety.org/10.1109/ICDCS.2010.71

38. Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: WWW 2007: Proceedings of the 16th International Conference on World Wide Web, pp. 601–610. ACM, New York (2007). http://dx.doi.org/10.1145/1242572.1242654
39. Joiner, R., Reps, T.W., Jha, S., Dhawan, M., Ganapathy, V.: Efficient runtime-enforcement techniques for policy weaving. In: Cheung, S., Orso, A., Storey, M.D. (eds.) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16–22, 2014, pp. 224–234. ACM (2014). http://doi.acm.org/10.1145/2635868.2635907
40. jQuery: Update on jQuery.com Compromises. http://blog.jquery.com/2014/09/24/update-on-jquery-com-compromises/
41. JSLint Error Explanations: Implied eval is evil. Pass a function instead of a string. http://jslinterrors.com/implied-eval-is-evil-pass-a-function-instead-of-a-string
42. Zyp, K.: Secure Mashups with dojox.secure. http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/
43. Dignan, L.: Developing a PayPal App, 20 February 2011. https://web.archive.org/web/20110220013816/https://www.x.com/docs/DOC-3082
44. Dignan, L.: MySpace: Caja JavaScript scrubbing ready for prime time. http://www.zdnet.com/article/myspace-caja-javascript-scrubbing-ready-for-prime-time/
45. Luo, T., Du, W.: Contego: capability-based access control for web browsers - (short paper). In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) Trust 2011. LNCS, vol. 6740, pp. 231–238. Springer, Heidelberg (2011). http://dx.doi.org/10.1007/978-3-642-21599-5_17
46. Maffeis, S., Mitchell, J.C., Taly, A.: Isolating javascript with filters, rewriting, and wrappers. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 505–522. Springer, Heidelberg (2009). http://dx.doi.org/10.1007/978-3-642-04444-1_31
47. Maffeis, S., Taly, A.: Language-based isolation of untrusted javascript. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8–10, 2009, pp. 77–91. IEEE Computer Society (2009). http://doi.ieeecomputersociety.org/10.1109/CSF.2009.11
48. Magazinius, J., Phung, P.H., Sands, D.: Safe wrappers and sane policies for self protecting javascript. In: Aura, T., Järvinen, K., Nyberg, K. (eds.) NordSec 2010. LNCS, vol. 7127, pp. 239–255. Springer, Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-27937-9_17
49. Maxthon: Maxthon Cloud Browser. http://www.maxthon.com/
50. Meyerovich, L.A., Felt, A.P., Miller, M.S.: Object views: fine-grained sharing in browsers (2010). http://doi.acm.org/10.1145/1772690.1772764
51. Meyerovich, L.A., Livshits, V.B.: ConScript: specifying and enforcing fine-grained security policies for javascript in the browser. In: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16–19 May 2010, Berleley/Oakland, California, USA, pp. 481–496. IEEE Computer Society (2010). http://doi.ieeecomputersociety.org/10.1109/SP.2010.36
52. Mickens, J.: Pivot: fast, synchronous mashup isolation using generator chains. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18–21, 2014. pp. 261–275. IEEE Computer Society (2014). http://dx.doi.org/10.1109/SP.2014.24
53. Mickens, J., Finifter, M.: Jigsaw: rfficient, low-effort mashup isolation. In: Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 2012), pp. 13–25. USENIX, Boston (2012). https://www.usenix.org/conference/webapps12/technical-sessions/presentation/mickens

54. Microsoft: Internet Explorer Architecture. http://msdn.microsoft.com/en-us/library/aa741312(v=vs.85).aspx
55. Microsoft: Microsoft Internet Security and Acceleration (ISA) Server 2004. http://technet.microsoft.com/en-us/library/cc302436.aspx
56. Microsoft: Microsoft Security Bulletin MS04-040 - Critical. https://technet.microsoft.com/en-us/library/security/ms04-040.aspx
57. Microsoft: Mitigating Cross-site Scripting With HTTP-only Cookies. http://msdn.microsoft.com/en-us/library/ms533046(VS.85).aspx
58. Microsoft Live Labs: Live Labs Websandbox. http://websandbox.org
59. Mihai Bazon: UglifyJS. https://github.com/mishoo/UglifyJS/
60. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008
61. Miller, M.S.: Robust composition: towards a unified approach to access control and concurrency control. Ph.D. thesis, Johns Hopkins University, Baltimore, MD, USA (2006). aAI3245526
62. MITRE: CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition. http://cwe.mitre.org/data/definitions/367.html
63. MongoDB, Inc.: MongoDB. http://www.mongodb.org/
64. Mozilla: Gecko. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko
65. Mozilla: JavaScript Strict Mode Reference. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode
66. Mozilla: MDN - Building an extension. https://developer.mozilla.org/en/docs/Building_an_Extension
67. Mozilla The Narcissus meta-circular JavaScript interpreter. https://github.com/mozilla/narcissus
68. Mozilla: The "with" statement. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with
69. Namita Gupta: Facebook Platform Roadmap Update, 19 August 2010. https://developers.facebook.com/blog/post/402
70. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You are what you include: large-scale evaluation of remote JavaScript inclusions. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) the ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, October 16–18, 2012, pp. 736–747. ACM (2012). http://doi.acm.org/10.1145/2382196.2382274
71. Opera: Opera Browser. http://www.opera.com
72. Patil, K., Dong, X., Li, X., Liang, Z., Jiang, X.: Towards fine-grained access control in javascript contexts. In: 2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20–24, 2011, pp. 720–729. IEEE Computer Society (2011). http://dx.doi.org/10.1109/ICDCS.2011.87
73. Phung, P.H., Desmet, L.: A two-tier sandbox architecture for untrusted JavaScript. In: JSTools 2012, Proceedings of the Workshop on JavaScript Tools, Beijing, 13 June 2012, pp. 1–10 (2012)
74. Phung, P.H., Sands, D., Chudnov, A.: Lightweight self-protecting JavaScript. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, ASIACCS 2009, pp. 47–60. ACM, New York (2009). http://doi.acm.org/10.1145/1533057.1533067
75. Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S.: ADsafety: type-based verification of javascript sandboxing. In: 20th USENIX Security Symposium, San Francisco, CA, USA, August 8–12, 2011, Proceedings. USENIX Association (2011). http://static.usenix.org/events/sec11/tech/full_papers/Politz.pdf

76. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: BrowserShield: vulnerability-driven filtering of dynamic HTML. In: OSDI 2006: Proceedings of the 7th symposium on Operating Systems Design and Implementation, pp. 61–74. USENIX Association, Berkeley (2006). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.1661
77. Richards, G., Hammer, C., Burg, B., Vitek, J.: The eval that men do: large-scale study of the use of eval in javascript applications. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 52–78. Springer, Heidelberg (2011). http://dx.doi.org/10.1007/978-3-642-22655-7_4
78. Sam Pullara: Introducing Y!OS 1.0 - live today! 28 October 2008. https://web.archive.org/web/20081029191209/http://developer.yahoo.net/blog/archives/2008/10/yos_10_launch.html
79. Sandra Liu Huang: Platform Updates: Promotion Policies, Facepile and More, 4 December 2010. https://developers.facebook.com/blog/post/2010/12/03/platform-updates--promotion-policies--facepile-and-more/
80. Mozilla SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey
81. Stack Exchange (Jasvir Nagra): Why hasn't Caja been popular? http://programmers.stackexchange.com/a/147014
82. Stack Overflow (Kevin Reid): Uses of Google Caja. http://stackoverflow.com/questions/16054597/uses-of-google-caja
83. Taly, A., Erlingsson, U., Mitchell, J.C., Miller, M.S., Nagra, J.: Automated analysis of security-critical javascript APIs. In: IEEE Symposium on Security and Privacy, pp. 363–378 (2011)
84. Ter Louw, M., Ganesh, K.T., Venkatakrishnan, V.N.: Adjail: practical enforcement of confidentiality and integrity policies on web advertisements. In: 19th USENIX Security Symposium, Washington, DC, USA, August 11–13, 2010, Proceedings, pp. 371–388. USENIX Association (2010). http://www.usenix.org/events/sec10/tech/full_papers/TerLouw.pdf
85. Ter Louw, M., Phung, P.H., Krishnamurti, R., Venkatakrishnan, V.N.: SafeScript: javascript transformation for policy enforcement. In: Riis Nielson, H., Gollmann, D. (eds.) NordSec 2013. LNCS, vol. 8208, pp. 67–83. Springer, Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-41488-6_5
86. Ter Louw, M., Venkatakrishnan, V.N.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers (2009). http://dx.doi.org/10.1109/SP.2009.33
87. Tessel: Tessel 2. https://tessel.io
88. The FaceBook Team: FBJS. http://wiki.developers.facebook.com/index.php/FBJS
89. Troy Hunt: How I got XSS'd by my ad network. http://www.troyhunt.com/2015/07/how-i-got-xssd-by-my-ad-network.html
90. Twitter: How to embed Twitter timelines on your website. https://blog.twitter.com/2012/embedded-timelines-howto
91. Van Acker, S., De Ryck, P., Desmet, L., Piessens, F., Joosen, W.: WebJail: least-privilege integration of third-party components in web mashups. In: Zakon, R.H., McDermott, J.P., Locasto, M.E. (eds.) Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5–9 December 2011, pp. 307–316. ACM (2011). http://doi.acm.org/10.1145/2076732.2076775
92. W3C: Same Origin Policy - Web Security. http://www.w3.org/Security/wiki/Same_Origin_Policy
93. W3C: W3C - Web Workers. http://www.w3.org/TR/workers/

94. W3C: W3C Standards and drafts - Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/
95. W3C: XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/
96. W3Techs: Usage of JavaScript for websites. http://w3techs.com/technologies/details/cp-javascript/all/all
97. Webkit Blog - David Carson: Android uses WebKit. https://www.webkit.org/blog/142/android-uses-webkit/
98. WHATWG: HTML Living Standard - Timers. https://html.spec.whatwg.org/multipage/webappapis.html#timers
99. Yu, D., Chander, A., Islam, N., Serikov, I.: JavaScript instrumentation for browser security. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, pp. 237–249. ACM, New York (2007). http://doi.acm.org/10.1145/1190216.1190252

http://www.springer.com/978-3-319-43004-1

Foundations of Security Analysis and Design VIII
FOSAD 2014/2015/2016 Tutorial Lectures
Aldini, A.; Lopez, J.; Martinelli, F. (Eds.)
2016, VII, 163 p. 36 illus., Softcover
ISBN: 978-3-319-43004-1