SafeJS: Hermetic Sandboxing for JavaScript

Damien Cassou, Stéphane Ducasse, Nicolas Petton

SafeJS: Hermetic Sandboxing for JavaScript

Damien Cassou Stéphane Ducasse Nicolas Petton

Abstract

Isolating programs is an important mechanism to support more secure applications. Isolating program in dynamic languages such as JavaScript is even more challenging since reflective operations can circumvent simple mechanisms that could protect program parts. In this article we present SafeJS, an approach and implementation that offers isolation based on separate sandboxes and control of information exchanged between them.

In SafeJS, sandboxes based on web workers do not share any data. Data exchanged between sandboxes is solely based on strings. Using different policies, this infrastructure supports the isolation of the different scripts that usually populate web pages. A foreign component cannot modify the main DOM tree in unexpected manner.

Our SafeJS implementation is currently being used in an industrial setting in the context of the Resilience FUI 12 project.

1 Introduction

A mashup is a web page aggregating multiple sources of information. In a JavaScript-based mashup, such a source of information often requires including an external script in the hosting web page for data retrieval or UI building. Including a JavaScript script from an external source into a web page raises security concerns, as well as unresolved questions about secure communication between the script and the hosting web page.

JavaScript semantics is permissive with respects to security [MT09,GSK10,RHBV11]. Several attempts to isolate JavaScript programs have been made : Caja [MSL+08], FBJS[1], AdSafe [PEGK11]. These solutions either use a subset of JavaScript or rely on code rewriting.

Our approach is based on separate thread-based local sandboxes and controlled policies to access the main environment. On this aspect, the idea is close to the one of Software Isolation Processes (SIP) of Singularity OS [HL07]. SafeJS is available for download under a free open-source license.[2]

The contributions of this technical report are:

- strong isolation of JavaScript based on web workers ;
- fully working implementation used in an industrial setting.

[1] a subset of JavaScript for Facebook
[2] https://gitorious.org/safe-js/safe-js

2 DOM and Web Workers

This section presents some web standards (DOM and web workers) that are key to SafeJS design and implementation. The DOM is the document tree of any XML and HTML document. The DOM is virtually used by all web applications. The web worker is an infrastructure to run isolated processes in a web page. The web worker draft specification is already implemented by modern web browsers.

DOM. The W3C defines the *DOM* (Document Object Model) as "*an application programming interface (API) for HTML and XML documents*" [ABC+98]. The DOM specification consists of 3 stacked levels, each level building on the level below. The DOM Level 1, provides an API to let developers add, modify and delete elements from an HTML (in fact any XML) document. The DOM represents any HTML document like a tree of node objects, each object representing a particular tag of the document. In its Level 2, the DOM additionally specifies how developers can manipulate other parts of HTML documents such as CSS and mouse events. The DOM Level 3 adds other APIs such as a keyboard event handling API and an XML serialization API. JavaScript applications manipulate the DOM (*e.g.,* adding and removing nodes) to impact the visual representation of the web page.

Web Workers. The W3C is in the process of specifying a *worker* API that "*allows Web application authors to spawn background workers running scripts in parallel to their main page*"[3]. This draft specification enforces that the code executing within a worker is completely isolated from the main thread and other workers: a worker can only communicate through string-based messages, *i.e.,* a worker has no access to the DOM API and thus can not modify the web document. In the following, we will talk about the "worker specification" even though this document is still in a draft stage and subject to change. Still, this document is sufficiently advanced to have compatible implementations in all modern web browsers.

The idea behind SafeJS is to run each unsafe script inside its own worker and give this worker a virtual DOM for the script to play with. This virtual DOM is compatible with the standard DOM but restricts which changes impact the document.

3 SafeJS runtime overview

Figure 1 summarizes the high-level architecture of SafeJS. A standard web browser provides a view of the web document, a main JavaScript thread, and a DOM (accessible from the main JavaScript thread). The web developer must include the SafeJS library inside the document and pass each unsafe script he wants to embed in the document to SafeJS. As soon as an unsafe script is passed to SafeJS, SafeJS creates a dedicated JavaScript worker (step 1.). This worker first creates a SafeJS Virtual DOM (known as the SafeJS VDOM) mimicking the main JavaScript thread's DOM (step 2.). The worker then loads and starts the unsafe script (step 3.). The unsafe script can freely

[3] http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

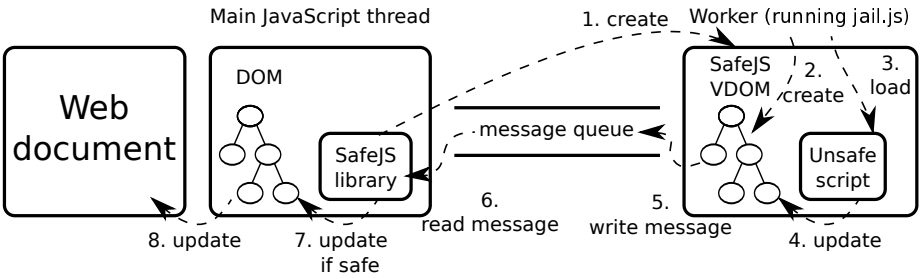

Figure 1: High-level architecture of SafeJS

read from the SafeJS VDOM. The unsafe script can also update the SafeJS VDOM (step 4.). When this happens, the SafeJS VDOM writes a message to the message queue (step 5.). The SafeJS library then reads the message at the other end of the queue (step 6.). When SafeJS detects a forbidden update, SafeJS will either ignore the request or kill the worker and its malicious script; this choice depends on the SafeJS configuration. If the update is authorized, SafeJS performs the equivalent update on the real DOM (step 7.). The web browser finally updates the web page (step 8.).

We now detail each step.

Step 0. – Initializing SafeJS

The web developer must first load the SafeJS library through RequireJS:[4]

```
<script type='text/javascript' data-main="safe" src='js/require.js' />
```

The data-main declaration instructs RequireJS to load SafeJS. The next step for the web developer is to pass each unsafe script to SafeJS:

```
<script type='text/safe-javascript' src='js/spy.js' node='#unsafeDiv' policy='read-write' />
```

This declaration instructs SafeJS to load the unsafe js/spy.js script. This script can be any JavaScript code compatible with client-side usage. The type parameter value must be set to text/safe-javascript: this prevents the web browser from loading the script in the main thread and this indicates to SafeJS that the script must be sandboxed. The node parameter indicates the HTML target node which will be made accessible from the unsafe script: in this case, the HTML node with the unsafeDiv id. This target node will become the only node visible from the spy.js unsafe script. The policy parameter indicates the kind of operations the unsafe script can apply on the target node. A read-only policy only grants reading capabilities of the target node to the unsafe script whereas a read-write policy grants full privileges to this node. Finally, the web developer must include the HTML target node that the unsafe script will read and update:

```
<div id="unsafeDiv"></div>
```

[4] http://requirejs.org/

Step 1. – Creating the worker

When SafeJS is loaded, it navigates the DOM to find any script declaration with the text/safe-javascript type and creates a dedicated worker for each. Each worker is initialized with a SafeJS jail.js script that waits for its configuration by reading its message queue. No other form of communication is allowed by the worker draft specification between the main thread and the worker.

Step 2. – Creating the SafeJS VDOM

The message queue only accepts strings. SafeJS must then serialize each message going through the queue: we use JSON[5] as the serialization format. The first message SafeJS sends to the worker is a copy of the HTML target node, unsafeDiv in our example. With this message, the worker creates a SafeJS VDOM that mimics a real DOM but that can not directly impact the web document.

Step 3. – Loading the unsafe script

After the SafeJS VDOM is created, SafeJS sends the URL of the unsafe script to load (js/spy.js in our example) to the worker through the message queue. When the worker receives this message, it loads the unsafe script.

Step 4. – Reading from and updating the SafeJS VDOM

The unsafe script can use the SafeJS VDOM as if it was a real DOM. Because of this, our approach allows any script to be embedded in a web page with no modification. For the unsafe script, reading and updating the SafeJS VDOM is done using the standard JavaScript API. Because the SafeJS VDOM contains a (partial) copy of the web page, reading is done without any communication with the main thread.

Step 5. – Writing to the message queue

When the unsafe script updates the SafeJS VDOM, the worker sends a serialization of the change to the message queue. This change information contains the state of the SafeJS VDOM as the unsafe script would like it to be.

Step 6. – Reading from the message queue

Then, SafeJS deserializes the change and checks the policy to verify whether the change is allowed. If the change is not allowed, SafeJS can either kill the worker or ignore the update request depending on its configuration.

Step 7. – Safely updating the DOM

If the change is allowed, SafeJS updates the real DOM. SafeJS then sends a (partial) copy of the DOM to the worker which then updates its SafeJS VDOM.

[5] http://json.org/

Step 8. – Updating the web document

When SafeJS updates the real DOM, the web browser takes care of updating the web document.

4 Architecture of the SafeJS VDOM

The SafeJS VDOM is a full-JavaScript implementation of the DOM [ABC+98] level 3. SafeJS VDOM is actually a fork of the well-known jsdom DOM implementation[6] that is used with Node.js by server-side JavaScript applications.[7] Our SafeJS VDOM implementation is available under a free open-source license.[8]

In this section, we present some of the key differences between SafeJS VDOM and jsdom and explain why they are critical for SafeJS.

Replace the dependency manager

The jsdom library depends on multiple other JavaScript libraries such as htmlparser2 and contextify. These dependent libraries are resolved and loaded with a specific instruction that blocks the execution until the library is ready to be used:

```
defaultParser = require(htmlparser2)
... code that uses 'defaultParser'...
```

If this blocking code is perfectly valid for a server-side JavaScript application, a web browser only allows non-blocking code and there is no way to force the browser to wait for anything: the web browser would kill any script with an active waiting loop for example. We thus had to rewrite all similar code to use an equivalent and non-blocking function using RequireJS to resolve and load external libraries:

```
define(["htmlparser2"], function(defaultParser)
  ... code that uses 'defaultParser'...
)
```

Remove accesses to the file-system jsdom features the ability to read and write files on the hosting file system, thanks to a dedicated API provided by node.js. Such unrestricted access to a file system is not possible in a web browser so we removed this ability in SafeJS VDOM.

Remove references to the global object jsdom uses Contextify, a C++ library, to bind the pseudo variable this to its window object. In SafeJS VDOM, we can not depend on a C++ library and we thus had to create a global object that mimics the behavior of window. Because window is, for example, responsible for providing a logger object (known as the console), the SafeJS VDOM global object also provides a virtual logger object.

[6] https://github.com/tmpvar/jsdom
[7] http://nodejs.org/
[8] https://gitorious.org/jsdom-client/jsdom-client

Recursively apply these changes to jsdom dependencies Finally, we applied the above-mentioned changes to the jsdom dependencies we had to integrate in SafeJS VDOM (*e.g.,* cssom, cssstyle and htmlparser). For this, we leveraged the browserify tool[9] to

5 Conclusion

In this document we reported on the design and implementation of SafeJS, an hermetic sandboxing mechanism for JavaScript. SafeJS allows mashup JavaScript developers to include external (potentially unsafe) scripts into a web page without risking data leaks from the web page to the external scripts. Our approach is based on web workers, a draft specification to run isolated processes in a web page.

The idea behind SafeJS is to run each unsafe script inside its own worker and give this worker a virtual DOM for the script to play with. This virtual DOM is compatible with the standard DOM but restricts which changes impact the document.

Our SafeJS implementation is currently being used in an industrial setting in the context of the Resilience FUI 12 project.

References

[ABC+98] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, and Lauren Wood. *Document Object Model Specification DOM Level 1 version 1.0*. World Wide Web Consortium, 1998.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[HL07] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[MSL+08] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja safe active content in sanitized javascript. Technical report, Google Inc., 2008.

[MT09] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted javascript. In *In CSF*, 2009.

[PEGK11] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[9] https://github.com/substack/node-browserify

[RHBV11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of Ecoop 2011*, 2011.