JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications

Pieter Agten†, Steven Van Acker†, Yoran Brondsema†, Phu H. Phung‡,
Lieven Desmet†, Frank Piessens†
{pieter.agten,steven.vanacker}@cs.kuleuven.be

†IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium
‡ProSec Security Group, Chalmers University of Technology, Gothenburg, Sweden

ABSTRACT

The inclusion of third-party scripts in web pages is a common practice. A recent study has shown that more than half of the Alexa top 10 000 sites include scripts from more than 5 different origins. However, such script inclusions carry risks, as the included scripts operate with the privileges of the including website.

We propose JSand, a server-driven but client-side JavaScript sandboxing framework. JSand requires *no browser modifications*: the sandboxing framework is implemented in JavaScript and is delivered to the browser by the websites that use it. Enforcement is done *entirely at the client side*: JSand enforces a server-specified policy on included scripts without requiring server-side filtering or rewriting of scripts. Most importantly, JSand is *complete*: access to all resources is mediated by the sandbox.

We describe the design and implementation of JSand, and we show that it is secure, backwards compatible, and that it performs sufficiently well.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: Web-based services

Keywords

Web Application Security, Web Mashups, Script Inclusion, Sandbox, Security Architecture

1. INTRODUCTION

In the last decade, the web platform has become the number one platform on the Internet. There is a clear paradigm shift from desktop applications and proprietary client-server solutions towards web-enabled services. An important catalyst for this paradigm shift has been the power of JavaScript as well as the advent of HTML5, giving web developers the tools to build rich and interactive websites.

As a consequence of this enormous growth in popularity, the web has also become the primary attack platform: SANS [30] reported in 2009 that more than 60% of all cyber attacks are aimed at web applications, and more than 80% of discovered vulnerabilities are web-related. A whole range of web attacks exists in the wild, ranging from Cross-Site Scripting, Cross-Site Request Forgery and SQL injection to the exploitation of broken authorization and session management. This paper focuses on one particular and important class of web attacks, namely attacks due to the insecure integration of JavaScript.

To enrich the functionality and interactivity of a website, a common and wide-spread approach is to integrate JavaScript from third-party script providers. Recent studies [40, 23] have shown that 96.9% of websites include scripts from external sources, and on average each website includes scripts from 3.1 external sources. For example, websites integrate among others JavaScript-enabled advertisements (such as Google AdSense and adBrite), Web analytics frameworks (such as Google Analytics, Yahoo! Web Analytics and Tynt), web widgets and buttons (such as Google Maps, addToAny button and Google +1 button), and JavaScript programming libraries (such as jQuery and Dojo).

The de facto browser security model today is defined by the Same-Origin Policy (SOP). The SOP restricts access of client-side scripts to resources belonging to the same origin[1]. For instance, the SOP ensures that document data and cookies from one origin cannot be read by scripts belonging to another origin. However, the SOP includes some important relaxations with respect to navigation and content inclusion (e.g. embedded images and scripts) [41]. In particular, if a page from one origin includes a script from another origin, the included script is treated as if it belongs to the including origin, and hence it inherits all the capabilities and permissions of the hosting page. This makes malicious script inclusion a very powerful attack vector.

Several countermeasures have been proposed to limit the capabilities of third-party JavaScript, including (1) the introduction of safe subsets of JavaScript [33, 3, 15], (2) client-side reference monitors [17, 34], and (3) server-side transformations of the script to be included [22, 32]. However, all of these have at least one of the following limitations.

First, some approaches [17, 34] require intrusive *browser modifications*, in particular to the JavaScript engine and the

[1]An origin is a (protocol, domain name, port) tuple.

binding between browser and JavaScript engine. Such modifications hinder short-term deployment of the countermeasure.

Second, some approaches do *not support client-side script inclusion*: in order to perform server-side pre-processing (e.g. source-to-source translation or filtering) of the scripts, the scripts have to pass through the web server [22, 33, 3]. This effectively changes the architectural model of client-side script inclusion to server-side script inclusion.

Third, some approaches do *not provide complete mediation* between different scripts on the same page, or to all resources exposed in the browser. Self-Protecting JavaScript (SPJS) [26, 16] assumes that all scripts included on a hosting page need identical security constraints. It does not differentiate between different external scripts nor between local and remote inclusions. AdJail [32] successfully isolates untrusted advertisements from the Document Object Model (DOM) of the hosting page, but since it uses iframes as isolation units, it cannot fully protect security-sensitive APIs such as XHR, Geolocation and local storage.

Inspired by recent advances in achieving object-capability guarantees for JavaScript [22, 14, 20, 25, 8], this paper presents JSand, a novel security architecture to securely integrate third-party JavaScript. We improve upon the state-of-the-art with the following contributions:

1. JSand is the first JavaScript sandbox that (1) does not need browser modifications, (2) supports client-side script inclusion and (3) completely mediates different scripts and the browser APIs.
2. We show evidence that JSand is secure, compatible with complex and widely used scripts (such as Google Maps, Google Analytics and jQuery) and performs sufficiently well.

The rest of this paper is structured as follows. Section 2 introduces the necessary background and defines the problem statement. In Section 3, the JSand architecture is presented, and Section 4 discusses several relevant implementation aspects. Section 5 evaluates the security, compatibility and performance of JSand. Finally, Section 6 discusses related work, and we conclude in Section 7.

2. PROBLEM STATEMENT

2.1 Integrating third-party JavaScript

To enrich the functionality and interactivity of a website, a common and wide-spread approach is to integrate JavaScript from third-party script providers. The two most wide-spread techniques to integrate third-party JavaScript in web pages are through script inclusion and via iframe integration [6].

Script inclusion HTML script tags are used to execute JavaScript as part of a web page. If the JavaScript code is integrated from an external source, the browser will still execute the code within the security context of the web page, without any restrictions of the SOP.

Iframe integration HTML iframe tags allow a web developer to include one document inside another. The advantage of iframe integration is that the integrated document is loaded in its own security context: integrated content from another origin is isolated from the integrating web page by the SOP.

Script inclusion is the de facto script integration technique on the web, both for local scripts as well as for external scripts. The iframe integration technique is used for web gadgets that don't have strong integration needs with the embedding web page, or have an out-of-band service-to-service communication channel (such as the Facebook Like button or Facebook Apps). In the remainder of this paper, we focus on third-party JavaScript integration through script inclusion.

2.2 Malicious script inclusion

The browser security model for integrating third-party JavaScript is problematic. Once included in a website, a malicious script cannot only access all the document data and cookies, but with the advent of HTML5, the malicious script has also access to local storage data (e.g. Web Storage, IndexedDB), intra-window communication (Web Messaging), remote resource fetching via XHR and user-consented privileges (such as Geolocation, media capture, access to System Information API, and many more). This makes malicious script inclusion a very powerful attack vector. One can distinguish between two types of attackers.

Malicious script provider The script provider has malicious intentions (but covers up by providing appealing functionality to potential customers), or becomes malicious over time (e.g. intentionally, or by selling out or quitting his business [23]).

Benign script provider under attack The script provider has no malicious intentions, but the scripts delivered to its clients become under control of an attacker. This can be due to the inclusion of other untrusted resources (e.g. in advertisement networks), due to a bug in the delivered script (e.g. a DOM-based XSS vulnerability [12]), due to a server-side take-over (e.g. via SQL injection) or due to in-transit tampering with the scripts by a network attacker.

In both cases, the attacker controls the scripts included by the hosting page, and by default gains full access to the execution environment of the web page.

2.3 Requirements

Given the wide spread of script inclusion and the increasing impact of malicious script inclusion, there is a clear need for a novel security architecture to **securely integrate third-party JavaScript**, but **without introducing disruptive changes**. Preserving backwards compatibility is crucial in the web context. We therefore identify the following requirements:

R1 Complete mediation All access to security-sensitive functionality should be completely mediated by the security mechanism. This includes access to the DOM, as well as security-sensitive JavaScript APIs (such as Geolocation and local storage). The attacker must be unable to circumvent the security mechanisms in place.

R2 Backwards compatible The security mechanism should seamlessly operate in the current web ecosystem, i.e. it should not rely on browser modifications or disable the direct delivery of scripts from the script provider to the browser. In addition, the security mechanism should support the integration of legacy scripts.

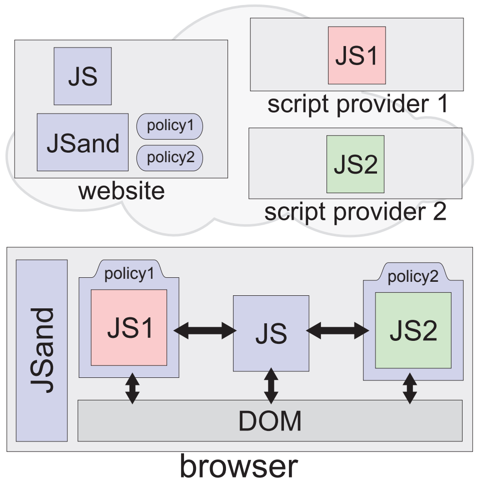

Figure 1: the JSand architecture. Inside the browser, all access from JSand sandboxes to the JavaScript environment is mediated according to server-supplied policies.

R3 Performance The security mechanism should introduce only a minimal performance penalty, unnoticeable to the end-user.

3. JSAND SECURITY ARCHITECTURE

The JSand architecture enables the owner of a website to securely integrate third-party scripts, without needing disruptive changes to either server-side or client-side infrastructure. We first give a high-level overview of the architecture and then discuss the architectural choices under the hood.

3.1 Architectural overview

Figure 1 depicts the JSand architecture. A website owner deploys JSand by including the JSand JavaScript library in his webpages. When one of these pages is loaded in a visitor's browser, the third-party scripts to be sandboxed are fetched directly from the servers of the script provider. The JSand library confines each script to its own secure sandbox, which isolates the script from other scripts and from the DOM.

3.2 Under the hood

The JSand architecture is based upon the secure confinement of third-party JavaScript. JSand realizes this through the use of an object-capability environment. Such environment provides an appropriate device for isolating untrusted JavaScript: without an explicit and unforgeable reference to a security-sensitive resource (i.e. an object or a function), a script is unable to access the resource or make use of its capabilities. The object-capability model is at the basis of Caja [22], and many other safe subsets of JavaScript [14].

The JSand library invokes third-party scripts, initially giving them only a minimal set of unforgeable references. To maintain control over all references acquired by a sandboxed script, JSand applies the Membrane pattern proposed by Miller [21]. Our implementation of this pattern consists of placing policy-enforcing wrappers around objects that provide potentially security-sensitive operations. Whenever one of these objects returns a reference to another object, the membrane is extended to cover that object as well. This ensures a sandboxed script never has direct access to a security-sensitive operation.

The membrane's wrappers intercept all operations performed on the objects they wrap and hence implement the security policy enforcement points. On each enforcement point, the wrapper consults the security policy to determine whether or not the corresponding operation is permitted. If not, this will be indicated by the security policy and the operation will be blocked. The architecture is not bound to any specific type of security policy, which gives website owners the freedom to enforce arbitrarily complex policies.

Since all interactions between a script and the browser are performed by calling DOM methods, it suffices to place a wrapper around each DOM object in order to enforce a policy on all security-sensitive operations. These include not only operations to read or modify content of the hosting page, but also to communicate with other scripts and to use browser-provided JavaScript APIs.

In conclusion, the JSand architecture provides an end-to-end solution for securely integrating third-party JavaScript scripts on a website. The website owner is able to define and enforce security policies on scripts, which puts him back into the driver's seat. JSand does not require disruptive changes to the architecture of the web: it does not break direct script delivery towards the browser, and can be deployed without additional server-side or client-side infrastructure. The combination of the object-capability model and the Membrane pattern ensures that all access from a sandboxed script to security-sensitive operations passes through a membrane's wrappers, which enforce the security policy.

4. PROTOTYPE IMPLEMENTATION

This section reports on the development of a mature JSand prototype, which is designed to work in ECMAScript 5 (ES5) compatible browsers with support for the proxy features of the upcoming ES Harmony standard. The current prototype runs seamlessly in Google Chrome v20.0.1132.21.

In Subsections 4.1 and 4.2, we present the client-side technology for executing third-party JavaScript in a confined sandbox. Subsection 4.3 describes the type of security policies that are enforced. Next, Subsection 4.4 illustrates how access to security-sensitive operations is completely mediated. Subsection 4.5 discusses how our prototype deals with dynamic script loading and Subsection 4.6 describes a set of automatic script transformations to improve compatibility with legacy scripts.

4.1 Object-capability system

As described in Section 3.2, the JSand architecture relies on an object-capability environment to provide complete mediation. The ECMAScript language does not qualify as an object-capability language by itself. For instance, any script has access to all global variables by default, and consequently has capabilities that are not under control of any security framework. However, in 2008, Miller et al. [22] identified a subset of ES3 which forms a true object-capability language. More recently, the Google Caja team has identified a subset of ES5 strict, named Secure ECMAScript (SES), that also provides such an object-capability language. Moreover, they have developed a JavaScript library that enables the execution of SES on ES5-compatible browsers [20]. This library provides methods for safely evaluating SES-compliant code in an isolated environment. A key feature of the library is that it can execute completely at the client side and hence doesn't rely on any custom server-side architecture. JSand uses the SES library to realize its underlying object-capability environment.

However, since SES is a subset of ES5 strict, which in turn is a subset of ES5 non-strict, not all currently deployed JavaScript scripts are SES-compliant. Furthermore, the language supported by the SES library differs from true SES in several minor ways, further reducing compatibility with legacy scripts. Two important incompatibilities between ES5 and the SES-like language supported by the SES library are described below.

Global variables In ES5, the global `window` object can have arbitrary properties and for each of these properties there is a corresponding global variable with the same name. Conversely, for any global variable, a corresponding property with the same name is defined on the global object. In SES, this is no longer the case: global variables are not aliased by properties on the global object or vice versa.

Strict mode SES enforces strict mode for all scripts. Hence, ES5 non-strict code might be incompatible with SES. For instance, strict mode drops support for the `with` keyword, prevents the introduction of new variables into the outer scope by an `eval` and no longer binds `this` to the global object in a function call.

SES was designed to support (only) recognized ES5 best practices. Therefore, scripts that adhere to these best practice standards are SES-compliant and hence we expect the number of fully SES-compliant scripts to increase progressively as these best practices become more widespread. Although not all legacy scripts run without errors under the SES library, the secure confinement of these scripts is never at stake. Nevertheless, we have developed a support layer to improve compatibility with legacy scripts. This layer is described in detail in Section 4.6.

To enforce the object-capability model and to provide support for legacy scripts, the SES library and the support layer need access to the source code of scripts to be sandboxed. Our prototype fetches this code using the XMLHttpRequest (XHR) API. By default, this API is subject to the SOP, but recently added web features have facilitated cross-domain interactions, namely Cross-Origin Resource Sharing (CORS) [37] and the Uniform Messaging Policy (UMP) [38]. In case CORS or UMP are not supported by the script provider, our solution can fall back to a server-side JavaScript proxy [39].

4.2 Policy-enforcing membranes

4.2.1 *The Proxy API*

To implement the Membrane pattern in an efficient and transparent way, our prototype uses the Harmony Proxy API, which is scheduled to be standardized in the next version of ECMAScript [35]. This API enables us to create wrappers that generically intercept all property accesses and assignments on specific objects, as shown in the code below.

```
function wrap(target, policy) {
  var handler = {
    get: function(proxy, propertyName) {
      if (policy.isGetAllowed(propertyName)) {
        return target[name];
      }
      return null;
    }
    set: function(proxy, propertyName, value) {
      if (policy.isSetAllowed(propertyName)) {
        target[name] = value;
        return true;
      }
      return false;
    }
  }
  return Proxy.create(handler, Object.getPrototypeOf(
     target));
}
```

The `wrap` function creates a simple policy-enforcing wrapper around a specific `target` object. All property accesses and assignments on this wrapper are intercepted by the `get` and `set` traps of the handler object, which uses the `policy` object to determine whether or not the access or assignment is allowed.

4.2.2 *Membrane implementation*

To implement the Membrane pattern, the handlers used in JSand transitively wrap all objects they return from the `get` trap and unwrap the objects they receive in the `set` trap. The entire prototype chain of a wrapper must be wrapped as well, to prevent an attacker from piercing the membrane by accessing an unwrapped prototype.

If an object to be returned from the `get` trap is a function, a function proxy that wraps the original function is returned. This function proxy first unwraps all its arguments, then calls the original function using these unwrapped arguments and finally wraps the return value before returning it to the caller, thereby further expanding the metaphorical membrane. Some methods, such as `window.addEventListener`, take a callback function as an argument; like all other arguments, this callback must be wrapped appropriately to uphold the membrane. Because a callback function is executed in the context of a sandbox, its wrapper must wrap each of its arguments and must unwrap the return value after calling the original function with the wrapped arguments.

Each sandbox keeps a mapping from its wrappers to the target objects they wrap and vice versa. This makes it possible to unwrap previously wrapped objects and to ensure that there is at most one wrapper (per sandbox) corresponding to each target object, making the membrane identity-preserving [4]. The mapping from wrappers to their corresponding targets is only accessible from outside the sandbox, for otherwise an attacker could use it to escape from the sandboxed environment.

Whereas sandboxed code should always be confined to the bounds of its own sandbox, many use cases require an operation to introduce code from outside a sandbox into an existing sandbox. Such operations enable a website owner to extend or interact with a sandboxed script. JSand sandboxes provide two functions for introducing new code into them: `innerEval(code)` and `innerLoadScript(url)`. The first function evaluates a literal code string, while the second loads a script at a given URL.

In conclusion, the Membrane pattern transparently isolates a sandbox from code running outside of it or in other sandboxes. Since the handlers intercept each property access and assignment made on a wrapper, they contain the enforcement points which consult the security policy to determine whether or not an operation is permitted.

4.3 Security policies

Defining good security policies is important for ensuring the secure confinement of sandboxed scripts. To avoid needing a *known-good* version of a script to be sandboxed, a policy should be based on the claimed functionality of a script, as opposed to being based on actions performed by any specific version of the script. Generic templates can be provided to support website owners in defining good security policies.

Since the JSand architecture is independent of the specific type of security policy to enforce, policies can range from simple stateless policies, to arbitrarily complex policies. In both cases, the security policy can be specified as a JavaScript function that takes information about the operation to be performed as input and returns a boolean indicating whether or not the operation is allowed. We discuss three types of policies in more detail below.

4.3.1 Stateless policies

Stateless policies determine whether or not an operation is permitted based on information associated with that operation alone. For instance, a stateless security policy could specify that a specific function call performed on a specific object is only allowed when the value of the first argument is on a predefined whitelist.

WebJail [34] is an example of such a stateless policy for securely integrating third-party JavaScript. It classifies security-sensitive operations into nine categories, including DOM access, cookies, external communication, device access, etc., which can be permitted or blocked individually. A WebJail policy is based on a static whitelist of each of these categories, and could easily be implemented with JSand.

4.3.2 Stateful policies

Stateful policies can accumulate internal security state over multiple calls and use this global state as part of the policy, in addition to the local information made available on each operation request. For instance, a stateful security policy could specify that the use of XHR is allowed as long as no cookies have been read. This type of policy is more expressive than its stateless counterpart, but it is also more complex to specify and more prone to mistakes.

The shadow page in AdJail [32] is another example of internal state that could be accumulated over multiple calls. This page represents a ghost DOM, which is not directly rendered to the user, but allows an advertisement to execute various DOM operations in a confined environment.

4.3.3 Advanced policies

More complex policies can be used to enforce more advanced security properties, such as information flow security. One example of this is a set of policies to implement *noninterference* through *secure multi-execution* (SME) [7, 5]. For any script, SME can classify each input and each output channel as either H (high security, confidential) or L (low security, public). A script is noninterferent if its low-level outputs are not influenced by high-level inputs. Consider for instance the following script on a webserver at `mydomain.com`.

```
var cookies = document.cookie;
document.getElementById('some-img').src = 'http://
    attacker.com/img.jpg?c=' + escape(cookies);
```

The first line can be classified as H input, since cookie values are security sensitive. The second line can be classified as L output, since this triggers an HTTP request to a different domain. This program is interferent, because the low-level output statement at line 2 is clearly influenced by the high-level input statement at line 1.

Under secure multi-execution, a script is run multiple times, once for each security level. Outputs of a given security level are only generated in the execution belonging to that security level and inputs of a given security level are replaced by `undefined` in all executions of a lower level. Hence, high-level, security-sensitive input can never leak to low-level, public output channels, or even have an influence on them.

To multi-execute a script using JSand, that script must be executed once for the low security level and once for the high security level, each time in a different sandbox, with a different security policy. The low-level policy should disable all high-level inputs and ignore high-level outputs, while the high-level policy should simply ignore low-level outputs. Since each output statement is executed in only one of the executions, the net effect of a noninterferent script under secure multi-execution will be the same as the net effect of executing the same script without multi-execution.

4.4 Wrapping the DOM

All interactions between a script and the browser are performed through the DOM. Hence, to control access to all security-sensitive operations, JSand needs to control access to all facets of the DOM. To implement this, each sandboxed script is initially only given a single reference to a wrapper of the `window` object, which is the root of the DOM tree. As described in Section 4.2, all property accesses, property assignments and function calls on this wrapper or on any object transitively reached from it are intercepted by a handler. These handlers can thus enforce an arbitrary policy on the entire DOM, and hence effectively control access to all security-sensitive operations.

For any DOM object wrapper, a distinction can be made between two categories of properties. The first category consists of *standard DOM properties*, i.e. properties that are part of the DOM as defined by the ECMAScript standard (or implementation-specific properties provided by the browser). The second category consists of *custom properties* that have been added to a DOM object wrapper by a sandboxed script. For instance, `window.document` belongs to the first category, while `window.googlemaps` could belong to the second. Properties from these two categories need to be handled differently. Assignments to standard DOM properties should be propagated outside the sandbox to the corresponding target property on the real DOM object (if allowed by the security policy), since this is the only way a sandboxed script can interact with the browser. Custom properties should however be confined to the bounds of the sandbox, to prevent sandboxed code from polluting the global namespace and from reading or modifying properties defined outside the sandbox.

To make the distinction between standard DOM properties and custom properties, JSand uses a statically defined *DOM description*, derived from the W3C DOM specification [36]. This description consists of an array of property descriptors, indexed by a DOM object name (e.g. `Window`) and a property name (e.g. `alert`). Since each descriptor corresponds to a standard DOM property, they enable the handlers to determine whether or not a given property is a standard DOM property

4.5 Dynamic script loading support

From experience, we have learned that many scripts dynamically load additional scripts during their execution. This is typically accomplished by inserting a new script tag with a `src` attribute in the document, because this method is not under restriction of the same origin policy. However, when a script is included this way, it is executed in the global context. Hence, if we would allow sandboxed scripts to simply add new script tags to the document, they could trivially break out of their sandbox. Any script included by a sandboxed script should execute within that same sandbox.

For this reason, JSand uses special handlers to intercept methods that allow script tags to be added to the document, including `node.appendChild`, `node.insertBefore`, `node.replaceChild`, `node.insertAfter` and `document.write`. The first four of these take a (partial) DOM tree as argument and append or insert it at a certain place in the DOM. Our handlers for these methods search the given DOM tree for script tags, extract the value of the `src` attribute and execute the corresponding scripts in the sandbox that included them, using the `innerLoadScript` function described in Section 4.2.2. The `document.write` method is similar but takes an HTML string as argument and appends that string verbatim to the document. The handler for this method parses the given HTML string, extracts script tags out of it and loads them as described above.

We have considered two different techniques for parsing a given HTML string in JavaScript. The first technique consists of creating an iframe and setting its `srcdoc` attribute [1] to the given HTML. To prevent the iframe from fetching and executing scripts included in the HTML, its `sandbox` attribute [1] must be set as well. The second technique consists of using a pure JavaScript library to parse the HTML [11]. The iframe-based technique has a potential performance benefit, since the parsing is done by native code in the browser instead of in JavaScript. Moreover, using the first technique ensures that the HTML is parsed exactly as the browser will interpret it. However, one of the problems of this approach, is that the parsing is performed asynchronously. That is, we can only access the iframe's fully populated DOM tree from its `onload` callback, which is triggered some time after setting the `srcdoc` attribute. Consequently, scripts that immediately make use of the HTML written by `document.write` could fail, since the HTML might not yet have been processed. Performing a continuation-passing style transformation on these scripts could solve this problem, but this is a complex transformation which we leave for future work. Our prototype uses the second technique, since it does not suffer from this problem.

4.6 Support for legacy scripts

Although the SES library natively supports scripts adhering to recognized ES5 best practices, as described in Section 4.1 not all currently deployed JavaScript scripts do so. Although the secure confinement of legacy scripts is never at stake, not all of them run without errors under the SES library. Therefore, we have developed a support layer to further improve the compatibility with these legacy scripts, based on three abstract syntax tree (AST) transformations.

T1 Adding a property to the global `window` object normally introduces that property as a global variable, but this does not hold in a SES environment. This transformation introduces a global alias variable for each property of `window`. The variable is updated whenever an assignment is made to its corresponding property.

T2 Conversely, declaring a global variable normally creates an alias property on the `window` object, but this doesn't hold in a SES environment. This transformation adds a property on `window` for each global variable. The property is updated whenever an assignment is made to its corresponding global variable.

T3 Since SES enforces strict mode for all scripts, ES5 non-strict code might be incompatible with SES. The most common incompatibility we have encountered is the lack of `this`-coercion. That is, `this` is no longer bound to the global `window` object in a function call. This transformation replaces `this` by the expression `(this === undefined ? window : this)`.

We have implemented a client-side component for applying these transformations, using the UglifyJS JavaScript parser [19]. These transformations do not provide a full translation from ES5 to SES, but they are sufficient to make many legacy scripts work with our prototype.

5. EVALUATION

In this section we evaluate to what extent JSand satisfies the requirements set forth in Section 2.3.

5.1 Complete mediation

All sandboxed scripts are executed in an object-capability environment, set up by the SES library. Our implementation of the Membrane pattern ensures that each DOM access and JavaScript API call made by a sandboxed script is assessed by the security policy. Based on the theory of object-capability systems, this provides complete mediation.

Note that JSand provides a one-way isolation and hence makes no attempt to protect a sandboxed script from its environment. That is, code running in the global security context, such as browser plugins and unsandboxed scripts, have the power to modify a sandbox's security policy or to inject a DOM proxy that allows access to any DOM object. However, since malicious global code has already full power over the web page, we consider protecting against such scenarios out of scope for our solution.

5.2 Backwards compatibility

We have extensively and successfully tested our prototype on a variety of JavaScript scripts. In this section we report and discuss in detail three of the most widespread included scripts around: Google Analytics, Google Maps and the jQuery library. Google Analytics is included from more than 68% of all domains from the Alexa Top 10 000, making it the most included script on this list [23]. Google Maps is the most included web mashup API according to [28], being used in 17.41% of registered mashups. jQuery is the most popular JavaScript library in use today, included in more than 57% of the top 10 000 websites to date [2]. As future work, we would like to extend our evaluation to more legacy scripts.

5.2.1 Google Analytics

Google Analytics (GA) is a web analytics service that generates statistics about visitors to a website. The GA API allows web administrators to collect custom visitor properties, in addition to the standard properties that are collected by default (such as referrer and geographical location). The collected statistics can be monitored using a dashboard interface on the GA website.

To enable GA, the website owner must add a small JavaScript code template provided by Google to the header of the page to track. This template sets up an array of options to pass to the GA service and dynamically adds a new script tag to the page to include the main GA script. Any script included like this has unrestricted access to the DOM, making the page vulnerable to malicious script inclusions.

Manual inspection of the GA script is practically impossible, since the code is minified. Moreover, since the main GA script is loaded dynamically from the Google servers, any static, offline security analysis would fail to detect malicious changes introduced to the script after the initial analysis. However, by running GA in a JSand sandbox with a policy that permits only the operations necessary for a benign web analytics script, the impact of a malicious action on behalf of the GA script can be reduced to a minimum. The code snippet below shows how this can be implemented.

```
// main page:
var sb = new jsand.Sandbox('ganalytics.js',policy);
sb.load();
```

This code snippet creates a new sandbox and initializes it with the `ganalytics.js` script, which is shown below and consists of the code template provided by Google.

```
// ganalytics.js:
var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-xxxxxxxx-x']);
_gaq.push(['_trackPageview']);

(function() {
  var ga = document.createElement('script');
  ga.src = 'http://www.google-analytics.com/ga.js';
  var s = document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(ga, s);
})();
```

Both the `ganalytics.js` script and the main `ga.js` script (which is loaded from the code above) are executed in the same sandbox and are patched-up automatically, based on the AST transformations described in Section 4.6. The following code fragment shows the first two lines of the patched-up `ganalytics.js`.

```
// patched-up ganalytics.js:
var _gaq = _gaq || [];
window._gaq = _gaq;
[...]
```

The global variable `_gaq` is explicitly aliased as a property on `window`. This transformation is necessary because the `ga.js` script frequently refers to the `_gaq` array as `window._gaq`. Such references would fail without the patch shown here.

The `_gaq` array exposes an API to interact with GA after it has been initialized, for instance to add a custom property to collect or to track the click of a button. The website owner can access this array using the `innerEval` method described in Section 4.2.2. To facilitate these interactions and to make abstraction of the fact that GA is running in a sandbox, the website owner could implement an object that automatically forwards its calls to the `_gaq` array inside the sandbox.

Clearly, the effort required to run GA in a JSand sandbox is minimal and introduces no disruptive changes whatsoever. Nevertheless, the power of the GA script is reduced to a safe minimum, dramatically reducing the impact of a malicious script inclusion attack.

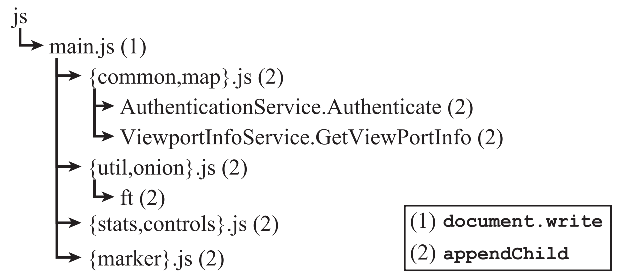

Figure 2: Tree of scripts dynamically loaded by Google Maps.

5.2.2 *Google Maps*

The Google Maps (GM) API enables website owners to embed a Google Maps gadget on their website. The standard way to add this gadget to a page is to (1) place a div element somewhere in the body where the map should be displayed, (2) add a script tag to the head of the page, which loads the GM library from the Google servers and (3) add a small piece of JavaScript code to the page, to create a new GM instance in the div element.

As with Google Analytics, the default way of including the GM script lets it have unrestricted access to the DOM and JavaScript APIs, putting the confidentiality and integrity of the entire web page at risk. JSand enables the website owner to confine the GM gadget to a sandbox with the minimal privileges required for legitimate operation.

The steps required to run GM in a JSand sandbox are very similar to the standard steps described above. In step (1), in addition to placing a div element somewhere in the body, the integrator must include the JSand library and the libraries it depends on. In step (2), instead of adding a script tag to directly load the GM library in the global page context, a new sandbox must be created for the GM script to run in. In step (3), the website owner can use the `innerEval` method to create a new GM instance in the sandbox. These steps are depicted in the following code fragment.

```
var sb = new jsand.Sandbox('http://maps.googleapis.
    com/maps/api/js?sensor=false', policy);
sb.load();
sb.innerEval(
 "var m = window.google.maps;
  var options = {
    center: new m.LatLng(-34.397, 150.644),
    zoom: 8, mapTypeId: m.MapTypeId.ROADMAP
  };
  var map = new m.Map(document.getElementById('
      map_div'), options);"
);
```

When the main GM script is loaded, a complex process of dynamically loading and patching other scripts is performed in the background. Figure 2 depicts the sequence of scripts that are dynamically loaded from the main `js` script initially loaded in step (2). In addition to the scripts shown in this figure, more scripts are loaded and patched whenever the user changes the map's viewport (by dragging it or changing the zoom level). All three translations described in Section 4.6 are required for the GM gadget to work.

The GM API provides extensive support for customization, to support feature-rich web mashups built around the

GM gadget. For instance, website owners can provide custom map overlays, place markers, register callbacks for mouse events, etc. As with GA, a website owner can use the **innerEval** method to interact with the sandboxed GM gadget.

The fact that JSand can successfully execute this gadget in a sandbox without any compatibility issues, illustrates that our solution is able to sandbox complex JavaScript gadgets that depend on dynamic script inclusions and that feature advanced DOM interactions.

5.2.3 *jQuery*

The jQuery library aims to provide a simple cross-browser API for performing common JavaScript operations, such as creating and selecting DOM elements, handling events, invoking Ajax interactions, etc. While jQuery can be used as an abstraction layer on top of an extensive set of JavaScript APIs, a website owner typically uses only a limited subset of what the library has to offer. By running jQuery in a sandbox with tight restrictions on the permitted JavaScript API and DOM operations, the risk and impact of a malicious script inclusion attack are reduced dramatically.

For our jQuery evaluation scenario, we executed jQuery together with the jQuery-geolocation plugin [24] in a sandbox, using a fine-grained security policy that allows us to toggle access to the JavaScript Geolocation API. Disabling the Geolocation API in the policy effectively prevents jQuery from using it in the sandbox. The following code fragment shows how this scenario is implemented.

```
var sb = new jsand.Sandbox('jquery-1.7.2.js',policy);
sb.load();
sb.innerLoadScript('jquery-geolocation-0.1.js');
sb.innerEval(
 "if (jQuery.geolocation.support()) {
    jQuery.geolocation.find(function(loc) {
      alert(loc.latitude+\", \"+loc.longitude);
    });
  } else { alert('Geolocation not supported'); }");
```

This scenario illustrates that, with minimal effort, a website owner can create a secure JSand sandbox around an extensible JavaScript library, while still being able to interact with it from outside the sandbox.

5.3 Performance benchmarks

To evaluate the runtime overhead of our prototype, we have conducted micro- and macro-benchmarks. All benchmarks were run using Google Chrome v20.0.1132.21 on Ubuntu 11.04 x86-64, running on an Intel Core 2 Duo T8300 2.4GHz processor with 4 GiB of RAM.

5.3.1 *Micro benchmarks*

JSand framework load time. To measure the load time of the JSand framework, a page was created that loads the framework but doesn't use it. This page was reloaded 1000 times and the elapsed time was recorded. The average load time measured in this way was 71.5±1.8 ms. The same experiment was run with all JavaScript code commented out, so the same network load time would be maintained, but the code would not be executed. The load time in this case was 23.0±0.2 ms. This means that once loaded from the network, the framework takes on average 48.5 ms to deploy on the client side.

Third-party library load time. Similar experiments were performed to measure the overhead of loading and parsing a third-party JavaScript library into a JSand sandbox. We chose jQuery as a representative JavaScript library and loaded it in a JSand sandbox, as well as a regular, unsandboxed JavaScript environment, using XHR and eval(). In both cases, we supplied a real JavaScript library as well as a commented-out version to factor out network overhead.

In a regular JavaScript environment, the code loads in 53.0±0.8 ms and 26.8±0.2 ms for normal and commented-out code respectively. Inside a JSand sandbox, the code loads in 1458.2±16.0 ms and 107.6±1.4 ms respectively, so that the overhead of parsing the library code is about 1350.6 ms.

A large portion of this overhead is due to the script rewriter of the legacy support layer described in Section 4.6. Since jQuery is SES-compliant, this rewriting step is not required. Disabling it lowers the average load time from 1458.2 ms to 705.8±1.1 ms, and the average overhead from 1350.6 ms to 598.2 ms. This means that 44.3% of the overhead can be contributed to our efforts for making legacy code SES-compliant.

Membrane transition cost. To verify the runtime overhead of a function call crossing the membrane, a function was executed both inside and outside a JSand sandbox 1 million times and the elapsed time is recorded. We chose the **window.clearTimeout** function as a representative function, because intuitively it should return quickly when no timer is registered. When called from inside the sandbox, the **window.clearTimeout** call must cross the membrane separating the sandbox from the real JavaScript environment. Outside the sandbox, the average execution time is 0.9±0.0 μs, while inside the sandbox it is 8.0±0.1 μs.

5.3.2 *Macro benchmarks*

The most important metric that counts when executing JavaScript in a browser, is the user experience. Ideally, the user should not notice that JSand is being used at all. To measure how much overhead the user experiences, we created a typical web application using Google Maps and measured two things: the total load time of the web application, and the delay a user experiences when interacting with it.

The load time of the web application was measured from the time the page is loaded until the Google Maps API emits a 'tilesloaded' event, signaling that the application is ready to be used. Running outside of the JSand sandbox, this load time is 308.0±13.7 ms, and 1432.8±24.2 ms inside of it. Keeping in mind that a large portion of this overhead is due to script-rewriting for legacy code, the total overhead without the legacy support layer can be estimated to be about 626.5 ms.

To measure the delay experienced when interacting with the application, we waited until the application was loaded, and then panned 400 px to the right, 100 times. The average time elapsed between two pans was considered as a reasonable approximation of the user-experienced delay. This delay is 320.2±0.8 ms outside and 420.0±2.7 ms inside the sandbox.

The overall performance of a JSand sandbox is acceptable. The overhead when loading a reasonably-sized SES-compliant JavaScript library inside the sandbox, is about 203%. For legacy scripts, JSand requires a code transformation step that results in a total overhead of about 365%, but it is expected that this step can be removed or at least sped up significantly for future JavaScript code in future browsers. Furthermore, the tendency of users to keep certain websites open using persistent tabs, makes the load time overhead less important. Additionally, despite the ninefold execution time of a function-call traversing the sandbox membrane, the delay experienced by a user when using a realistic web application inside a JSand sandbox, is an acceptable 31.2%, corresponding to an absolute delay on the order of 100 ms.

6. RELATED WORK

Server-side processing of scripts.

A common technique for preventing undesired script behavior is to restrict the untrusted code (i.e. the third-party component) to a safe subset of JavaScript [15]. Compliance to the subset is verified at the server side. The allowed operations within the subset prevent the untrusted code from obtaining elevated privileges, unless explicitly allowed by the integrator. ADSafe [3], ADsafety [27] and FBJS [33] are examples of techniques where third-party JavaScript must conform to a certain JavaScript subset. Techniques such as Caja [22], Jacaranda [9] and Live Labs' Websandbox [18] on the other hand, statically analyze and rewrite the third-party JavaScript on the server side into a safe version.

Instead of forcing the use of a JavaScript subset, the JavaScript code can also be instrumented with extra checks that mediate access to certain functionality. BrowserShield [29] and Browser-Enforced Embedded Policies (BEEP) [10] are examples of such instrumentation on the server-side.

While safe subsets, code rewriting and server-side code instrumentation can restrict third-party code at the source, their adoption by mashup integrators is problematic. These techniques require either access to code running on the server-side, or require the website owner to implicitly trust the JavaScript provider to deliver safe JavaScript code. In real-world scenarios, it is infeasible to impose any such restrictions on third-party code providers. In contrast, JSand requires no server-side processing of the third-party code and imposes no fundamental restrictions on included code.

Extending the browser with a reference monitor.

A second class of techniques extends the browser to enforce code restrictions. Systems like ConScript [17], WebJail [34] and Contego [13] require modifications to the JavaScript engine to enforce policies on third-party code, while AdSentry requires the installation of a Firefox extension to restrict the functionality available to advertisements.

Browser modifications to restrict third-party JavaScript can be implemented very efficiently and can guarantee that enforcement cannot be circumvented. The major disadvantage of this approach however, is that the browser must be modified. Unless all the users of a web application are using a browser which implements the desired modification, there is little or no incentive for the website owner to make use of it. Because of the large variety of active browser vendors and versions on the internet, it is unrealistic to assume that a certain modification will ever be implemented in all browsers. For this reason, JSand does not depend on any special browser-side features except for what is available in the web standards.

Leveraging existing browser security features.

Finally, some approaches leverage recent browser security extensions to contain scripts. The new `sandbox` attribute of the iframe element in HTML5 [1] can restrict third-party JavaScript in a very coarse-grained way: it only supports to completely enable or disable JavaScript.

The Content Security Policy (CSP) [31] allows the insertion of a security policy through HTTP response headers and meta tags, which must be enforced in the browser. This policy can restrict the locations a web application loads its content from, thus preventing some forms of content-injection. However, CSP does not provide any fine-grained control over which JavaScript functionality is available to a script.

AdJail [32] is geared towards securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page that the web developer wishes the ad to have access to, and it relies on the SOP to isolate the shadow page. Changes to the shadow page are replicated to the hosting page if those changes conform to a specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy. AdJail is a good approach to restrict access to the DOM, but cannot enforce a policy on the other JavaScript APIs like JSand does.

Self-protecting JavaScript (SPJS) [26, 16] is a client-side wrapping technique that applies advice around JavaScript functions, without requiring browser modifications (unlike [17] or [34]). It builds on standard aspect-oriented libraries for JavaScript. The wrapping code and advice are provided by the server and are executed first, ensuring a clean environment to start from. SPJS does not guarantee that all access-paths to certain JavaScript functionality can be restricted, because the aspect library it relies on was not designed with security in mind. JSand uses the Membrane pattern instead, which was designed to provide complete mediation.

Secure ECMAScript (SES) [20] is a subset of ES5 strict which provides an object-capability language. Unlike Caja, from which it originated, SES runs completely on the client-side without any browser modifications. To the best of our knowledge, JSand is the first fully functional JavaScript integration technique built on SES, capable of handling legacy scripts such as Google Maps and Google Analytics.

7. CONCLUSION

This paper introduced JSand, a server-driven but client-side JavaScript sandboxing framework that does not rely on any browser modifications. We have implemented a prototype of this framework and evaluated it on the most widespread JavaScript scripts around. Although there has been a lot of activity in this research area, we are the first to deliver a solution that provides complete mediation, backwards compatibility and an acceptable performance overhead.

Acknowledgements

This research is partially funded by the Research Fund KU Leuven, the EU-funded FP7 projects NESSoS and WebSand and by the IWT-SBO project SPION. Pieter Agten holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO). With the financial support from the Prevention of and Fight against Crime Programme of the European Union European Commission - Directorate-General Home Affairs.

8. REFERENCES

[1] R. Berjon. W3C HTML5 Working Draft. http://www.w3.org/TR/html5/, September 2012.

[2] BuiltWith. jQuery Usage Statistics. http://trends.builtwith.com/javascript/jQuery.

[3] D. Crockford. ADsafe – making JavaScript safe for advertising. http://adsafe.org/.

[4] T. V. Cutsem and M. S. Miller. On the Design of the ECMAScript Reflection API. Technical Report VUB-SOFT-TR-12-03, Department of Computer Science, Vrije Universiteit Brussel, February 2012.

[5] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proc. of CCS'12*. ACM, 2012.

[6] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. In *Proc. of NordSec'10*. Springer, 2011.

[7] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc of SP'10*, IEEE, pages 109–124, Washington, DC, USA, 2010.

[8] M. Heiderich. Locking the Throne Room - How ES5+ will change XSS and Client Side Security. http://www.slideshare.net/x00mario/locking-the-throneroom-20, November 2011.

[9] Jacaranda. Jacaranda. http://jacaranda.org.

[10] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proc. of WWW'07*, pages 601–610, New York, NY, USA, 2007. ACM.

[11] John Resig. Pure JavaScript HTML Parser. http://ejohn.org/blog/pure-javascript-html-parser/.

[12] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. http://www.webappsec.org/projects/articles/071105.shtml, April 2005.

[13] T. Luo and W. Du. Contego: capability-based access control for web browsers. TRUST'11, pages 231–238, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc. of SP'10*. IEEE, 2010.

[15] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009.

[16] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *Proc. of Nordsec'10*, 2010.

[17] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *Proc. of SP'10*, 2010.

[18] Microsoft Live Labs. Live Labs Websandbox. http://websandbox.org.

[19] Mihai Bazon. UglifyJS. https://github.com/mishoo/UglifyJS/.

[20] M. S. Miller. Secure EcmaScript 5. http://code.google.com/p/es-lab/wiki/SecureEcmaScript.

[21] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control.* PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. AAI3245526.

[22] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.

[23] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proc. of CCS'12*, October 2012.

[24] NoMoreSleep. jquery-geolocation. http://code.google.com/p/jquery-geolocation/.

[25] P. H. Phung and L. Desmet. A two-tier sandbox architecture for untrusted javascript. In *Proc. of JSTools'12*, pages 1–10, New York, NY, 2012. ACM.

[26] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[27] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: type-based verification of JavaScript Sandboxing. In *Proc. of USENIX'11*, SEC'11, pages 12–12, Berkeley, CA, USA, 2011.

[28] Programmable Web. Keeping you up to date with APIs, mashups and the Web as platform. http://www.programmableweb.com/.

[29] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *Proc. of OSDI'06*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[30] SANS Institute. SANS: Top Cyber Security Risks. http://www.sans.org/top-cyber-security-risks/, 2009.

[31] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proc. of WWW'10*, pages 921–930, New York, NY, 2010. ACM.

[32] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *19th USENIX Security Symposium*, Aug. 2010.

[33] The FaceBook Team. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

[34] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM.

[35] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession APIs. *SIGPLAN Not.*, 45(12):59–72, Oct. 2010.

[36] W3C. Document Object Model (DOM) Technical Reports. http://www.w3.org/DOM/DOMTR.

[37] W3C. W3C Standards and drafts - Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.

[38] W3C. W3C Standards and drafts - Uniform Messaging Policy, Level One. http://www.w3.org/TR/UMP/.

[39] Yahoo! Developer Network. JavaScript: Use a Web Proxy for Cross-Domain XMLHttpRequest Calls. http://developer.yahoo.com/javascript/howto-proxy.html.

[40] C. Yue and H. Wang. Characterizing Insecure JavaScript Practices on the Web. In *Proc. of WWW'09*, pages 961–961, April 2009.

[41] M. Zalewski. Browser Security Handbook. http://code.google.com/p/browsersec/wiki/Main.