JaTE: Transparent and Efficient JavaScript Confinement

Tung Tran
Stony Brook University
tunghack@gmail.com

Riccardo Pelizzi
Stony Brook University
r.pelizzi@gmail.com

R. Sekar
Stony Brook University
sekar@cs.stonybrook.edu

ABSTRACT

Inclusion of third-party scripts is a common practice, even among major sites handling sensitive data. The default browser security policies are ill-suited for securing web sites from vulnerable or malicious third-party scripts: the choice is between full privilege (`<script>`) and isolation (`<iframe>`), with nearly all use cases (advertisement, libraries, analytics, etc.) requiring the former. Previous work attempted to bridge the gap between the two alternatives, but all the solutions were plagued by one or more of the following problems: (a) lack of compatibility, causing most existing third-party scripts to fail (b) excessive performance overheads, and (c) not supporting object-level policies. For these reasons, confinement of JavaScript code suitable for widespread deployment is still an open problem. Our solution, JaTE, has none of the above shortcomings. In contrast, our approach can be deployed on today's web sites, while imposing a relatively low overhead of about 20%, even on web pages that include about a megabyte of minified JavaScript code.

1. INTRODUCTION

A recent study [26] found that nearly 90% of web sites include third-party scripts. Unfortunately, this practice poses serious security threats to the first-party web site, threatening its integrity and confidentiality. Vulnerabilities in third-party code can expose the first-party to attacks such as cross-site scripting, or the third-party server may be outright malicious or be compromised. Major web sites such as Yahoo and New York Times [8, 6] have exposed their users to malware by including third-party content in the form of advertisements. As a result, there is a pressing need for approaches to protect web sites from third-party scripts, while preserving their functionality.

In order to protect first-party code, it is necessary to isolate third-party code from accessing (sensitive) first-party data or functions. There are two main approaches in this regard:

- *Frame-based isolation:* The browser's SOP isolates code running in different frames, while providing a controlled means for communicating through the `postMessage` API. AdJail [34], Mashic [17] and Pivot [23] rely on this approach for isolation. MashupOS [41] also relies on frames and similar isolation mechanisms. While COWL [32] extends a browser's SOP further to support a MAC policy, it continues to rely on frame-based isolation. The main drawback of frame-based isolation is that it limits interactions (between first- and third-party code) using familiar means such as passing objects, or calling another party's functions. This limits compatibility with existing first-party and third-party code.
- *Language-based isolation:* This class of techniques aims at isolating individual objects, so that objects can be shared between parties, and controlled interactions can take place through function calls. However, works in this area must first address the challenge of mediating all of the numerous avenues by which JavaScript programs can interact. Early works such as Caja [20] and BrowserShield [29] resorted to rewriting the code to introduce all the necessary runtime checks. Unfortunately, because of the dynamic nature of JavaScript, most operations need to be transformed and/or checked at runtime, often slowing programs down by an order of magnitude or more. An alternative approach is to develop static analysis techniques that can eliminate the need for most (or all) runtime checks. ADsafe [11], GateKeeper [13], SES [24], JSand [9] and others [19] opt for this approach. However, full JavaScript is not amenable to static analysis, thus forcing these techniques to impose language restrictions. Among these techniques, SES and JSand place the fewest language restrictions, but these are still too severe for real-world code: we found that 80% of the Alexa's Top 500 websites are not supported by them.

Our Goals. We seek a secure object-granularity policy enforcement infrastructure compatible with existing browsers as well as web sites, including all their first- and third-party code. Specifically, we seek:

- *Transparency*: The enforcement infrastructure should not change the execution semantics of benign code[1]. Our solution achieves this goal except for a few rare corner cases, none of which could be observed on any of the Alexa Top 500 websites. (See Section 6.2.)
- *Object-granularity policy*: The infrastructure should allow third-party code to access any subset of objects deemed safe by a policy developer, while preventing access to others. Even on permitted objects, access to individual operations can be sand-boxed.
- *Deployability on existing browsers:* To facilitate adoption, the approach must not require modifications to the browser (specifically, its JavaScript engine), nor can it impose unreasonable performance overheads.

Our Approach. We present JaTE, a new approach that satisfies the above requirements. Every object is associated

†This work was supported in part by grants from NSF (CNS-0831298 and CNS-1319137) and ONR (N00014-07-1-0928).

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA

ISBN 978-1-4503-3682-6/15/12...$15.00

DOI: http://dx.doi.org/10.1145/2818000.2818019

[1]Note that the goal of any security policy is to change the execution semantics of code that violates the policy. Thus, it is generally infeasible to ensure transparency in the presence of a nontrivial security policy. Moreover, since malicious code can easily detect the presence of a policy framework by simply trying out operations that any sensible policy must deny, we do not attempt to be transparent to malicious code.

with a principal, and this principal has direct access to the object, while the access of other principals is mediated using a *wrapper* object that can enforce a policy. The set of all objects belonging to a principal is held within the principal's *compartment* [40].

Many of the key challenges in JaTE, including *complete mediation* and the *realization of a secure multi-principal compartment model,* arise from the complexity and highly dynamic nature of JavaScript. We discuss these challenges in Section 2, followed by an overview and illustration of how our design overcomes them in Section 3. The design and implementation of JaTE is described in Sections 4 and 5 respectively. A detailed experimental evaluation is presented in Section 6, followed by a discussion of related work (Section 7) and concluding remarks (Section 8). Below we summarize the technical contributions of this paper.

Contributions.

- *Object-capability environment for full JavaScript.* Object capability ensures that only objects explicitly given to third-party code can be reached by it. It provides the basis for complete mediation. Ours is the first work to realize this feature without placing significant restrictions on the JavaScript language.
- *Secure and transparent muti-principal JavaScript confinement without browser modifications.* Our solution is ready for deployment on any web site because existing code does not need to be modified. It can support policies that protect mutually untrusting principals, e.g., two advertisers.
- *Efficient fine-grained object-level access control.*
- *Large-scale experimental evaluation of compatibility, performance, and functionality.* When enforcing an allow-all policy, our implementation demonstrates full compatibility with all sites from the Alexa Top 500, while incurring an average overhead of about 20%.

2. CHALLENGES

Complete mediation. To ensure complete mediation, all mechanisms for object access must be handled. This is a difficult task in JavaScript because the language supports several unusual ways to reference objects:

- *Global object access.* Securing global object access is critical because all other objects are reachable from it. In addition to the explicit mechanism of accessing the variable `window`, JavaScript provides implicit access to the global object via (a) free variables that are interpreted as property accesses on the global object, and (b) accesses to the `this` keyword within a function invoked without an object argument.
- *Native prototype access.* JavaScript relies on prototypes to support object inheritance. Prototypes of native objects are shared, thus providing a mechanism for third-party code to affect the semantics of first-party's use of native objects. Controlling this access is complex because third-party code can not only rely on direct access (e.g., update `Object.prototype`), but also indirect access. For instance, even a seemingly "safe" access to a third-party's own object `x` can allow it to update `Object.prototype` using the expression `x.__proto__`.
- *Call stack access.* JavaScript allows third-party code to travel up the call stack. This access can be used by a third-party function to access sensitive first-party data such as the arguments of the first-party function that invoked it.

Dynamic code. Dynamic code poses a well-recognized challenge to security. Previous works forbade most dynamic code (ADSafe, GateKeeper), or replaced `eval(s)` with a safe wrapper, say, `safeeval(s)` (Caja, SES, JSand). Unfortunately, use of a wrapper function might change the semantics of `s`: the free variables occurring in `s` are no longer resolved in the context where the original `eval` occurred, possibly altering the semantics of code such as:

```
var x=0; eval("alert(x)")
```

2.1 Discussion

Using an object-capability runtime is a well-established approach for achieving complete mediation [21, 24, 9, 18]. The major effort in this area is Secure ECMAScript (SES) [24], an object-capability language based on ES5. SES relies on ES5's *strict mode* to prevent the use of `caller` and implicit accesses to the global object via `this`. To eliminate the threat of code injection into native prototypes, it prevents their modification by freezing them all. Moreover, it replaces `eval` with a safe wrapper. All of these restrictions tend to break existing code, and indeed, backward compatibility wasn't their focus. As a result, we found that the vast majority of Alexa Top 500 web sites experience compatibility problems with SES.

JSand [9] uses the object-capability environment of SES to build a policy enforcement framework for third-party JavaScript code. JSand exposes permitted objects to third-party code using Miller's membrane pattern [25]. In JSand, a membrane consists of policy-enforcing wrappers around these objects. If any operation on a wrapped object returns another object, the membrane is extended to wrap the returned object as well.

A second major goal of JSand is to achieve compatibility with existing web sites. In addition to handling implicit access to `window` via `this`, JSand addresses a frequent incompatibility posed by SES: it performs a simple analysis to identify global variables in the third-party code, and transforms the code to explicitly synchronize their values with the correspondingly named attributes of `window`. While properties referenced statically can be synchronized this way, dynamic property accesses (e.g., `window[p]`) pose a challenge. Moreover, other incompatibilities posed by SES, including the remaining restrictions of strict mode, the use of an eval wrapper and the use of native prototype extensions, continue to affect JSand. We found that over 80% of Alexa Top 500 web sites fail to "compile" because of strict mode violations, while 30% and 49% violate the other two restrictions.

Instead of first denying access to the global object using SES and then partially mitigating these restrictions, JaTE is designed from the ground up with a single goal: intercept every access to protected objects, so that a policy can be applied to each of those accesses. JaTE exploits the dynamic and reflection features of JavaScript, together with a simple lexical analysis[2] and transformation of third-party code, to ensure that all object accesses are mediated at runtime. It does not place any significant restrictions on JavaScript, a fact confirmed by our evaluation on Alexa top 500 sites. (See

[2]Unlike JSand, JaTE does not require full parsing of JavaScript, but only a lexical analysis. All rewriting is done Just-In-Time and cannot be circumvented through obfuscation.

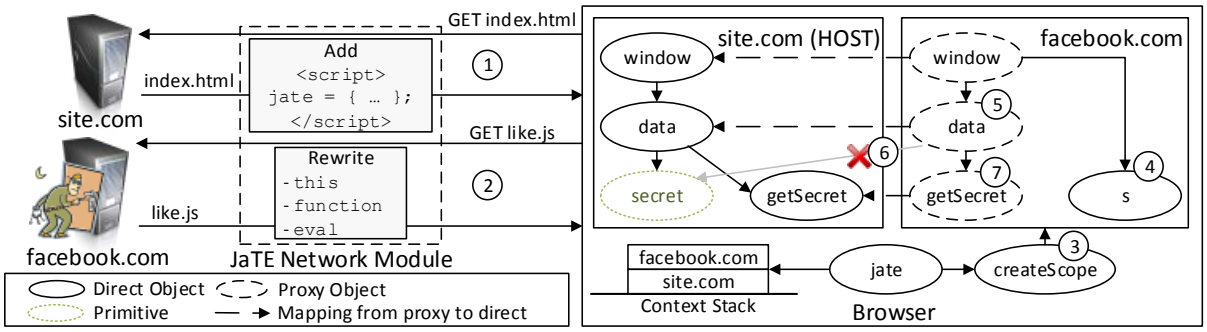

Figure 1: Example for a malicious Facebook "Like" Button

```
1  var stolen=data["se" + "cret"];
2  function s() {
3   var stolen=this.data.secret;
4  };
5  s();
6  stolen = data.getSecret();
7  eval("stolen=this.data.secret;");
```

Figure 2: Malicious "Like" Script

Section 6.2 for details.)

An important feature of JaTE is that it supports multiple mutually-distrusting principals, which arise in web pages that integrate content from multiple sources, e.g., several advertisers.

3. OVERVIEW

This section provides a high-level overview of how the compartment model confines third-party scripts using code transformation and runtime checking. We illustrate this using an example of first-party (also called host) web page that includes sensitive content in an inline script:

```
data = { secret:'xxx',
  getSecret:function(){return this.secret}}
```

Also assume that the page includes a Facebook "Like"-button, but Facebook's servers have been compromised to replace the button with malicious code that attempts to steal the value of `secret`.

The scenario begins with an HTTP request (1) in Figure 1 for retrieving the first-party web page. The JaTE Network Module intercepts this request and modifies the page to add an object `jate` that contains our confinement library. This module could be implemented in one of three ways: a client-side proxy, a browser extension, or a server-side proxy[3]. Our implementation relies on a browser extension.

In step (2), the "Like"-script included in the page is fetched from Facebook. It is transformed by the network module to enable secure policy enforcement (note that a policy can decide: a) if code from a domain/url will be confined and b) its corresponding principal). To illustrate the main elements of this rewriting step, consider the malicious "Like" script shown in Listing 2. It includes four distinct mechanisms to steal the secret:

A: through dynamic property access (line 1),

B: using `this`, which resolves to global object (lines 2-5),

[3]Requires cross-origin resource sharing (CORS).

```
1  jate.enterContext("facebook.com");
2  var scope = jate.createScope("facebook.com");
3  try {
4   with (scope) {
5    var s = function s() {
6     var stolen=processThis(this).data.secret;
7    };
8    var stolen=data["se" + "cret"];
9    s();
10   stolen=data.getSecret();
11   eval(processEval("stolen=this.data.secret"))
12  }
13 } finally { jate.exitContext(); }
```

Figure 3: Rewritten malicious "Like" script. Underlined code segments are added by JaTE.

C: using a function defined in first-party code (line 6), and

D: by executing dynamic code (line 7).

Listing 3 shows the rewritten script, with the transformations underlined. First, we introduce a preamble to setup a scope and enclose the original script using a `with` statement (used to intercept free variable access). The script is then transformed using three simple rules:

1. a global function declaration (e.g., function `s`) is turned into variable declaration and assignment and moved to the top of the script to simulate declaration hoisting,
2. `this` is replaced with `processThis(this)`, and
3. direct `eval` is transformed to rewrite its argument before evaluation.

Using these rules, JaTE is able to mediate all cross-compartment accesses, even those from dynamic code.

Step (3) in Figure 1 shows the effect of lines 2-4 from the rewritten script: this setup creates a new *compartment* for `facebook.com`. This compartment starts its life cycle with only a mediated reference to `window` as a global object, but, if permitted by the policy, it can obtain mediated references to objects reachable by the original `window`. Mediation is achieved using ECMAScript 6 *Proxies*, which enable transparent interception of all operations on objects.

Compartments represent trust boundaries within the same JavaScript execution environment: each party is confined within its own compartment (see Figure 1), and JaTE mediates all cross-compartment interactions. While the JaTE framework itself is general enough to support mutually distrusting first- and third-parties, the threat model considered in this paper is more limited: our goal is to (a) protect the

first-party from third-party, and (b) if there are multiple (mutually distrusting) third parties, then protect them from each other. In this scenario, there is no need to transform first-party scripts, and hence the host compartment holds an unmediated reference to `window`. Although the host code does not run in a compartment set up by JaTE, it is helpful to think of it as running in privileged compartment.

Step (4) in Figure 1 depicts the effect of variable declaration and assignment (lines 5 to 7) in Figure 3, which was originally a function declaration (lines 2 to 4) in Figure 2. Note that the object `s` is unmediated in `facebook.com`'s compartment because it is created by `facebook.com`. Step (5) shows the effect of line 8: the policy permits obtaining a mediated reference to `data`, but does not allow reading the value of secret (Step 6), which is a primitive value of type String. This stops attack (**A**).

Line 9 is an unmediated function call. However, since our transformation has rewritten the body of `s`, accesses to `this` now return a reference to a mediated version of `window`. When this mediated version is dereferenced, the policy once again stops reading of `secret`, thus stopping attack (**B**).

Line 10 obtains a mediated reference to `getSecret` (Step 7) and performs a mediated cross-compartment function call, which is denied by the policy, stopping attack (**C**).

Finally, line 11 evaluates the string after rewriting it just-in-time. Note that the exact same technique of Step (2) is applied again, using the same light-weight rewriting based on lexical analysis. The rewritten code is:

```
stolen=processThis(this).data.secret
```

This makes the attack semantically equivalent to the one on line 6 (Figure 2), and hence attack (**D**) is also stopped.

4. DESIGN

This section presents the core mechanisms to implement the compartment model for multiple mutually-distrusting principals. Specifically, Sections 4.1 and 4.2 describe JaTE's compartments, while Section 4.3 describes the handling of JavaScript's challenging features outlined in Section 2. Finally, Section 4.4 addresses secure DOM access.

Our compartment design relies on *Proxies,* a feature of the recently finalized ECMAScript 6 (ES 6) standard. A proxy could be created for any object `w` as follows:

```
pw = new Proxy(w, {get:getHandler})
```

where `getHandler` is a function. A read operation `pw.x` will invoke the function `getHandler`. This function can check if the access should be permitted, and if so, invoke `w.x`. If the policy check fails, the operation is not passed on to `w`, but instead, our handler raises an exception. (Alternatively, a safe default value can be returned, allowing the caller to continue normally.) Thus `pw` behaves like `w`, while enabling transparent interposition of policy checks before any access.

ES 6 defines several *traps* in addition to the get-trap illustrated above. These include the *has* trap (invoked to check if an object possesses a certain property), the *set* trap (invoked when a property is modified), and the *call* trap (invoked before calling a member function). Any subset of these trap handlers can be specified in the second argument to `Proxy`.

4.1 Mediating global object access

Assuming that `caller`, `this`, native prototypes and DOM are safely handled, the only way third-party code can access the global object is through free variables, which are interpreted in JavaScript as accesses to properties of the global object. We intercept all free variable accesses by exploiting JavaScript's dynamic nature: we construct a `scope` object as shown in Figure 3, and enclose third-party code inside a `with (scope) { }` block. This causes all free variable accesses in the enclosed code to be looked up on `scope`.

We construct `scope` to be a proxy object, and define its has-trap so that it returns *true.* As a result, JavaScript runtime never looks up any variable outside the `with` statement, thwarting any attempt by third-party code to directly access the global object. We also define the remaining traps of `scope` so that it forwards these accesses to the virtual global object, which is a proxy of the global object. This enables all policy checks to be performed in the virtual global object. Appendix 9.2 shows more details how the scope object is constructed.

While the discussion so far has considered accesses, declarations require additional care:

- *global variable declarations* (`var a`): The `with`-statement does not prevent enclosed code from declaring `a` as a property of the global object. However, note that this declaration has no effect if `a` is already a property of the global object. If not, it ends up declaring a new property with the value `undefined`. Note that any subsequent access to `a` will be intercepted by `scope`, so this declaration won't allow the enclosed code to bypass policy checks[4].
- *global variable declaration with initialization* (`var a = 1`). This case is treated by JavaScript as if it consisted of a variable declaration, followed by an assignment. Since we have already dealt with both statements, JaTE needs to take no additional step for this case.
- *global function declarations* (`function f() {}`): We transform this as `var f = function f(){}`, and move it to the top of the script. This means function declarations get handled in the same way as variable declarations.

4.2 Mediating cross-compartment accesses

In our construction, the first-party (aka "host") has direct access to the global object, as well as most built-in objects. We say that these objects are within the host compartment. The third-party code, as discussed in the above construction, starts its execution with just the virtual global object in its compartment.

During execution, a principal can introduce new objects into its compartment in two ways:

- It can create new objects. Ultimately, all object construction occurs using literals (e.g., `[]`), or built-in constructors (e.g., `Array`). We refer to these as *direct objects,* i.e., the principal's accesses to these objects are not mediated. Thus, JaTE introduces no additional overheads when a principal accesses the objects it *owns.*
- The principal can import objects owned by other principals through interactions that get mediated in the following proxy traps:
  - **get**: If a principal A reads a property of an object owned by principal B, and the result is an object owned

[4]There is a possibility that first party code will behave differently based on the existence of property `a`, and in this case, third-party can alter the behavior of first-party code. We consider this a side-channel that is unlikely to pose a security threat. A safer alternative, however, would be to simply delete any `var` declarations at the top level in the enclosed script.

by A, then a direct reference is returned. Otherwise, a proxy for that object is created and returned to A.

- **set**: This is handled in a similar manner, except that the direction of transfer is reversed in this case.
- **call**: A call can be treated as a switch from caller's to callee's compartment, followed by get operations to retrieve actual parameter values from the caller's compartment. When the function returns, a switch back to the caller's compartment takes place, followed by a get operation to retrieve the return value from the callee.

Note that all these operations are subject to the permissions specified by the policy. In other words, the above behavior would be observed with a default "allow all" policy, while a more restrictive policy would deny some of these accesses. Also note that other traps can be handled similarly, e.g., `deleteProperty` can be handled like the **set** trap.

Tracking current context. JaTE relies on the single-threaded nature of JavaScript: the context can only explicitly switch in two ways, either at the beginning and at the end of third-party code execution (handled by rewriting), or during a cross-compartment function call (handled by the call trap). In both cases, JaTE tracks the current context by updating the property `jate.currentContext`.

Tracking object ownership. JaTE does not need to know the owner of a direct object until it first crosses a compartment. When this happens, a proxy for the object is created, and its owner is determined and stored for future use.

Cross-Compartment Exceptions. As a general rule, a principal should always have direct access to its own objects, but only have proxies to the objects owned by other principals. However, there are a few exceptions: (a) certain built-in functions are frozen and always seen as direct to improve performance, (b) certain objects such as DOM nodes are always accessed via proxies, even by their owner, and (c) for security reasons, even the host sees only proxies of built-in constructors.

4.3 Handling JavaScript challenges

4.3.1 *Handling* `this`

JaTE replaces all occurrences of the `this` keyword with `processThis(this)`, where `processThis` returns the virtual global object if `this` is the global object.

4.3.2 *Handling* `caller`

Note that it is not possible to statically recognize and rewrite occurrences of `caller`: such an approach can be circumvented by obfuscation, e.g., `f["c"+"aller"]`. Moreover, since `caller` is non-standard, we cannot rely on its "official" semantics either. Instead, we have developed a solution that is based on how it has been implemented on major browsers, including Chrome, Firefox, Safari, and Internet Explorer. On these browsers, `caller` is not determined from a stack frame, but simply has a single value that records the most recent (and still active) caller of a function. As a result, if a function `f` is recursive, after the first recursive call, `f.caller` becomes `f`. Hence

```
(f.caller).caller = f.caller = f
```

In otherwords, regardless of how many times `caller` is invoked, it becomes impossible to get to the caller of the outermost invocation of `f`.

JaTE relies on the above semantics of `caller` to ensure that third-party code, when called by another principal X, cannot reach X's stack frames. To illustrate the approach, suppose that `g` is a host function that needs to call a third-party function `h`. Since this is a cross-compartment call, it will go through a call-trap handler, which then calls a function `f` defined below:

```
var t=1;
function f() {if (t) {t=0; f();} else h();}
```

When `h` tries to use `caller` to get to functions in the call stack, it cannot get any further than `f`, hence it cannot get to `g`.

4.3.3 *Handling native prototypes*

Intercepting native prototype accesses. JaTE lets principals handle direct references to the objects they create. However, a direct object contains references to the object's native prototype and its properties. Since native prototypes are shared among all principals, JaTE must ensure that third-party code does not obtain direct references to them. The most natural way to achieve this is to set a native prototype to a proxy. Unfortunately, all native prototypes are non-configurable and non-writable, and so JaTE cannot change them. For instance, `Object.prototype` cannot be made to point to a proxy of the real prototype. Instead, it is necessary to intercept every possible way to get to a native prototype, and at that point, return a proxy.

Native prototypes can be accessed through native constructors or `__proto__`. For example, native `Array` prototype can be accessed using `Array.prototype` or `x.__proto__` (where `x` denotes any array value). Therefore, we first replace native constructors with proxies. This is done for all native constructors such as `Object` and `Array`. For instance, `Object` is transformed as:

```
var origObject = Object;
Object = jate.createProxy(origObject);
origObject.prototype.constructor = Object;
```

To handle `__proto__`, we replace `__proto__`'s built-in getter. The new getter will return a proxy if it is one of the native prototype objects.

Handling accesses to native prototype properties. For performance reasons, we identified and white-listed native prototype functions that can be safely called directly by any principal, e.g., prototype functions of `Array`, `Object` and `String`. These functions are frozen to prevent malicious principals from replacing their implementations[5], and then a direct reference to these functions is returned. For functions determined unsafe (for direct calls from untrusted code), only a proxy is returned. While performing this safety analysis, we found a bug in V8's `Array` prototype functions. This bug, which led to a leak of the global object, was reported [7] and promptly fixed by Google.

For any property `p` that is newly added to native prototypes[6], JaTE instead stores a proxy to `p`. When the current context is switched to a principal P, JaTE converts all properties added to native prototypes and owned by P to direct

[5] In theory, this can affect transparency as it would break code that attempts such replacement. In practice, however, we find that overwriting of these built-in functions don't seem to occur.

[6] Even though this is considered a bad practice, it seems to be fairly common — our tests have shown that nearly half of the Top 500 websites extend native prototypes, perhaps because many of them use the popular `Prototype` library.

```
// indirect eval calls
(1, eval)('...')
var e = eval; e('...');
window.eval('...')
this['eval']('...')
// direct eval calls
eval('...')
(eval)('...')
with({ eval: eval }) eval('...')
```

Figure 4: Examples of direct and indirect `eval`'s.

references, while properties added or owned by other principals are replaced by proxies. (We don't have to do such replacements for user-defined prototypes because they are already handled through proxies; it is the non-configurability of native prototypes that necessitates this special handling.)

4.3.4 *Handling* `eval`

ECMAScript defines four constructs to execute dynamic code: `eval`, `Function`, `setTimeout` and `setInterval`. Use of eval can either be *direct* or *indirect* [2], as illustrated in Figure 4 using examples. All of these instances of dynamic code, with the exception of direct eval, are to be executed in the global scope. JaTE wraps these instances using a function, while third-party code is given a proxy to this function. At the call-trap of this proxy, we first rewrite the code contained in the string argument to perform the transformations needed to ensure its safe execution. The rewritten string is then evaluated within the compartment of the currently executing principal.

Direct `eval` cannot be handled this way since it should not execute in the global scope, but in the same scope in which it appears. Therefore, we cannot use the method described above for handling indirect `eval`. We describe correct handling of direct `eval` below. Our evaluation shows that about 30.9% of Alexa top 500 websites use direct `eval`.

Almost all direct eval calls can be identified because they use the keyword `eval`. Our implementation transforms `eval(x)` into `eval(processEval(x))`[7]. Obfuscated, or unusual instances of direct `eval` may not be recognized by this approach. This is not a security threat since an unrecognized direct `eval` will to be treated as an indirect `eval`.

4.4 Supporting DOM Access

Normally, when a principal creates a JavaScript object, it receives a direct reference. However, this wouldn't be safe for DOM nodes, since they contain read-only built-in properties that can get to the global object and the whole DOM tree (e.g., `aNode.ownerDocument.defaultView` is `window` where `aNode` is a DOM node).

For the above reason, we ensure that only the host has direct access to DOM-nodes. Third-party code can access DOM-node creation operations only through proxies. As a result, all DOM nodes will have the host as the owner, and be accessed using proxies by third-party. Since object ownership information is no longer enough to tell who created a DOM-node, we record this information explicitly in a field, and call it *DOM-ownership*.

JavaScript code generated from HTML. Certain DOM-operations, such as the setting of `innerHTML` property and calling `document.write`, can generate new JavaScript code from HTML. It is necessary to parse HTML, identify script code, and rewrite it so that it executes in the same compartment as the principal invoking the HTML operation.

Malicious third parties can attempt to confuse our HTML parser with malformed HTML so that our parser does not recognize all the scripts that would be recognized and executed by the browser. We can rely on the solution used in Blueprint [35] for this purpose, namely, parsing HTML, filtering the parse tree, and then converting the parse tree directly into actual DOM nodes using safe DOM API calls.

5. IMPLEMENTATION

We implemented JaTE in Firefox 33. The implementation consists of (a) a Firefox extension that implements the JaTE network module, and (b) the JaTE script, written in JavaScript. When minified, this script is about 30KB in size. JaTE source has been released under GPL [36].

5.1 Use of Proxy

Use of shadow objects. To provide consistent semantics, ES6 proxies enforce several invariants within each trap handler. For example, a non-configurability invariant is enforced in the `get` trap to ensure that the return value is consistent for a frozen property. This prevents JaTE from creating a proxy to such a property. To work around this, instead of creating a proxy to an object O, JaTE creates a proxy to a *shadow object* [38] S that contains a reference to O. The traps on the proxy are set so as to access O. Since S does not undergo any modification, all invariants enforced by ES6 proxies will always be satisfied.

Fixing built-in functions. Proxy is still a new concept and Firefox 33 does not yet completely conform to the ES6 specification. For example, some `String` prototype functions such as `replace` and `match` that take a regular expression argument don't work if a proxy is supplied instead. To work around this problem, JaTE wraps such problematic functions to replace proxies with direct versions before calling the original function, and also creates proxies as needed for return values.

5.2 JavaScript rewriting

JaTE's rewriting requires static recognition of certain keywords. We can perform this safely because all dynamic code is analyzed and rewritten just before execution. By considering all formats of JavaScript comments, our rewriting is resilient to lexer confusing attacks [5].

Code undergoes three transformations: direct `eval` rewriting, `this` rewriting, and global function declaration rewriting [8]. These rewriting steps are efficient because they only require lexical analysis, plus maintaining the current parenthesis nesting level, as opposed to more extensive transformations that require full parsing.

While rewriting, we introduce some identifiers, such as `processThis`, `processEvalSrc`, etc., to the source code. In the actual implementation, these identifiers are randomly generated with a safe length to avoid the possibility of colliding with the names used by third-party code.

5.3 Supporting EcmaScript 6

Even though JaTE was developed to confine ES 5 code, it can support new ES 6 features. Some require minor changes:

[7] We have shown a simplified transformation here. The full version, with additional security checks, can be found in Appendix 9.3.

[8] To support strict mode, we perform simple global variable declaration rewritings. More details can be found in Appendix 9.1.

for example, `let` statements need a new rewriting rule to convert them to `var` declarations if they are in the global scope. Other new constructs such as *Arrow Functions*, *Proxies*, and *WeakMaps* do not require changes to JaTE and are already supported.

6. EVALUATION

6.1 Performance Evaluation

6.1.1 Page Load Overhead

To calculate page load overhead, we developed a test extension for Firefox. The extension loads URLs sequentially from an input list, measuring the time it takes for the browser to emit the `load` event. The measurement is first performed 10 times without any JaTE components, and then repeated another 10 times with JaTE enabled. To avoid problems with network and caching, the extension disables caching and discards the load time for the first request of each site.

Social Media Widgets. Since JaTE mediates all security-relevant operations, it can support any policy. Although we leave the design of a flexible policy framework as future work, we have developed a suitable policy for our evaluation. The starting point for this policy is *one-way isolation* [16], which allows untrusted code to read or modify any data, but the modifications are visible only to untrusted code. We then tighten this policy to enforce confidentiality: all reads of primitive types return a "null" value. Specifically, the following rules are enforced:

- *traversable objects*: cross-compartment objects can be obtained but not modified or called. (Built-in functions can be called). This allows navigating the whole object graph.
- *primitive zeroing*: reading cross-compartment primitives always returns a default value, e.g., empty string.
- *global object shadowing*: property writes on the global object do not affect other principals. The updated value is only visible to the current principal.

We then relaxed this policy to support the functionality of Facebook's "Like"-button script. This script first creates a new global variable `FB`. Since this variable is not shared with other principals, the default *global object shadowing* policy is already permissive enough. The script then looks for two `DIV`s, one with id `fb-root` and one with class name `fb-like`, by looping through all DOM nodes using `document.getElementsByTagName('*')`. The default policy allows calling the built-in DOM functions and looping through the DOM nodes (*traversable objects*), but zeroes out their properties (*primitive zeroing*). Our policy relaxation is to avoid such zeroing and providing access to the two `DIV`s. Then, the script writes into them. Finally, the script inserts a new `script` tag and a new iframe, both of which are allowed by the default policy since they pose no security threats with JaTE. In summary, the default policy needs only a small change to allow write access to the two `DIV`s.

We used a process similar to that described above for Facebook "Like"-button to create policies for Google+, Twitter, etc. Much like the Facebook button, they also required write access to a small set of DOM nodes.

Figure 5 shows the overhead for the confinement of each button. The interception overhead dominates because it includes rewriting these rather large scripts, while the policy checks only need to approve the creation of a handful of DOM nodes. We used a blank enclosing (i.e., first-party) page for each button, so the overhead figures represent the worst-case. (A non-empty enclosing page would reduce the overall overheads because first-party scripts are not confined — and hence not slowed down — by JaTE.)

Widget	Size	Intercept. Over.	Policy Over.
Facebook	177kB	12.06%	0.44%
Google Plus	222kB	18.79%	2.50%
Twitter	361kB	11.77%	2.99%
StumbleUpon	15kB	8.02%	6%
LinkedIn	187kB	9.24%	1.17%
Average		11.97%	2.62%

Figure 5: Performance for Social Media Widgets

Website	Interaction	Delay	Overhead
Yahoo	Scroll Page	89.6ms	6.9%
Yahoo	Next news item	32.6ms	8.3%
YouTube	Scroll Page	73.6ms	7.4%
YouTube	Start a video	50ms	2.5%
Google	Instant search	50ms	1.3%
Google map	Panning	88ms	6.2%
Google map	Zooming in/out	202.3ms	15.0%
Amazon	Item details	15.1ms	3.0%
Amazon	Search suggestions	20.3ms	5.3%
Average			6.21%

Figure 6: User interaction overhead

Advertisements. In this experiment, we measured the overhead for confining advertisement scripts on Alexa's Top 500 websites. Since interception overheads dominate, we did not develop a specific policy for advertisements, but used an "allow-all" policy. To identify which scripts on a page are related to advertisement, we relied on a popular advertisement host list [1]. These scripts were confined by JaTE, while the remaining scripts were not confined. The average page load overhead was 19.5%.

6.1.2 User Interaction Overhead

We also measured the perceived overhead of JaTE on common user interactions, such as scrolling the page and moving to the next image in a gallery. These actions trigger one or more callbacks, which might schedule asynchronous callbacks of their own (e.g. making an HTTP request and evaluating the data when it has arrived).

To estimate the interaction delay, we leveraged the single-threaded nature of JavaScript, instrumenting all mechanisms used to register callbacks (e.g., `addEventListener` and `XmlHttpRequest`) to wrap the callback in a special function which stores its running time. Since only one callback is executing at a time, the sum of the running times of all callbacks is the total time spent executing code for the interaction. Adding this number to the time spent loading new network resources yields a reasonable estimate of the perceived user delay for the action. Figure 6 shows the delay in JavaScript execution and the total overhead perceived by the user. Since the network delay is unaffected by confinement and usually dominates, the overhead is quite small.

6.1.3 Rewriting Overhead

We assessed the performance of the rewriter by rewriting 6 common scripts. Figure 7 shows the time required to rewrite the scripts. Our rewriter is much faster than JSand's rewriter — JaTE's 58ms Vs JSand's 753ms for rewriting JQuery in 753ms. This is because their rewriting is signif-

Script	Size	Time
Google AdSense	22kB	37ms
Google Analytics	40kB	25ms
Google Maps	50kB	47ms
JQuery 2.1	83kB	58ms
Twitter "Share" Button	96kB	60ms
Facebook "Like" Button	160kB	101ms
Total	451kB	328ms

Figure 7: Rewriting overhead

Test	Type	JaTE	JSand
Blank Page	Page Load	169%	208%
JQuery	Page Load	219%	1230%
Google Maps	Page Load	98%	364%
Google Maps (Pan)	Interaction	6.2%	31%

Figure 8: JaTE vs JSand Overhead Comparison

icantly more complex than ours. But even JaTE's smaller overhead may be deemed significant, e.g., 100ms on Facebook "Like" button, and hence in our future work, we plan to implement it in C.

6.1.4 Comparison With Related Work

Comparison with JSand. We compared the performance of JaTE with that of JSand, a JavaScript confinement solution based on SES. To compare JaTE's and JSand's perfomance, we replicated JSand's benchmarks. Figure 8 shows the overhead for opening a blank page, loading the jQuery library, Google Maps and finally interacting with Google Maps. Two reasons for the difference in performance are the full parsing required by JSand during rewrite, which affects page load times, and its compatibility layer: their confinement setup makes all global variables local, which requires expensive global object synchronization.

Comparison with Caja. We also compared JaTE against Caja using a subset of the demos provided by the Caja authors. The chosen subset consisted of programs that could easily be benchmarked: a canvas clock, a markdown converter and a Game of Life. We modified the code for each demo to stop after a fixed amount of computations (e.g. 200 generations in Game of Life) and measured the average time required to complete the computation with Caja, JaTE and without any confinement to assess the overhead. For Caja, we tested both ES5/3 mode (compatible with ES3, uses rewriting to isolate code and a virtual DOM implementation) and ES5 mode (compatible with ES5, uses SES for isolation and the same virtual DOM implementation as ES5/3). Figure 9 shows the results; ES5/3 mode is slower than ES5 mode and JaTE because of its heavy runtime checks; Caja ES5 mode is faster than ES3/5 mode due to their use of SES (which realizes object capability without runtime checks), but still substantially slower than JaTE because of its virtual DOM implementation.

6.2 Transparency Evaluation

6.2.1 JaTE Transparency

There are three corner cases where JaTE can change the semantics of a script: (a) use of a cross-compartment `caller`, (b) special forms of direct `eval`, and (c) modification of white-listed built-in functions.

To assess the prevalence of these corner cases, we undertook a large-scale evaluation involving all sites from the

Benchmark	JaTE	Caja ES5	Caja ES5/3
Canvas Clock	16.8%	64.9%	1091%
Markdown	3%	136%	2310%
Game of Life	4.1%	566%	640%

Figure 9: JaTE vs Caja Overhead Comparison

Feature	Top 500
Strict mode error	87.4%
Use of direct eval	30.9%
Native prototype extension	48.5%

Figure 10: Related Work Transparency

Alexa Top 500. Using the same extension used to calculate page load overheads, we loaded each site, waited 5 seconds after the `load` event, took a screenshot and logged JavaScript errors, both with and without JaTE. To automate the inspection of a large number of sites and minimize false negatives, we compared the error logs and the screenshots of both runs for each site. If we found different error messages in the two logs, we inspected the screenshots side-by-side for missing content. If content appeared to be missing, we confirmed the test results manually. We did not find any page that could not be loaded correctly due to a shortcoming of our approach. Thus, we conclude that JaTE achieves transparency for today's web sites.

6.2.2 Related Work Transparency

To estimate the transparency of related work, we used the test extension again to load the same set of pages while confining all code in the *strict mode* subset used by Caja ES5 mode, SES and JSand. As shown in Figure 10, over 80% of sites use third-party scripts that break in strict mode, and hence these sites are not transparent with the aforementioned solutions.

Forcing strict mode is not their only shortcoming. For example, they also prevent the use of direct eval semantics and freeze native prototypes. To estimate the transparency impact of these two features, we ran our testing harness again and logged the use of these features in each web site, as shown in Figure 10. Both restrictions causes enough transparency problems to discourage websites from adopting these confinement solutions.

We also estimated the impact of *strict mode* on the social media buttons confined in Section 6.1.1. All the buttons failed to load.

6.3 Security Evaluation

To evaluate the security of JaTE, we tested it against a collection of attack vectors maintained by Google Caja [4], which contains 48 different attacks. 23 of these attacks are not applicable as they rely on non-standard features and do not work on Firefox. We augmented the test suite with 5 attacks of our own. These attacks either attempt to obtain unmediated access to cross-compartment references or introduce unconfined code into the page. For example, the `Function` constructor can be accessed through the `constructor` property of the prototype of `Number`, to create dynamic code, such as `(3).constructor.constructor("return window")`. We put these attacks into categories as shown in Figure 11. JaTE successfully stopped all other attack vectors, mediating all accesses and running dynamically generated code

Category	# of attacks
Prototype poisoning	4
Global object leak	3
Dynamic Code	7
Private data access and poisoning	4
Code obfuscation	3
`caller` and `arguments` stealing	3
Lexer confusing	2
Policy related	5

Figure 11: Caja Attack Vectors

using the correct principal.

7. RELATED WORK

In this section, we discuss previous related research, focusing our attention on efforts that have not already been discussed in detail.

7.1 Language-based isolation

ADsafe [11] and GateKeeper [13] define a subset of JavaScript amenable to static analysis to enforce policies using static verification. Gatekeeper [13] restricts JavaScript to perform static points-to analysis to reason about unreachability of security-sensitive resources. ADsafe [11] provides controlled DOM access to third-party code by offering a narrow interface through the `ADSAFE` object, while imposing significant language restrictions aimed at ensuring that all DOM interaction happens through the object. For example, ADsafe prevents access to `eval` and the use of subscript notation. Despite these restrictions, bugs were found [28] in ADsafe, demonstrating the difficulty of realizing object-granularity access control in JavaScript.

BrowserShield [29] was one of the earliest works in this area. It avoided language restrictions by relying primarily on runtime checking. They were the first to propose the idea of runtime rewriting to handle `eval` that we have used in JaTE as well. Caja [20] also relies heavily on rewriting and runtime checking. In particular, accesses to identifiers, attributes and functions need to be checked for safety, which can lead to slowdowns by an order of magnitude or more for some programs.

7.2 Frame-based isolation

AdJail [34] isolates third-party code in an `iframe` and uses `postMessage` to transparently cooperate with the first-party page. The advantage of this approach is that it is easier to reason about complete mediation, since every communication must explicitly pass through the `postMessage` primitive. Specifically, it sets up a shadow iframe containing third-party code and DOM data from the real page that was explicitly shared by the first-party. Any modification to the shadow DOM by the third-party code is transmitted to the real page and subject to a policy check before it is reflected there. Treehouse [14] is a conceptually similar approach using Web Workers instead of `iframe`s.

Instead of propagating DOM changes, Mashic [17] and Pivot [23] provide a transparent, synchronous interface for cross- domain operations on top of `postMessage`, to support confinement of general-purpose code. Mashic rewrites all code to continuation-passing style, while Pivot uses Generators to achieve the same goal using minor rewriting. However, they still fail to support complex interactions, such as pass-by-reference.

AdSentry's [12] goal is not only to fully mediate access to DOM resources, but also to protect against drive-by-downloads. To meet both goals, AdSentry runs third-party code on a separate JavaScript engine secured using Native Client sandbox [42]. DOM resources are kept in the main engine, and complete mediation is achieved by forwarding all DOM accesses from the shadow engine to the main engine.

MashupOS [41] criticizes the all-or-nothing approach of the SOP and extends it to better support the trust relationships commonly found in web mashups. It identifies four modes of interaction and introduces new HTML elements and security abstractions. On the other hand, COWL [32] leverages traditional mandatory access control and tracks the secrecy labels of each frame, preventing the leakage of confidential information to unauthorized parties. However, both MashupOS and COWL still only support coarse-grained policies; they don't tackle object-granularity access control that we seek in this paper.

The main problem with solutions in this category is that they are not able to support complex interactions involving passing object references or cross-frame function calls. As a result, to preserve functionality, people are taking risk to run third-party code directly in their websites.

7.3 Other

BEEP [15] allows a browser to examine and approve scripts before they are executed, according to a policy provided by the website as a JavaScript function. Content-Security Policies (CSPs) [31] are a mechanism developed by Mozilla to restrict the inclusion of resources such as scripts, images and frames into the web page to a specific subset of third-party servers. These works were motivated at preventing code injection attacks, e.g., cross-site scripting (XSS). Thus, their mechanisms are helpful for classifying entire scripts as "allowed" or "disallowed," but they don't help with the object-level isolation and access control problem faced by JaTE. Indeed, policy enforcement is not a promising approach for blocking XSS since the inferred origin of the malicious script would be the same as that of the first-party. This is why XSS defenses are mainly focused on detecting invalid script content, such as whole script [10] or partial script [27] content that has been reflected from HTTP parameters.

ConScript [22] augments Internet Explorer with policy check callbacks embedded directly in the JavaScript engine. Its goal is to securely mediate the operations made by a script, and apply a user-specified policy. WebJail [37] uses an approach similar to ConScript but implemented on Firefox. Its goal is to provide a higher-level interface to express policies that impose further restrictions over the SOP, e.g., restricting access to local storage, or network operations. The new HTML 5 specification [3] includes coarse-grained support for sandboxing `iframes` by specifying a subset of capabilities for the contained document, such as running JavaScript code or opening pop-up windows. While all of these techniques are helpful for further restricting untrusted scripts, note that they still only provide a single security context (such as a frame) for the code. In contrast, JaTE requires distinct security contexts to be maintained for the host and third-parties, and distinct policies to be enforced on them, while allowing them all to run within the same frame.

8. CONCLUSION

This paper presented JaTE, a compartment-based solution for confining third-party JavaScript code. Although this problem is of great practical significance, previous solu-

tions to this problem have not been amenable to real-world deployment because they impose significant restrictions that break existing sites, or due to performance considerations. In contrast, by leveraging JavaScript language features and using a novel combination of code transformation and runtime checking, JaTE can safely support the full JavaScript language and full interaction among principals. Our evaluation shows that JaTE is efficient at confining third-party code on a range of web sites. Finally, JaTE requires no browser modification, and thus provides an easy path for deployment on today's web sites.

The focus of this paper has been on the development of a policy enforcement framework. The next important challenge is the development of policies that achieve high-level security objectives, without requiring undue amount of human effort. An important advantage of JaTE is that it intercepts every security relevant operation, and these can be logged for subsequent analysis using techniques for policy generation from such logs [30, 33, 39, 43].

9. REFERENCES

[1] Ad blocking with ad server hostnames. http://pgl.yoyo.org/as/.

[2] Direct and indirect eval. http://perfectionkills.com/global-eval-what-are-the-options/.

[3] HTML5. http://www.w3.org/TR/html5/.

[4] JavaScript Attack Vectors. https://code.google.com/p/google-caja/wiki/AttackVectors.

[5] Lexer confusing attack. https://code.google.com/p/google-caja/wiki/JsControlFormatChars.

[6] New York Times Latest Victim of Malware Ad Injections. http://mashable.com/2009/09/15/new-york-times-malware/.

[7] Security advisory 201308013. https://code.google.com/p/google-caja/wiki/SecurityAdvisory201308013.

[8] Yahoo's malware-pushing ads linked to larger malware scheme. http://www.pcworld.com/article/2086700/yahoo-malvertising-attack-linked-to-larger-malware-scheme.html.

[9] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: Complete client-side sandboxing of third-party javascript without browser modifications. ACSAC '12.

[10] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *ACM WWW*, 2010.

[11] D. Crockford. ADsafe. http://www.adsafe.org, 2011.

[12] X. Dong, M. Tran, Z. Liang, and X. Jiang. AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements. ACSAC '11.

[13] S. Guarnieri and B. Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. USENIX SECURITY '09.

[14] L. Ingram and M. Walfish. Treehouse: JavaScript sandboxes to help web developers help themselves. USENIX ATC '12.

[15] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 601–610. ACM, 2007.

[16] Z. Liang, W. Sun, V. Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. *ACM TISSEC*, 2009.

[17] Z. Luo and T. Rezk. Mashic compiler: Mashup sandboxing based on inter-frame communication. CSF '12.

[18] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. S&P '10.

[19] S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *CSF*, CSF '09, Washington, DC, USA, 2009. IEEE Computer Society.

[20] M. S. Mark S Miller, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, 2008.

[21] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of java. NDSS '10.

[22] L. A. Meyerovich and B. Livshits. ConScript: specifying and enforcing fine-grained security policies for JavaScript in the browser. S&P '10.

[23] J. Mickens. Pivot: Fast, synchronous mashup isolation using generator chains. S&P '14.

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

[25] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, 2006.

[26] 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. CCS '12.

[27] R. Pelizzi and R. Sekar. Protection, usability and improvements in reflected XSS filters. In *ACM ASIACCS*, 2012.

[28] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. USENIX SECURITY '11.

[29] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. OSDI '06.

[30] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *ACM SOSP*, 2003.

[31] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. WWW '10.

[32] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazieres. Protecting users by confining JavaScript with COWL. OSDI '14.

[33] W. Sun, R. Sekar, G. Poothia, and T. Karandikar. Practical proactive integrity preservation: A basis for malware defense. In *IEEE S&P*, 2008.

[34] M. Ter Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. USENIX SECURITY '10.

[35] M. Ter Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. S&P '09.

[36] T. Tran and R. Pelizzi. JaTE System. Available for download from http://www.seclab.cs.sunysb.edu/seclab/download.html.

[37] 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.

[38] T. Van Cutsem and M. S. Miller. Trustworthy proxies: Virtualizing objects with invariants. ECOOP'13.

[39] V. Venkatakrishnan, R. Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *Proceedings of the 2002 workshop on New security paradigms*, 2002.

[40] G. Wagner, A. Gal, C. Wimmer, B. Eich, and M. Franz. Compartmental memory management in a modern web browser. ISMM '11.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. SIGOPS '07.

[42] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. S&P '09.

[43] Y. Zhou and D. Evans. Understanding and monitoring embedded web scripts. IEEE S&P '15.

Appendix

9.1 Rewriting global strict code

For global strict code, the rewriting we use in Section 3 does not work because: a)`'use strict'` needs to be placed before any other statements and b) strict mode prohibits the use of the `with` statement. As a result, we employ a different way to rewrite code that uses strict mode in the global scope. In specific, we move all global variable declarations to the global scope while putting everything else inside a function scope where strict mode is enabled. This rewriting is also simple and only requires lexical analysis. Listing 1 shows how JaTE rewrites the malicious Like script when the script runs under strict mode in the global scope.

```
1  jate.enterContext("facebook.com");
2  var scope = jate.getScope("facebook.com");
3  try {
4   with(scope){
5    var s;var stol;
6    (function(){
7     'use strict';
8     s = function s() {
9      var stol=processThis(this).data.secret;
10    };
11    stol=data["se" + "cret"];
12    s();
13    stol=data.getSecret();
14    eval(processEvalSrc("stol=this.data.secret"
          ))
15   }();
16  }
17 } finally { jate.exitContext(); }
```

Listing 1: Rewritten malicious global strict-mode Like script (underlined code segments are added by JaTE)

9.2 Scope object

```
jate.createScope = function (domain, window) {
  var vWindow=jate.createProxy(window);
  var scope = new Proxy({}, {
    has: function(target, n){
      return true; // pretend to have all
          properties
    },
    set: function (target, n, value, receiver){
      vWindow[n] = value;
    },
    get: function(target, n, receiver){
      if (!(n in window)){
        throw new Error('Ref error');
      }
      else{
        return vWindow[n];
      }
    }
  });
  jate.allScopes[domain] = scope;
  return scope;
}
```

Listing 2: Scope object creation

The scope object is created as shown in Listing 2. JaTE does not use the virtual global object directly as the scope object because we want to correctly handle the `in` operator like `x in window` and reference errors of undeclared global variables.

9.3 Secure direct eval handling

An `eval` call is direct if the following conditions hold: (a) The `MemberExpression` in the `CallExpression` must be evaluated to a reference, not a value; and the identifier is "eval" within the `MemberExpression`; and (b) the reference must be evaluated to the standard builtin function `eval`.

```
// inside jate.createScope
...
var directEval = false;
var nEval = jate.getOrigEval(window);
var scope = new Proxy({}, {
  ...
  set: function (target, n, value){
    if (n==='directEval'){
      directEval = value;
    }
    ...
  }
  get: function(target, n){
    if (n==='nativeEval'){
      directEval = false;
      return nEval;
    }
    if (n==='eval' && window.eval === nEval &&
        directEval){
      directEval = false;
      return nEval;
    }
    ...
  }
}
...
```

Listing 3: Handling eval

Simply replacing `eval(...)` with `eval(processEvalSrc(...))` does not work because, at the `get` trap of the scope object, JaTE can't tell if a get request for "eval" is direct or indirect. For this reason, `eval(...)` is replaced with `(directEval=true, (eval===nativeEval)?(directEval=true, eval(processEvalSrc(...))):eval(...))`.

JaTE uses `directEval` to distinguish between direct and indirect `eval`. Setting `directEval` to `true` is used to let the `get` trap of the scope object know that this might be a direct `eval` call. In addition, comparing `(eval===nativeEval)` is used to make sure that `eval` in this execution context is the native built-in `eval`. Specifically, as shown in Listing 3, if a get request for "eval" is seen: JaTE checks if `eval` is actually the native built-in one and `directEval` is true, then returns the original built-in `eval` and resets `directEval` to `false`; otherwise process the get request as usual. As a result, all indirect `eval` requests will be proxified and the dynamic code is treated as new code from the same principal and set up to run in the same compartment.