Towards Fine-Grained Access Control in JavaScript Contexts

Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang
School of Computing
National University of Singapore

Xuxian Jiang
Department of Computer Science
North Carolina State University

***Abstract*—A typical Web 2.0 application usually includes JavaScript from various sources with different trust. It is critical to properly regulate JavaScript's access to web application resources. Unfortunately, existing protection mechanisms in web browsers do not provide enough granularity in JavaScript access control. Specifically, existing solutions partially mitigate this sort of threat by only providing access control for certain types of JavaScript objects, or by unnecessarily restricting the functionality of untrusted JavaScript. In this paper, we systematically analyze the complete access control requirements in a web browser's JavaScript environment and identify the fundamental lack of fine-grained JavaScript access control mechanisms in modern web browsers. As our solution, we propose a reference monitor called JCShadow that enables fine-grained access control in *JavaScript contexts* without unnecessarily restricting the functionality of JavaScript. We have developed a proof-of-concept prototype in the Mozilla Firefox browser and the evaluation with real-world attacks indicates that JCShadow effectively prevents such attacks with low performance overhead.**

I. Introduction

JavaScript is one of the most important components on the web platform. It enables a new generation of dynamic and interactive web applications, which commonly use JavaScript from various sources, such as third-party JavaScript libraries. Some of these third-party scripts are untrusted and can become malicious. To ensure web application security, it is critical to properly regulate accesses by JavaScript to web application resources. Web browsers control the accesses made by JavaScript using the same-origin policy (SOP) [30]. In SOP, an origin is defined as the triplet of *protocol type*, *host*, and *port*. SOP allows a piece of JavaScript to access an object only if the JavaScript and the object are from the same origin. Under SOP, loading untrusted JavaScript, such as JavaScript libraries and advertisement scripts, opens up the resources of the whole origin to the script. This is the root of a series of attacks.

To better understand the attacks and existing solutions, we need to understand the environment in which JavaScript is executed. A typical JavaScript environment includes three components: the *JavaScript engine*, the *JavaScript context*, and the *host objects*. The JavaScript engine executes JavaScript; The JavaScript context contains the objects defined by the JavaScript standard and the objects created in JavaScript code; the host objects are supplied by the hosting environment, such as the Document Object Model (DOM) [37] in web browsers. Under SOP, scripts from the same origin share a JavaScript context and host objects.

A common solution for protecting web applications against a malicious JavaScript component is to host it on a separate domain and isolate it in an iFrame in the web application. In this way, JavaScript in the third-party component is automatically isolated by browser's same-origin policy. However, the third-party component often needs to access host objects to obtain information from other parts of the web application. To address this problem, one type of solutions is to restrict the access from untrusted JavaScript [3], [15], [24], [28], [29], but they unnecessarily impede existing web applications that require a rich set of JavaScript features. For less restriction on JavaScript functionality, another type of approaches [18], [25] develops access control mechanisms to regulate the access to host objects. However, the access control to the objects in a JavaScript context is still not regulated.

In this paper, we systematically analyze the complete access control requirements in the JavaScript environment, and identify the fundamental lack of fine-grained JavaScript access control mechanisms in modern web browsers. More specifically, existing access control solutions in the JavaScript context are still too coarse-grained and are insufficient to mitigate the threats from third-party JavaScript. The overly restrictive policy that blocks a certain JavaScript feature affects normal functionality of legitimate web applications. For example, JavaScript allows functions to be overridden during execution, which is commonly used in web application toolkits to smooth out browser differences or fix browser bugs [39]. On the other hand, this very feature is being exploited by attackers to change the behavior of trusted JavaScript where it should be blocked. When one native object is overridden by a malicious script, all other scripts accessing that object in the same JavaScript context may be compromised [2], [6], [32]. Therefore, instead of imposing an all-or-none restriction, the JavaScript context needs a fine-grained access control mechanism to accommodate both security and functionality.

As our solution, we present JCShadow, a reference monitor that provides the desired fine-grained access control mechanism for JavaScript context protection. In essence, JCShadow partitions JavaScript objects in a

JavaScript context into multiple groups, and confines each group using a *shadow JavaScript context*. With the presence of a (shadow) context for each group, we can efficiently isolate one group from another and effectively regulate cross-group accesses with a security policy so that untrusted JavaScript can execute potentially dangerous JavaScript features without affecting trusted JavaScript from the same origin.

We have implemented a proof-of-concept JCShadow by extending the JavaScript engine of Mozilla Firefox. Our evaluation with real-world example attacks indicates that JCShadow can effectively block malicious JavaScript code from compromising benign JavaScript from the same origin. The capability of performing fine-grained access control on JavaScript objects is achieved with a low performance overhead. Moreover, our experience indicates that our solution is not limited to web browsers. Instead, it can be generally applicable to a variety of JavaScript environments that integrate JavaScript from different sources, such as bookmarklet-based tools and Firefox extensions.

To summarize, our paper makes the following contributions:

- We systematically analyzed the access control problem of a JavaScript environment and identified the common weakness of existing solutions in handling untrusted JavaScript, i.e., the lack of fine-grained access control mechanism for JavaScript context protection.
- We presented a novel solution called JCShadow. By effectively dividing JavaScript objects into different groups and providing each group with its own shadow context, JCShadow enables fine-grained access control in a JavaScript context.
- We demonstrated the effectiveness and practicality of our approach by implementing a JCShadow prototype in Mozilla Firefox 3.5. The evaluation with a number of example attacks confirmed its effectiveness and practicality.

The rest of this paper is organized as follows. Section II discusses the problem and existing research work, and illustrates the attack threat with an example. Next, Section III explains the detailed design of JCShadow and Section IV presents key implementation details in our Mozilla Firefox-based prototype. After that, Section V reports our evaluation results and Section VI examines possible limitations and suggests ways for improvement. Finally, Section VII covers the related work and Section VIII concludes this paper.

II. Background

In this section, we first introduce the JavaScript environment and classify existing research work that provides access control in the JavaScript environment. We then present a motivating attack example.

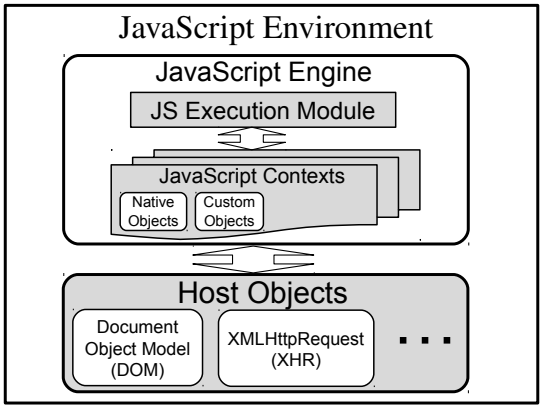

Fig. 1. Components in a JavaScript Environment.

A. Access Control in the JavaScript Environment

Figure 1 illustrates the components of a JavaScript environment. JavaScript runs in the execution module of the JavaScript engine, which has access to the JavaScript context and the host objects [14]. The JavaScript context contains two types of objects, *native objects* and *custom objects*. Native objects (a.k.a., built-in objects) are defined by the JavaScript standard, such as `Date`, `String`, and etc. Custom objects are defined by JavaScript code, including variables and functions. *Host objects* are objects provided by the hosting application (for example, the web browser) of the JavaScript engine for accessing peripheral resources outside the JavaScript engine, for example, DOM and network services. Therefore, for each origin, browsers create a JavaScript context and a set of host objects under the same-origin policy. JavaScript in one JavaScript context can access all objects in the same context, as well as those host objects from the same origin, but it is not allowed to access objects from other origins. Therefore, the access control in the JavaScript environment is on an all-or-none basis.

To address the lack of granularity in the JavaScript environment, researchers have proposed a number of systems. According to the way these systems handle various components in the JavaScript environment, we categorize them as follows:

- To prevent malicious JavaScript from accessing objects from an origin, a few projects recognize unwanted JavaScript and exclude it from the JavaScript environment [4], [13], [19], [22], [26], [31], [34]–[36].
- Another group of research work allows third-party JavaScript to be included in a JavaScript environment, but the functionality of third-party JavaScript is restricted [3], [10], [15], [29].

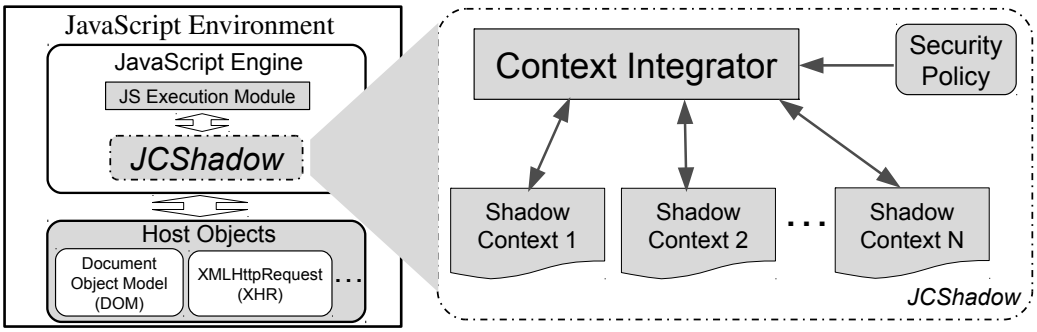

Fig. 2. Overview of JCShadow. It extends the JavaScript engine to support shadow JavaScript contexts.

- Other work provides fine-grained access control to host objects [18], [25].

Complete mediation to object accesses in a JavaScript environment involves fine-grained access control in both JavaScript context and host objects. However, none of existing approaches provides fine-grained access control in the JavaScript context.

B. Attack Example

We use an example to demonstrate the weakness caused by the coarse-grained access control in the JavaScript context.

```
<script
  src='http://untrusted.com/ulib.js'>
</script>
...
<script>
  result = function_in_ulib();
  ...
  if (location.href.toString()
        == "http://public.com") {...}
  ...
</script>
```

Fig. 3. An example of using a JavaScript library in web applications.

Third-party JavaScript libraries are commonly used in web applications. Figure 3 is an example of using JavaScript library in a web application, which shows scripts on a web page from `http://public.com`. It uses a JavaScript library `ulib.js` hosted on `untrusted.com`. On the page, there is another piece of JavaScript, which calls `function_in_ulib` in the JavaScript library `ulib.js`. This piece of JavaScript is shared among several pages and behaves differently on different pages. It checks its location through `location.href.toString()`.

According to the same-origin policy, the library `ulib.js` runs in the origin of the web page on `public.com`. Therefore, the script can access all resources of the page. The threat from the untrusted JavaScript can be partially mitigated by existing solutions to provide fine-grained access control to host objects. For example, it can disallow the untrusted script to modify the body of the web page. However, in practice, without fine-grained access control to the JavaScript context, either the JavaScript library is isolated from other scripts on the page, which breaks the page functionality, or full access to the JavaScript context is allowed, which is vulnerable to the following attack through the JavaScript context.

If the access to JavaScript context is fully allowed, code in the untrusted JavaScript library `ulib.js` can override the native `toString` function of the `String` object with its own function:

```
String.prototype.toString=function(){
  // code to add the malicious AD content
}
```

When the trusted script calls the `location.href.toString()` function to retrieve the location of the web page, its call to `location.href.toString()` is answered by the function defined by the untrusted JavaScript library, which for example, can add malicious contents to the page. In this way, attackers can bypass the access control to host objects and breaks the integrity of web applications.

III. Our Approach

The goal of JCShadow is *to provide fine-grained access control in JavaScript contexts.* To achieve this goal, JCShadow mediates all accesses to objects in the JavaScript context, and prevents malicious JavaScript from affecting the integrity of other JavaScript running in the same JavaScript environment. We stress that JCShadow's goal is to provide access control to objects in JavaScript contexts. It complements other solutions [18], [25] that develop fine-grained access control to host objects of a JavaScript environment.

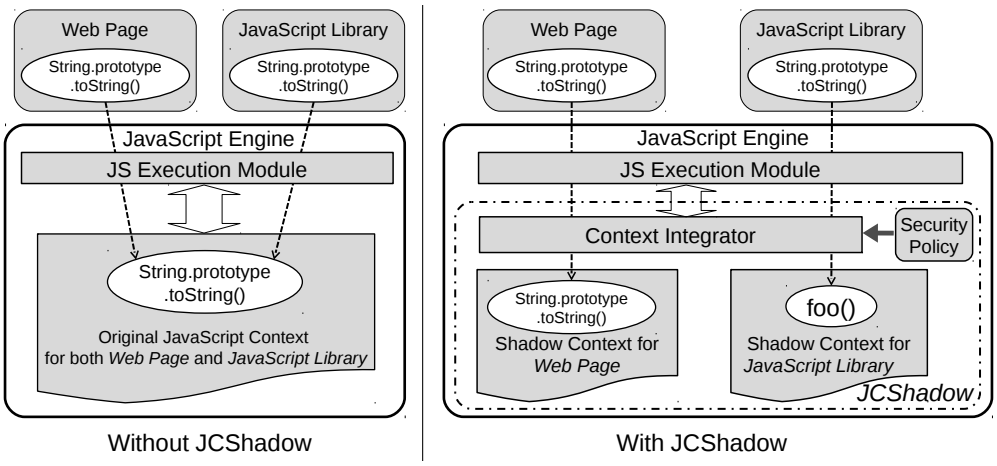

Fig. 4. Illustration of the scenario when an included JavaScript library attempts to override the `toString` function.

A. Overview of JCShadow

Figure 2 shows the design of JCShadow. It extends the JavaScript engine to mediate accesses to objects in the JavaScript context. Rather than simply allowing or denying access to JavaScript objects, JCShadow creates *shadow contexts* to support less restricted JavaScript functionality. A shadow context is an isolated copy of the JavaScript context, which is associated to selected JavaScript in the JavaScript environment. The *context integrator* integrates shadow contexts according to user-specified security policies and presents a single view of JavaScript context to JavaScript in the execution module. It intercepts access to the JavaScript context, and decides which shadow context should be accessed. By default, JavaScript associated with one shadow context is only allowed to access objects in its own shadow context. A permissive security policy would allow JavaScript to have regulated access to objects in other shadow contexts.

JCShadow divides JavaScript running in a JavaScript environment into groups and assigns each group a shadow context. Users or developers can group JavaScript by the trust they have toward the sources of JavaScript. For example, mashup web sites may have gadgets from long-term collaborator domains in one group, and other newly included gadgets in another, or they can put gadgets from each different domain in a separate group to prevent them from affecting one another.

B. Shadow JavaScript Contexts

Stemming from ECMAScript [14], JavaScript code is executed in *execution contexts*. In browsers, a unique JavaScript context is typically assigned for each web page, including those embedded in iFrames, to prevent scripts of different web pages from affecting each other. Each JavaScript context contains custom and native objects, and it also provides interfaces to host objects provided by the hosting application, such as DOM objects and XMLHttpRequest.

JavaScript contexts are isolated from each other to enforce the all-or-none same-origin policy: JavaScript code running in one context is unable to access objects defined in other contexts; However, once a piece of JavaScript is allowed to execute in a context, it has full control of all objects in that context. To provide finer granularity of access control to JavaScript contexts, JCShadow creates *shadow contexts* to further isolate JavaScript in the same JavaScript context.

Each shadow context is conceptually a dedicated subset of the original JavaScript context with all of the objects created or updated in the corresponding JavaScript group. As a result, objects in the original JavaScript context are now separated into different shadow contexts according to the way JavaScript is grouped. Native objects accessible in the original JavaScript context are provided for each derived shadow context. Custom objects created by the JavaScript are created only in the shadow context of the JavaScript to which it belongs.

Figure 4 illustrates the idea of shadow contexts using the JavaScript library attack example described in Section II. The attack scenario is illustrated on the left hand side. Without JCShadow, the untrusted JavaScript library and the web page share the same JavaScript context, and thus the JavaScript on the web page can be affected by the JavaScript library. The result of JCShadow is illustrated on the right hand side of Figure 4. JCShadow creates separate JavaScript shadow contexts for the trusted JavaScript in the web page and the untrusted JavaScript library. When the `toString` native function of the `String` object is modified by the JavaScript library by overwriting it with the custom function say,

`foo`. A new `toString` function object is created in the library's shadow context, which is assigned to the `foo` function. To the trusted JavaScript on the web page, the `toString` native function object referred to remains unchanged. Note that our solution allows JavaScript to override functions, but the overridden `toString` function is only visible to JavaScript in its own shadow context. Later, when the trusted JavaScript invokes the `location.href.toString()` function, the context integrator redirects the invocation to the original function that would return `http://public.com`. Therefore, JCShadow prevents the execution of trusted JavaScript from being affected by the malicious behavior of the untrusted JavaScript library.

Isolating JavaScript execution in shadow contexts. The main challenge in achieving JCShadow's confinement is from the dynamic behaviors of JavaScript. JavaScript can create new custom objects or redefine existing native or custom objects during its execution. JCShadow should confine objects inside the corresponding shadow context to prevent breaches.

New JavaScript objects can be created indirectly, for instance, by generating a new <script> element on the web page through the DOM interfaces, or adding a new `onclick` event handler to an existing element. JCShadow needs to track these new scripts and associate them to the corresponding shadow contexts where they are created; otherwise, untrusted JavaScript would easily escalate its privilege by injecting code into a shadow context with higher privilege via DOM interfaces. JCShadow monitors DOM element creation and attribute addition that introduce new JavaScript, and associates the newly added JavaScript to the shadow contexts of their creator JavaScript. JavaScript can also dynamically redefine native or custom objects. JCShadow handles object redefinition in a similar way as object creation.

C. Context Integrator and Security Policies

JCShadow provides a framework for fine-grained access control in the JavaScript context. After intercepting the access to JavaScript objects, JCShadow passes the access to the context integrator, which in turn integrates shadow contexts to serve the request. In this way, JCShadow supports controlled object sharing between shadow contexts, a unique feature that enables less restrictive JavaScript confinement.

As a reference monitor, JCShadow mediates all types of *accesses* to objects in a JavaScript context, namely, read, write, and invoke operations. The read operation is to read a value or a property of an object. The write operation is to create a new object, to assign a value to an existing object, or to define or redefine a function object. The invoke operation is to invoke a JavaScript function. Note that parameter passing in function invocations also results in read operations.

Access Type	Source Trust	Dest. Trust	Action
Read, Invoke	Low	High	Disallow
Read	High	Low	Allow
Invoke	High	Low	Degrade trust
Write	Low	High	Allow-In-Isolation
Write	High	Low	Allow

TABLE I
SECURITY POLICY 2 FOR HOST AND CUSTOM OBJECTS

If the requested object exists in the requester's own shadow context, the context integrator directly allows the access. Otherwise, the context integrator uses security policies to choose objects in the shadow contexts. An access request is in the form of <*Source, Destination, AccessType*>. The security policy specifies actions for the access based on the properties of the elements in the request. For example, when JavaScript in a shadow context i requests an access to a custom object o defined in another shadow contexts i', the context integrator can abort the access, create a new object in the shadow context i, or allow the access to o.

Next, we present sample security policies to demonstrate how JCShadow prevents different attacks. These security policies focus on protecting the execution integrity of trusted JavaScript against various threat scenarios and separating the code privilege according to shadow contexts.

1) Sample security policy 1, Horizontal isolation: When multiple JavaScript libraries are included into the same web page, they may cause conflicts to each other if they override JavaScript objects shared by other scripts on the page. With this policy, JCShadow ensures that the scripts cannot interfere with one another.

Under this policy, JCShadow allocates a shadow context to each JavaScript library. When the JavaScript in one shadow context attempts to add new properties to native or custom JavaScript objects or to override certain objects, the modification occurs locally in its own shadow context and is not visible to other shadow contexts.

Therefore, each script running in one shadow context uses its own native and custom objects. This security policy isolates the execution of different JavaScript libraries and does not allow any sharing of native or custom JavaScript objects between them.

2) Sample security policy 2, Ring-based access control in JavaScript context: ESCUDO [18] uses a ring-based access control framework to regulate accesses to host objects. By default, all scripts in one web page run in the same JavaScript context and have access to all host objects of that page. To prevent untrusted JavaScript included in a web page from arbitrarily accessing the host objects, ESCUDO [18] divides the web page into rings according to the trust levels on page elements.

JavaScript in one region can only access regions with higher ring numbers, i.e., lower trust. ESCUDO provides fine-grained access control to host objects. This security policy of JCShadow complements ESCUDO to provide similar fine-grained access control to the JavaScript context.

The policy is shown in Table I. The JavaScript on a web page is assigned to rings based on its trustworthiness. The policy decides the action by the type of operation, the trust of the source JavaScript, and the trust of the destination object. Generally speaking, this policy only allows JavaScript with higher trust to access objects with lower trust. JavaScript with lower trust is not allowed to read objects with higher trust or invoke functions with higher trust. However, if the JavaScript with lower trust overwrites objects with higher trust, JCShadow allows the write operation in the JavaScript's own shadow context ("allow-in-isolation"). Read and write operations from scripts with higher trust to objects with lower trust are allowed. We need special treatment for invoking lower trust functions. To prevent lower trust function from accessing the JavaScript context with higher trust, the invoked function will run with temporarily degraded trust, i.e., in the functions own shadow JavaScript context. The parameters to the function will be explicitly copied into the function's shadow context. By allowing function invocations with degraded privilege, it allows legitimate use cases in the mashup scenario where JavaScript with higher trust may need to invoke the function of lower trust. For example, the integrator (i.e. hosting web page) may invoke the function of a third-party gadget integrated in the web page to initialize it. With privilege degradation for function invocations, this security policy supports legitimate functionality.

With this policy, JavaScript with lower trust is not allowed to modify the functions defined in JavaScript with higher trust. Unless JavaScript with higher trust intentionally reads lower trust data and turns them into code via `eval`, etc., this policy guarantees the integrity of JavaScript with higher trust and enforces the ring-based policy as well. For this reason, this security policy currently disallows `eval`.

IV. IMPLEMENTATION

We implemented JCShadow in the Mozilla Firefox version 3.5 by extending its JavaScript engine TraceMonkey. In TraceMonkey, `JSRuntime` is the top-level object that represents an instance of the JavaScript engine. The JavaScript context, `JSContext`, is a child of JSRuntime. In Firefox, JavaScript from the same origin runs in an instance of `JSContext`. `JSContext` contains a global object, which holds pointers to all variables and functions that are available to the JavaScript code.

For example, the `toLowercase()` function is a property of the `String` class, which is in turn the property of the global object. As a script runs within a `JSContext`, all the newly created global functions and variables will be added as properties of the global object of that `JSContext`.

A. Extending TraceMonkey

JCShadow extends TraceMonkey with three main components: assigning shadow contexts to JavaScript, tracking JavaScript in shadow context, and enforcing the access control policy. JCShadow relies on web application developers to divide the web page into regions and assign each region with a shadow context ID. JCShadow keeps track of the JavaScript shadow context ID of the script running in JavaScript engine and uses it to enforce policies. JavaScript interpreter in TraceMonkey has 234 different bytecode for JavaScript operation, for example, bytecode `JSOP_DEFFUN` for function object creation and bytecode `JSOP_DEFVAR` for variable creation. JCShadow intercepts accesses to objects in TraceMonkey's interpreter by intercepting object creation, function invocation, and object property `setter`/`getter` operations.

JCShadow marks it with the shadow context ID of the JavaScript when an object is created by JavaScript. It also intercepts the object property `setter` and `getter` functions. In Firefox, all global functions and variables are properties of the *global object*. If a script modifies an object, it actually modifies the property of the object's parent object, which will eventually trigger the set property operation in the JavaScript engine. In the property `setter`, if the operation is to modify an object that belongs to a different shadow context, the context integrator consults security policies to decide the appropriate operation. In the property `getter`, JCShadow consults security polices when JavaScript attempts to get the value of an object. JCShadow also instruments the function invocation in the JavaScript engine. Function invocations are handled in a similar way as the object property getter by the context integrator.

B. Dynamic Script Introduction

Assigning the correct shadow context IDs to dynamically generated JavaScript is critical in JCShadow. Otherwise, JavaScript web pages may easily escalate their privilege by injecting code into DOM elements. Hence, JCShadow assigns dynamically generated JavaScript the shadow context ID of the script that creates it. In practice, third-party JavaScript makes use of DOM interface functions such as `setAttribute`, `createElement`, etc. to attach a piece of JavaScript code into an HTML element event handlers (for example, `<a onclick='maliciousfun()'/>`) or to create new script element dynamically.

We implement attribution to the original JavaScript by simulating the call stack functionality. We intercept the JavaScript engine before it invokes DOM functions, and record the current shadow context ID. After the DOM function call returns, the shadow context ID is cleared. We also intercept functions that may bring in new JavaScript, and checks whether the JavaScript shadow context ID is set. If so, and if the DOM function is a call from user JavaScript, we assign the newly generated JavaScript with the shadow context ID stored.

C. Configuration Files of Shadow Contexts and Security Policies

JCShadow uses XML-based configuration files to specify the grouping of JavaScript on web pages, assign shadow context IDs to them, and specify security policies. We also implemented a GUI tool that is integrated into Firefox preferences dialog to help users to specify configurations. The configuration files can also be specified by web developers. In contrast to using markups in a web page, the external configuration files make it flexible to adapt the context ID assignment to JavaScript environments beyond web applications, such as Firefox extensions.

V. Evaluation

We evaluated JCShadow with real-world examples to ensure that it is able to protect JavaScript context. We also measured the runtime overhead of JCShadow, and tested its compatibility with real-world websites. Our experiments were performed on a computer with an Intel Core 2 Duo 2.33GHz and 4GB RAM, running Ubuntu 9.10.

A. Effectiveness

We tested JCShadow using a web mashup example, as well as attacks to bookmarklets where native JavaScript functions were overwritten by malicious web pages.

`Bookmarklet Attack`: A bookmarklet is a piece of JavaScript saved as a bookmark entry in the browser's bookmark menu. When activated by a user click, the JavaScript code in the bookmarklet executes in the JavaScript context of the current web page. MashedLife [1] is a bookmarklet-based password manager, providing an online password management service that helps users remember passwords of web sites. Users need to first store their passwords on the MashedLife's server. If they want to log into some web site, say `www.example.com`, they invoke MashedLife's bookmarklet, whose JavaScript retrieves the password stored in MashedLife's server and fills in the log-in information to the web page.

More specifically, MashedLife's bookmarklet first checks the current web page's location by calling the JavaScript function `location.href.toString()`. The location is used to construct a request for a piece of external JavaScript from the MashedLife server, which fills the user's log-in information (user name and password) encoded in the script.

An attack on the bookmarklet has been reported in Adida et al. [2]. In the attack, a malicious web page at `http://www.malicious.com` overrides the native `toString` function of the `String` object with its own function:

```
String.prototype.toString = function() {
  return "https://www.example.com";
}
```

When MashedLife's bookmarklet checks the location of the malicious page, its call to `location.href.toString()` is answered by the function defined on the malicious page, which returns `https://www.example.com` instead of the actual location `http://www.malicious.com`. The bookmarklet then retrieves the user's password of `example.com` and fill it into the malicious page, which is accessible by the attacker.

With JCShadow enabled, using the security policy 2, the web page and bookmarklet ran in different shadow contexts, say `WPC` and `BMC` respectively. We assign higher trust to the JavaScript in the bookmarklet's shadow context. Scripts on the web page are allowed to override native JavaScript functions in the shadow context `WPC`. Specifically, the web page script with a shadow context ID of `WPC`, was allowed to replace the `toString` function object with its custom function in `WPC`. When bookmarklet invoked `toString` function to retrieve the location string of the web page, JCShadow intercepted the function invocation. Because the call was made from the more trusted BMC, JCShadow invoked native function object provided by JavaScript environment to `BMC`.

Similarly we used this security policy to successfully prevent malicious web mashup attack described in Section II.

B. Compatibility

To evaluate the compatibility of our sample policies, we tested JCShadow on 25 top web sites listed by Alexa [8] with our security policies. We locally cached the copies of these websites and assigned different shadow context IDs for web page scripts and third-party JavaScript libraries. We whitelisted content distribution networks (CDN) and sub-domains of web sites to run scripts from sub-domain and CDN in the same shadow context with the original web sites.

The security policy 1 is compatible with 16 websites out of 25 web sites evaluated. The broken functionality was because these web pages have interactions

Operation	Unmodified Browser (ms)	Time With JCShadow (ms)	Time With Web Sandbox (ms)	JCShadow Overhead (%)	Web Sandbox Overhead (%)
Function Invocation	8.4808	11.0522	95.153	30.32%	1021.99%
Get Object Member	5.4962	7.9173	94.6788	44.05%	1622.62%
Set Object Member	7.128	10.0705	95.1626	41.28%	1235.05%
Invoke Object Member	9.3994	12.6262	95.488	34.32%	915.9%

TABLE II
PERFORMANCE OF OUR SOLUTION FOR BASIC OPERATIONS. TIME IN FIRST THREE COLUMNS ABOVE IS MEASURED IN MILLISEC.

between scripts on the page and scripts in the libraries. For example, Google Analytics was included into web applications in the forms of both inline page JavaScript and external JavaScript. As a policy supporting sharing across shadow JavaScript contexts, the security policy 2 of the JCShadow is compatible with 19 of the 25 web sites in our experiments. In this test, we assigned scripts from the web page with high trust, and assigned third-party scripts with low trust. The broken functionality is caused by the use of `eval`, which was disabled by security policy 2. `Eval` were used to deobfuscate JavaScript. If `eval` is permitted for this usage, the sample security policy 2 was compatible with all 25 web sites. We discuss this limitation and its solution in Section VI.

C. Performance Overhead

Overhead incurred on real-world web sites. We measured the performance overhead of JCShadow on the 25 different web sites top-listed by Alexa. We locally cached the copies of these websites and assigned shadow context IDs for third-party scripts. In the JavaScript engine we measured the total time taken by the execution of all JavaScript in a web page (averaged on five runs). We noted that the overhead ranged from 0.44% to 13.45%, averaged at 3.65%.

JavaScript Runtime Overhead. JCShadow performs access control check on objects in the JavaScript engine. We measured the performance overhead incurred by JCShadow in JavaScript engine with basic operation tests and industry-standard benchmarks.

a) Basic Operation Overhead: In JCShadow, we interposed on object property `getter` and `setter` functions, as well as function object creation and invocation points in the JavaScript engine. In the experiment, we measured the performance overhead incurred by these basic operations. Table II shows the performance overhead (average computed on five runs) for basic operations against user-defined objects used in a loop for 10,000 iterations. We did this experiment to estimate the worst-case overhead of JCShadow and compared it with the translation-based WebSandbox [15]. JCShadow has a much lower overhead of 30 to 40 percent, while WebSandbox's overhead ranges from 900 to 1600 percent.

Benchmark Name	Original Browser (runs/sec)	JCShadow (runs/sec)	% Overhead
Dromaeo	9.386	9.06	3.60%
SunSpider	15.938	15.15	5.20%
V8 Test Suite	2.496	2.358	5.85%

TABLE III
OVERHEAD INCURRED BY JCSHADOW ON INDUSTRY-STANDARD JAVASCRIPT BENCHMARK

b) JavaScript Industry Standard Benchmark: We also measured the overhead of JCShadow on industry-standard benchmarks. Table III shows the results. On the Mozilla Dromaeo JavaScript benchmark, we observed a 3.60% performance overhead; On SunSpider JavaScript benchmark, we observed a 5.20% performance overhead; And on V8-test suite, we observed a 5.85% slowdown. As these numbers indicate, JCShadow incurs low performance overhead on industry-standard benchmarks. The current prototype of JCShadow uses a suboptimal search operation when searching for variables in shadow contexts, which has room for further improvement.

VI. LIMITATION AND DISCUSSION

Native-code JavaScript Engines. The TraceMonkey JavaScript engine in Firefox 3.5 compiles JavaScript code into bytecode, and interprets the bytecode. It also has a JIT feature that compiles JavaScript code into binary code to speed up JavaScript execution. Specifically, each time the interpreter interprets a backward-jump bytecode, TraceMonkey notes the number of times the jump target has been used. This number is called the *hit count*. It the hit count reaches the threshold, it is considered *hot*. TraceMonkey prepares traces for hot code and compiles it into native code. The native code of the traces needs to call JavaScript engine C/C++ functions to read and write to `JSObject` in the TraceMonkey's heap. However, in TraceMonkey tracing is enabled only for basic operations such as arithmetic operations. Advanced features, such as getters and setters, are not traced. Therefore, JCShadow is not affected by the JIT feature in TraceMonkey. The JavaScriptCore JavaScript engine in WebKit and the V8 JavaScript engine in Google Chrome compile JavaScript directly into native code to speed up JavaScript execution. Our next step is to extend JCShadow to compilation-based JavaScript engines through binary rewriting or compiler instrumentation.

Supporting `eval` via Information Flow Tracking. The current implementation of JCShadow cannot determine whether the parameter to `eval` is affected by information from low-trust objects. Information flow tracking techniques, such as in SIF [7], are needed to achieve better accuracy. With such improved information flow tracking ability, JCShadow will support `eval`. We leave it as the future work.

VII. Related Work

Access control in JavaScript environment. The research work closely related to JCShadow is the approaches that improve access control granularity in the JavaScript environment. ESCUDO [18] develops a new web application protection model providing mandatory access control. It enables fine-grained access control to the JavaScript host objects in web applications. Louw et al. [25] provide a policy enforcement framework for the JavaScript in Firefox extensions. It controls extensions' access to XPCOM services, network services, browser's password manager, etc. These two approaches provide fine-grained access control to the *host objects* of their JavaScript environment. In contrast, JCShadow develops fine-grained access control to the JavaScript context, which can be combined with the above two approaches to completely mediate accesses in a JavaScript environment.

Object Views [27] creates object proxies, called views, to support sharing in a browser JavaScript environment. This approach selects objects to form a view, and uses an aspect system to support fine-grained sharing. Compared to JCShadow, the advantage of Object Views is the support of secure cross-origin sharing, while JCShadow has the flexibility of integrating multiple shadow contexts within the sample origin. CONSCRIPT [28] introduces fine-grained access control to JavaScript by an aspect-oriented approach. It allows security advice to be executed before a JavaScript function is called or before a piece of JavaScript is executed. The granularity of access control in CONSCRIPT is at the whole-script and function level, compared to JCShadow's access control to each JavaScript object access. JCShadow's also allows less restrictive policy through the "allow-in-isolation" action. Phung et al. [33] associate aspects with JavaScript through JavaScript function wrapping, but it is not suited to be used in a hostile environment.

JavaScript Subset Various object-capability solutions exist that try to restrict the excessive power of the JavaScript language with a safe subset of the original language. ADsafe [3] is a static verifier that removes unsafe JavaScript features such as global variables, `this` pointer, and `eval`. Caja [12] dynamically translates a web application, perform static analysis to verify security properties, and adds runtime checks. Web Sandbox [15] uses dynamic translation to run untrusted JavaScript code in the virtual machine that provides an isolated environment. The virtual machine interacts with the JavaScript and DOM modules of the browser, and uses the policy rules to mediate runtime checks. FBJS [10] creates a separate virtual scope for every application running in Facebook. These solutions aim at preventing embedded third-party malicious JavaScripts from compromising other portions of web applications. In contrast, JCShadow provides less restrictive environments to execute untrusted JavaScript.

Untrusted JavaScript Prevention and Isolation Cross-site scripting (XSS) attack prevention has been extensively researched [4], [13], [19], [22], [26], [31], [34]–[36]. However, due to the lack of access control granularity in existing JavaScript context, these solutions aim to prevent malicious JavaScript from being executed in the web application.

AdJail [24] isolates web advertisements using iFrames and builds a communication channel to pass around interactions between the isolated advertisement and the hosting page. AdJail focuses controlling the access to host objects, and completely isolates the access to custom objects between the isolated advertisement and the rest of the web page. Finifter et al. [11] propose a safe JavaScript subset that whitelist known-safe properties by separating the JavaScript namespaces used by different parties, which completely isolate untrusted JavaScript into different JavaScript contexts. Barth et al. [5] create "isolated worlds" for DOM objects to isolate access to DOM from different browser extensions.

Compared to this line of work, JCShadow aims to improve the control granularity inside a JavaScript context, so that the untrusted JavaScript can be allowed more permissively without threatening web application security.

New security primitives and policies Mash-up applications integrate various JavaScript from different origins into their own pages. Therefore, all integrated applications run in privilege of the integrator. Several recent proposals develop new primitives for web applications to adapt to mash-up applications [9], [17], [21], [38]. Extensions [16], [20], [23] to the same-origin policy have also been proposed to allow better separation of JavaScripts even if they are from the same origin. Compared to JCShadow, the granularity of access control is coarse, and they do not support dynamic access control policies.

VIII. Conclusion

Web applications usually integrate JavaScript from various sources, which have different levels of trust, but existing solutions do not provide fine-grained access control to protect trusted resources in a JavaScript

context. In this paper, we systematically analyzed the access control problem in a JavaScript environment and identified the common weakness of the existing solutions in handling untrusted JavaScript: the lack of access control granularity in a JavaScript context. We developed a reference monitor called JCShadow, which isolates JavaScript using shadow JavaScript context and regulates each object access in a JavaScript context. We implemented JCShadow in the JavaScript engine of Mozilla Firefox. Our evaluation and measurement demonstrated the effectiveness and efficiency of JCShadow.

Acknowledgments We thank Wenliang Du for his insightful feedback on the draft. We also thank the anonymous reviewers for their valuable comments. This paper is supported in part by an NUS Young Investigator Award R-252-000-378-101.

REFERENCES

[1] Mashedlife. http://mashedlife.com.

[2] B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript Environments. In *the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[3] ADSafe. Adsafe. http://www.adsafe.org/.

[4] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *the IEEE Symposium on Security and Privacy*, 2008.

[5] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *the Network and Distributed System Security Symposium (NDSS)*, 2010.

[6] B. Chess, Y. T. O'Neil, and J. West. Javascript hijacking. Technical report.

[7] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *the USENIX Security Symposium*, 2007.

[8] A. T. W. I. Company. Top sites by category. http://www.alexa.com/topsites/category.

[9] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *the ACM conference on Computer and Communications Security (CCS)*, 2008.

[10] Facebook. FBJS - Facebok Developers Wiki. http://wiki.developers.facebook.com/index.php/FBJS, 2008.

[11] M. Finifter, J. Weinberger, , and A. Barth. Preventing capability leaks in secure javascript subsets. In *the Network and Distributed System Security Symposium (NDSS)*, 2010.

[12] Google. Caja. http://code.google.com/p/google-caja/.

[13] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *the Network and Distributed System Security Symposium (NDSS)*, 2009.

[14] E. International. Standard ECMA-262. http://www.ecma-international.org/publications/standards/Ecma-262.htm, 2009.

[15] S. Isaacs and D. Manolescu. WebSandbox - Microsoft Live Labs. http://websandbox.livelabs.com/, 2009.

[16] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *the International World Wide Web (WWW) Conference*, 2006.

[17] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *the International World Wide Web (WWW) Conference*, 2007.

[18] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A Fine-grained Protection Model for Web Browsers. In *the International Conference On Distributed Computing Systems (ICDCS)*, 2010.

[19] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *the International World Wide Web (WWW) Conference*, 2007.

[20] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *the ACM conference on Computer and Communications Security (CCS)*, 2007.

[21] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *the International World Wide Web (WWW) Conference*, 2008.

[22] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *the ACM Symposium on Applied Computing*, 2006.

[23] B. Livshits and U. Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2007.

[24] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *the USENIX Security Symposium*, 2010.

[25] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. In *Journal in Computer Virology*, 2008.

[26] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *the IEEE Symposium on Security and Privacy*, 2009.

[27] L. A. Meyerovich, A. P. Felt, and M. S. Miller. Object views: Fine-grained sharing in browsers. In *the International World Wide Web (WWW) Conference*, 2010.

[28] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *the IEEE Symposium on Security and Privacy*, 2010.

[29] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - Safe Active Content in Sanitized JavaScript. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf, 2007.

[30] M. Mozilla. Same origin policy for javascript. https://developer.mozilla.org/En/Same_origin_policy_for _JavaScript, 2009.

[31] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *the Network and Distributed System Security Symposium (NDSS)*, 2009.

[32] S. D. Paula and G. Fedon. Subverting ajax. In *the Chaos Communication Congress (CCC) annual conference*, 2006.

[33] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2009.

[34] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[35] R. Sekar. An efficient black-box technique for defeating web application attacks. In *the Network and Distributed System Security Symposium (NDSS)*, 2009.

[36] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *the Network and Distributed System Security Symposium (NDSS)*, 2007.

[37] W3C. Document object model (DOM). http://www.w3.org/DOM/.

[38] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[39] J. M. Wilson. IE JavaScript bugs: Overriding internet explorer's document.getElementById() to be W3C compliant exposes an additional bug in getAttributes(), 2007. http://www.sixteensmallstones.org/ie-javascript-bugs-overriding-internet-explorers-documentgetelementbyid-to-be-w3c-compliant-exposes-an-additional-bug-in-getattributes.