CHALMERS | GÖTEBORG UNIVERSITY

Technical Report No. 2011:11

A Two-Tier Sandbox Architecture to Enforce Modular Fine-Grained Security Policies for Untrusted JavaScript

PHU H. PHUNG

Dept. of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY/
GÖTEBORG UNIVERSITY
Göteborg, Sweden, 2011

A Two-Tier Sandbox Architecture to Enforce Modular Fine-Grained Security Policies for Untrusted JavaScript

Phu H. Phung

Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Gothenburg, Sweden

Abstract

Existing approaches to providing security for untrusted JavaScript include isolation of capabilities – a.k.a. sandboxing. Features of the JavaScript language conspire to make this nontrivial, and isolation normally requires complex filtering, transforming and wrapping untrusted code to restrict the code to a manageable subset. The latest JavaScript specification (ECMAScript 5) has been modified to make sandboxing easier and more widely applicable. This is illustrated in a sandboxing library recently developed by the Google Caja Team which allows untrusted code to interact with a restricted API.

However, specifying and enforcing fine-grained policies within an API implementation is complex and inflexible, since each sandboxed application (there may be several within a single web page) may need an application-specific policy. In this paper, we present a two-tier architecture for sandboxed code which combines a baseline sandbox with a stateful fine-grained policy specified in an aspect-oriented programming style. The implementation of the fine-grained policy part is an adaptation of lightweight self-protecting JavaScript mechanism proposed by Phung at el (ASIACCS'09). This enforcement mechanism allows the policies can be defined in a modular way so that different policies can be specified and enforced for different untrusted applications. The mechanism is realized in JavaScript libraries so that it does not require a modified browser and untrusted code can be dynamically loaded and executed without run-time checking or transformation. We show the effectiveness of the mechanism by deploying some empirical case studies and analyzing their security features.

1 Introduction

Embedding external content into web pages is becoming more and more popular. A recent report [1] shows that 97% of Fortune 500 web sites display content from external partners using e.g. JavaScript widget providers, ad networks, or packaged software providers. Embedding external content into web pages is becoming more and more popular. A recent report [1] shows that 97% of Fortune 500 web sites display content from external partners using e.g. JavaScript widget providers, ad networks, or packaged software providers. Mashup web application is an example of this trend where a hosting page injects content generated by third-party services. As an example, `padmapper.com` is a web mashup-based application that combines information of apartments for rent from `craigslist.com` and the Google Maps to provide a map interface for

users looking for available apartments. Another common example of web mashup is web advertisement where an ad JavaScript code is embedded into a hosting page to interact with the page in order to display advertising content to the page. As reported in [1] 69% of Fortune 500 web sites use external JavaScript to render portions of their sites. Typically, a web master of a hosting page needs to trust the external JavaScript code before inserting to the page since the external JavaScript (mashup) code run in the context of the hosting page. However sometimes this trust can be misplaced in practice. In September 2009, readers of the New York Times website faced a fake virus infection pop-up which directs the readers to a web page that claims to offer anti-virus software. The attack happened because the external content, e.g. the ad, is normally fetched dynamically from an external source by the user's browser [23], which is not under the control of the hosting page. In such a context, a possible approach to ensure the safety of untrusted external content generated by untrusted script code is to execute the code within a sandbox environment and provides APIs for the untrusted code to function. However, isolating untrusted code into a compartment fails in current version of JavaScript (ECMAScript 3-ES3) since the language and its implementation in most browsers contain several features that can break attempts to isolate code. For example, there are various ways to get access to the global window object, e.g. executing `function attack(){return this;}; var mywindow = attack();` can get the reference to the global (`window`) object since the `this` object is implicitly coerced to the global object. Encapsulation leaks and prototype chains in ES3 are also dangerous features can break a compartment implementation[1]. Due to these vulnerabilities, isolation normally requires complex filtering, transforming and wrapping untrusted code to restrict the code to a manageable subset [12]. Examples of such approaches include BrowserShield [20], Facebook FBJS [5], ADsafe [2], and Google Caja [16].

In the *"strict mode"* of the new specification of JavaScript - ECMAScript 5 (ES5)[3], the existing vulnerabilities in ES3 are solved by restricting the dangerous features. This makes the ECMAScript 5 strict mode (ES5S) more secure and robust to construct a sandbox environment to isolate untrusted code *without static check or transformation*. Based on ES5S, the Google Caja Team has recently developed a client-side JavaScript library [4] which can be used to create a sandbox to load and execute untrusted code dynamically in which the untrusted code can only access the world outside this compartment through an API provided to it via the sandbox. The policy enforcement in this model thus relies on the implementation of the API itself. ADsafe [2] API, or Domita in Google Caja [16] are the examples of such API implementations that a sandbox could expose to its untrusted guest. However, these APIs enforce static coarse-grained policies. In ADsafe API or Domita examples, in order to enforce fine-grained and application-specific policies, the policies can be defined and embedded into ADsafe or Domita implementations. Such a mechanism is unpractical and inflexible for two mains reasons. Firstly, combining policy code into API implementation is too complex and thus error-prone. Secondly, for different untrusted applications (within one hosting page or in different pages), the policy code must be changed and therefore the API implementation must be modified for each specific application. These difficulties motivate a modular approach in which application-specific policies can be defined modularly and separately from an API so that they can be enforced for different portions of untrusted code in different hosting application scenarios.

In this work, we propose a two-tier architecture to enforce a stateful fine-grained policy on a baseline API running within a sandbox. The policies are specified in a modular way in an aspect-oriented programming style so that different policies can be specified and enforced for different untrusted applications. The implementation of policy enforcement is an adaptation of lightweight self-protecting JavaScript mechanism proposed by Phung at el [18]. Lightweight Self-

[1]These will be revisited in detail in Section 2.1

Protecting JavaScript [18, 13] is an alternative approach, without restricting the language nor transforming code, to sandboxing the behavior of JavaScript by interposing a security reference monitor to intercept built-in calls with policies.

We adapt the self-protecting JavaScript approach to policy specification and enforcement to the sandboxed code model to allow application-specific policies to be defined in a modular way for different applications. The enforcement architecture is two-tier in the sense that a baseline API for a sandbox is enforced by a modular policy which is performed within a sandbox environment, and then the enforced API is provided to untrusted code through another sandbox environment. In summary, the main contributions of this study are:

- *A declarative pattern to specify application-specific, and fine-grained security policies on API objects provided to untrusted code through a sandbox environment.* A policy is specified in an aspect-oriented programming style so that it can be defined modularly for different untrusted programs within a hosting page. Policies are defined in terms of method calls and property accesses, and are expressed in pure JavaScript so that a policy writer can easily express stateful policies.

- *A two-tier architecture to enforce stateful policies for untrusted code running within a sandboxed environment.* The policy enforcement on a baseline API is executed within a sandbox environment and therefore the policy code can only access the API and other functionalities provided by the sandbox. This guarantees that the policy code – however badly written – cannot unintentionally leak critical objects not accessible through the baseline API.

- *Empirical Studies* We have implemented a framework for constructing mediator objects for DOM access on a page. The framework is used to deploy some case studies for the enforcement architecture, including some third-party widgets and a context-sensitive advertisement web application. These case studies illustrate the effectiveness of the enforcement mechanism for untrusted script scenarios.

Organization The rest of this paper is written as follows. The next section reviews background including new features in ECMAScript 5 and briefly the implementation of Secure ECMAScript which is used to create a sandbox environment in this work. We also review the key points in lightweight self-protecting JavaScript and its limitations which are revised and adapted in this work. Section 3 proposes the two-tier enforcement architecture for policy enforcement of untrusted code on an API in sandbox compartments. In Section 4 we present our design for specifying fine-grained policies and policy enforcement implementation. Empirical studies are presented and discussed in Section 5. Conclusion and future work are given in Section 7.

2 Background

In this section, we review new features in ECMAScript 5 and its strict mode, and briefly introduce an implementation of Secure ECMAScript [4] which is designed based on ECMAScript 5 strict mode and can be used to construct a sandbox environment that we use to deploy the enforcement mechanism in this study. We also review the lightweight self-protecting JavaScript implementation [18, 13], address some limitations that we revise and adapt to this work.

2.1 ECMAScript 5 and its strict mode

ECMAScript 5 (ES5) [3], released by the ECMA committee in December 2009, is a new standard specification of JavaScript language which represents, from a security perspective, a huge improvement over the previous (current) specification, ECMAScript 3. Besides new features such as providing a way (`Object.defineProperty` method) emulate platform objects, or providing new APIs, ES5 provides more robust programming write *secure* JavaScript. Firstly, objects in ES5 can be frozen such that the frozen objects are tamper-proof. Secondly, isolation problems in ES3 are solved in ES5 strict mode, a restricted subset of ES5.

ECMAScript 5's strict mode Strict mode can be set in ES5 by putting the directive "`use strict`" at the beginning of a function body or an entire script, including `eval` code, `Function` code, or event handler attributes. The strict mode creates a restriction on ES5 language to archive two isolation properties: *static lexical scope*, and *no encapsulation leak*.

Static lexical scope Strict mode provides complete lexical scoping by disallowing deletes on variable names, no prototype chain for scope objects, disallowing `with` statements, which provide a mechanism to insert scope objects.

No encapsulation leaks in a closure In ES3 or most current browsers, there are several channels that untrusted code running within a restricted closure can access the global object by referring to the implicit `this` parameter, or access critical resources by abusing caller-chain. ES5's strict mode repairs these leaks by disallowing these channels. These restrictions can plug encapsulation leaks that happen in ES3 so that make it possible to implement safe closure-based encapsulation.

2.2 Constructing a sandbox in Secure ECMAScript 5 (SES)

Secure ECMAScript 5 (SES) [4] is a library developed by the Google Caja team, aiming to solve two API confinement problems [22] in order to construct a sandbox without using server-side transformation like in Caja. When the SES library is initialized, it first patchs away built-in objects having bad and non standard behaviors such as `RegExp`. Non-whitelisted properties of built-in objects are deleted and the built-in objects are made *transitively immutable* [22] by being frozen using the `Object.freeze(..)` built-in method in ES5. The properties of the built-in objects are then immutable and the objects cannot be extended with additional properties.

The second step is to make untrusted code running inside an `eval` function has unaccessible to global variables and the global object. The SES library solves this issue by first enforcing the untrusted code running on strict mode. Then a super set of free variables[2] are computed such that free variables not in a set of accessible objects are rejected at run-time. This set of accessible objects is computed by combining the whitelisted built-in objects and the enumerable own properties of a provided API object argument. This is to ensure that the untrusted code can only have access to the whitelist built-in objects, the API and the objects created by itself. Once the SES library is initiated in a page (frame), a sandbox environment can be constructed in the frame for untrusted code representing in a string `untrustedCodeSrc` with a API `api` as illustrated as follows.

```
var moduleMaker = cajaVM.compileModule(untrustedCodeSrc);
var sandboxed = moduleMaker(api);
```

Taly *et al.* [22] have developed the formal semantics of SES_{light}[3] and verify the soundness

[2] As an example, in the expression **var x = y + z**, y and z are listed as free variables. More details can be founded in [22]

[3] A subset of SES obtained by removing setter/getter from SES

of API confinement in SES_{light}. We refer readers to [22] for full details of the semantics and the proof. In this work, we use this SES library to construct sandbox environments in order to employ our two-tier architecture for enforcing modular fine-grained policies.

2.3 Lightweight self-protecting JavaScript

Lightweight self-protecting JavaScript [18] is a policy enforcement mechanism that uses reference monitors to wrap built-in methods in a web page to make the page self-protected. The method is "lightweight" in the sense that it does not need to parse or transform JavaScript code or the body of the page. Instead, the policy code is injected into the header of the page ensure that the policy code is executed first in order to wrap the security critical methods before the attacker code can get a handle on them. Security policies in this mechanism are defined in pure JavaScript language in aspect-oriented programming (AOP) style so that they can express stateful and fine-grained policies.

Implementation issues in the self-protecting JavaScript Phung *et al* implemented the self-protecting JavaScript approach via an adaptation of an AOP library. The key point of this method is that the enforcement and policy code must be tamper-proof and the attacker cannot obtain references to the unwrapped methods. However, due to some unexpected features in current JavaScript, there are some vulnerabilities that the attacker can exploit to *subvert* the enforcement. Magazinius *et al* [13] addressed these issues and provided support for correct policy construction.

In summary, although the method provides a way to specify and enforce fine-grained policies, it does not distinguish between trusted and untrusted code, and therefore cannot be used to define modular policies for portions of untrusted code within a page. In this work we adapt this enforcement mechanism to fit on our two-tier enforcement architecture. Moreover, we revisit the issues addressed in [13] in order to fit in the new context of ES5.

3 Enforcement architecture

Given a piece of untrusted JavaScript and an API, we can dynamically load and execute the untrusted code, without static code validation, transformation or filtering, in a compartment created by a sandbox environment introduced in Section 2.2. Similar to any other sandbox models, the untrusted code can only interact with the outside environment through the API provided to it by the sandbox.

3.1 Assumptions on a baseline API

We assume there exists an API library providing baseline access to the hosting page, e.g. the Document Object Model (DOM). The soundness and confinement properties of the API are assumed to be already established and proved by an automated tool e.g. [22, 19]. The return value of an API call is safe and has no side-effects. The API may implement some static policies, such as sanitization of HTML input, which apply for any general untrusted application.

3.2 Two-tier sandbox architecture

Our goal is to specify and enforce modular and fine-grained policies on such an API for specific untrusted applications. As an example, untrusted code is allowed to call `getElementById(..)` method of a API `document` object. If the `document` object provides capability of accessing the

`getElementById(..)` method, the untrusted code can call the method to access any element of a page. In practice, a web master of a hosting page may enforce more restriction specific to the page. For example, (1) the untrusted code can only call the function a limited number of specific whitelisted element *IDs* in the hosting page, (2) the untrusted can call the function at most e.g. 3 times, and (3) enforce further policies on the returned object, for example, allow read-only for a critical element while allow full access to an element that is assigned for the untrusted code to read and write.

In principle, such a fine-grained policy can be embedded within the implementation of the API. However, this makes the implementation too complex and thus error-prone. Indeed, combining policy code with the implementation of a mediator object makes it difficult to maintain. Moreover, defining a new policy or modifying an existing policy requires changing the implementation of the API. In addition, application-specific policies should be defined and enforced for a particular untrusted code in one hosting page or cross different hosting pages. Therefore, it is also inflexible to include policy definition and enforcement within the API implementation.

Our approach for policy enforcement in untrusted JavaScript is to separate policy definition and enforcement from implementation of an API. Before providing the API to untrusted code through a sandbox, the API is enforced by specific policies predefined for the untrusted JavaScript. The policy enforcement is executed within a sandbox environment to ensure that the policy code can only interact with the API and therefore can prevent "idiot" policies which may e.g. return a original DOM object. The enforced API is returned and then provided as a usual API for the untrusted code through another sandbox environment. This architecture is illustrated in Fig. 1.

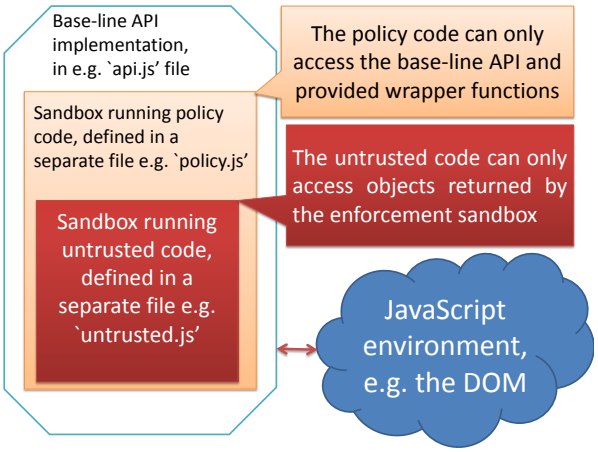

Figure 1: The two-tier architecture

3.3 Deploying untrusted code into the enforcement architecture

Untrusted code in e.g. mashup applications, is usually hosted at a third-party server, and embedded to a hosting page by `<script>` tag, i.e. `<script src='http://mashup.org/code.js'>`. To deploy untrusted code into a hosting page using the proposed sandbox architecture, the code must be first retrieved as a string in order to be executed within a sandbox environment. There are several alternatives to get a mashup code as a string: (1) Using the Uniform Messaging Policy (UMP) [25]: browsers supporting UMP allow client-side JavaScript can request a content from a cross domain source (2) Server-side support: server-side script may retrieve a mashup

source code and inject into the hosting page. In this work, we just assume that the code can be retrieved and uploaded on the hosting server so that it can be loaded at runtime using `XMLHtmlRequest`. The pseudo-code in Listing 1 illustrates this deployment.

Listing 1: Template for enforcing policy for untrusted code in the two-tier architecture

```
var api = ...//create an API object
//using XMLHtmlRequest to get the content of file
//'policy.js' into 'policyCode' variable
var moduleMaker = cajaVM.compileModule(policyCode);
var enforcedAPI = moduleMaker(api);
load_untrustedCode(enforcedAPI);
function load_untrustedCode(api){
   //using XMLHtmlRequest to get the content of file
   //'untrustedcode.js' into 'untrustedCode' variable
   var moduleMaker = cajaVM.compileModule(untrustedCode);
   moduleMaker(api);
}
```

4 Policy definition and enforcement

As mentioned briefly in introduction, a policy enforcement mechanism has been proposed in [18], however, the enforcement is proposed for JavaScript at page level which cannot be applied in untrusted scenarios. Moreover, the implementation of [18] is in current JavaScript specification and faced some vulnerabilities which have been patched in a later work [13]. One of our goals in this work is to revisit and revise the implementation in ES5 specification in order to adapt the enforcement mechanism to the sandbox model.

Similar to the approach in [18], our policy enforcement mechanism is to intercept the accesses on the methods and fields of the mediator objects of an API. A policy for such an access defines if the access is allowed, rejected or the returned modified according to a further policy. Within policy code, a policy writer can define helper-functions and variables as security states to keep some execution history of the code, or as some sensitive information such as whitelists. The basic idea of the enforcement is first to keep the reference to the method or property to be enforced, and then execute the policy which decides whether to allow access on the original method or property. Differing from [18], this enforcement is executed within a sandbox environment, therefore local variables and functions are protected. Moreover, the enforced object is sealed [4] so that it cannot be deleted. We present in detail the enforcement for method invocation and property access.

4.1 Policy definition

A policy for method invocation defines whether or not the invocation is proceeded depending on some conditions. In our enforcement model, a condition could be based on security states, patterns such as whitelists, and the value of the arguments. We elaborate in detail our language to specify such fine-grained policies to be enforced in our model.

We propose two types of policy definition: (1) property access policy and (2) method invocation policy since an object in an API contains properties and methods proxying accesses to the real corresponding object. We define a pattern for policy definition in which a policy is defined in an object variable specifying property or method name together with its policy. Property access policy defines read and write policy on the property, and method call policy defines the

[4] In ES5, sealing an object by e.g. Object.seal(obj) can set all existing properties of the obj object to non-configurable, i.e. all property descriptors cannot be changed and all the properties cannot be deleted.

response to the call, e.g. allow, deny, or enforce the returned object with further policy. Policies are written in a function. Therefore, it is possible to define stateful and fine-grained policies. Listing 2 illustrates a policy example.

Listing 2: A policy example defined for an object

```
var document_policy={
 getElementById : {
   method:function(args,proceed){
     var id = args[0];
     if(id === 'main'){
       return proceed(div_Main_policy);
     }
     //.. more cases
   },
   args:['string']
 }
 ...// other properties and methods' definition
}
var div_Main_policy = {
 style: {
   property:{
     read: function(){ return div_Main_style;},
     write: function(value){ return false; }
   },
   args:['*']
 }
 ...//properties and methods' policy definition
}
var div_Main_style = ... //define a further policy
```

The policy of a method of a specific object is defined in a function with two parameters `function (args,proceed){..}`, where *args* is the arguments of the invocation, and *proceed* is the function to control the execution of the original method. Calling the *proceed*(..) function is to allow the original method to be executed. Our policy definition proposal provides a systematic way to write fine-grained and stateful policies depending on invocation arguments (first parameter in the policy function) and security states (can be encoded in variables) at runtime. If the original method return an object, the object must be enforced by a predefined policy to ensure full mediation. Based on the above assumption on a return object of API call is safe and the fact that the policy writer knows exactly the type of the returned object and which policy should be enforced on the returned object, we provide a way to define this recursive enforcement by calling the *proceed*(..) function with one parameter as the desired policy. This implementation feature is different from [18] because we enforce policies on API object while the implementation in [18] enforce policies on built-in methods.

A property access policy includes *read* and *write* policy defined in a corresponding function which returns a boolean value indicating access permission.In the *write* function, the *value* argument is the real value assigning to the property at runtime, which the policy can inspect. In the *read* function, if the returned property is an object, the returned object must be enforced by a further policy by returning the policy object variable. Similar to method call policies, these policies are stateful in which *security states* can be defined and updated runtime to be used by the policies.

Inspecting arguments As mentioned, a policy may need to inspect the invocation arguments, *security states*, and/or patterns such as whitelists. As pointed out in recent work [18, 13], arguments in JavaScript are *non declarative*, thus could be the source of attacks [18, 13, 11, 15, 14] because of implicit type conversion in JavaScript when a policy inspects arguments provided by untrusted code. In [13], the authors proposed a way to define and enforce declarative arguments by coercing each argument value based on a declared type to ensure that the value when inspecting is the same value when using the argument. In above policy template, we propose a `args` field, which declares the types of the argument specified in an array using the grammar proposed in [13]:

$$\mathsf{type} \;::=\; \texttt{`string'} \mid \texttt{`number'} \mid \texttt{`boolean'} \mid \texttt{`*'} \mid \texttt{undefined} \mid \{\mathsf{field}_1 : \mathsf{type}_1, \ldots, \mathsf{field}_n : \mathsf{type}_n\}$$

The `'*'` type is a reference type which provides reference to a value without accessing the value itself. Only argument elements declared by the type array can be inspected by the policy and the value is explicitly coerced to the defined type. We do not proceed type for the return value as in [13] since our policy enforcement is on API objects which has been assumed a safe returned value from an API call. Instead, we propose more fine-grained enforcement for the returned object as argued above which is possible to specify policies for a returned object in order to recursively enforce full mediation.

4.2 Enforcement method

Our enforcement method is implemented in whitelist manner, i.e. only methods and properties defined in the policy are accessible, the other are absent from the enforced object. We provide an interface `enforceWhitelistPolicies(..)` to enforce a policy on an object. Given a policy defined in an object variable `pol` and an object to be enforced in `obj`, the object `obj` is enforced by the policy `pol` by calling `enforceWhitelistPolicies(obj,pol);`. The key functionality of `enforceWhitelistPolicies(obj,pol)` is traverse the policy object `pol` to get all the names of methods and properties together with the policy in order to enforce the policy on the same name of the object `obj`. The rest of methods and properties of the object not defined in the policy are redefined as an empty function or null value so that they are not accessible from untrusted code. We enforce a method call policy and a property access policy differently as follows.

4.2.1 Enforcing method call policy

We adapt the enforcement implementation available on [13] to handle the enforcement for returned object. In summary, the enforcement for a method invocation policy is a wrapper that keeps the reference to the original method of the object to be wrapped, and redefines the method by invoking a policy function which can control the execution of the original method. As described, a policy is defined as a function with two arguments: the first argument is the parameters of the invocation, the second argument is the `proceed` function, which is to control the execution of the invocation, i.e. calling the `proceed` function will execute the original method, otherwise, the method is suppressed. We modify the `proceed` function to take one argument as a policy for the returned object of the original method. If this policy is defined (from the policy to be enforced), the returned object will be recursively enforced by the provided policy.

4.2.2 Enforcing property access policy

Our enforcement on property access policies relies on the `Object.defineProperty(..)` method in ES5 to enforce desired policies. We first get the current getter-setter functions of the property of the object (using `Object.getOwnPropertyDescriptor(..)`) to be enforced. We then define a

new descriptor with getter and setter functions to execute *read* and *write* policy functions from the policy so that these policy functions are always invoked whenever the property is accessed. Depending on runtime policy results from the *read* and *write* policy functions, the original getter and setter functions which has been stored will be called to run in the context of the object. The returned value will be further enforced if there is a policy defined in *read* policy function.

4.3 Tamper-proofing arguments

These enforcement mechanisms on method calls and property accesses are performed on a baseline API in SES_{light} which the soundness and API confinement have been proved [22]. Our architecture is based on sandboxed compartments in a SES environment, the capability of untrusted code, therefore, is confined by the API which are provided by the enforced objects and the sandbox environment (cf. Section 2.2). It means that the untrusted code within the sandbox cannot access arbitrary references except the provided API.

In a SES environment, built-in objects are frozen so that untrusted code cannot modify a built-in prototype to launch *prototype poisoning* attack on policy enforcement code. *Prototype poisoning* is a attack vector in which the attacker can compromise trusted code by modifying a global prototype that is inherited by the trusted code [14, 13]. Our enforced objects are protected by using `Object.seal(obj)` in ES5 such that existing properties of the object to non-configurable, i.e. all property descriptors cannot be changed and all the properties cannot be deleted (in Mozilla, deleting a wrapped object recover the original object [18]). Therefore, the enforcement mechanisms are tamper-proof.

Our policy definitions allow a policy can inspect the parameters of a function call and property write. We provide a way to define the types of the parameters of a function call or property write so that the policy can inspect the parameters. Therefore, it is ensured that the parameters are inspected in the expected types to avoid implicit type coercion which is a source of attacks as mentioned earlier. Our policy definitions also provide a flexible way to write and enforce runtime policies on returned objects of the enforcement to ensure full mediation on API objects.

5 Empirical Studies

We deploy some selected use cases, representing different mashup web application scenarios, to the sandbox architecture to demonstrate the effectiveness of the approach.

5.1 Framework for constructing mediator objects for DOM access

The sandbox environment described in Section 2.2 can load and execute untrusted code dynamically. Within a sandbox environment in SES, there is a whitelist of built-in objects and functions, e.g. `Array`, `Math, Date`, etc that untrusted code in the sandbox can invoke. However, in order to interact with the hosting page containing the sandbox, the untrusted code must access e.g. the `DOM` objects to manipulate the hosting page. As such, untrusted code running within a sandbox environment must be provided an appropriate API which allows the code to interact with the hosting page to function properly, e.g. to display its content, or to handle user interaction in the hosting page. In our enforcement architecture, we assume there exists such an API so that our architecture can provide modular and fine-grained policy enforcement on that API. In this paper, to illustrate our enforcement architecture, we construct a simple API which mediate selected accesses to the DOM. We use a virtualization technique to implement

the API. Virtualization is a known technique which has been employed in e.g. Domita [16] or ADsafe. The technique is to construct mediator objects by creating virtual objects which provide predefined methods and properties to mediate accesses to the DOM. In this work, these objects will be provided to the untrusted code as the same name as in the real DOM so that the untrusted code does not need to be programmed or rewritten in a different interface[5].

Virtualization We virtualize a critical object by creating a constructor function and store the original critical object in a reference map pointing to the constructor itself. The map object is out of the scope of the constructor function, therefore it is inaccessible from the untrusted code. This can avoid a transformation or static validation of untrusted code as performed in e.g. Domita, and ADsafe to prevent untrusted code access special variables storing original critical objects. We use the WeakMap implementation in the SES library [4] to keep such references. We then build the prototype of the constructor with methods and properties having the same name of those of the critical object. In each method or property, we can check the arguments and enforce a static policy to ensure some security properties before invoking or returning the original method or property retrieved from the reference map. The returned object by the original method is also mediated to ensure the complete mediation. Arguments of a method call are also wrapped so that no side-effects can happen.

5.2 Testing the architecture with some selected JavaScript gadgets

We select some JavaScript gadgets representing useful mashup functionalities and specify some sensible fine-grained policies for each gadget and test the enforcement to demonstrate the effectiveness of the enforcement mechanism.

Displaying text in the hosting page We employ a digital clock gadget which represents the display function that displays current date time of system time to a hosting page. This gadget gets system time by creating a `Date()` object, calculates values for a digital clock such as hour, minute, date, day, month, and year, and displays all information in a hosting page. Our policies allow the gadget access to the $innerHTML$ property of some predefined elements, and only allow the content value matching with a predefined value list. Thus, the gadget cannot write arbitrary HTML to the hosting page. The other properties or methods of the accessed elements is not defined in the policy and thus inaccessible from the gadget.

User interaction We use a simple roman converter gadget which accesses a textbox in a hosting page to get its value when user types 'enter', then converts it to roman numerals and displays to a specific div element in the page. The gadget interacts with these two elements by using `getElementById`.

We define policies allowing the gadget to call `document.getElementById` for the two specific elements. We also define and enforce different policies for returned objects of each particular elements. For the textbox element, the gadget can call `addEventListener(..)` to register 'keydown' keyboard event and access the *value* property to read only. The policy can ensure that the value is in integer number before returning. For the div element, the gadget can only set to the element $innerHTML$ to display. No arbitrary HTML is allowed but only roman numerals.

[5] ADsafe is an example of using a different API interface so that the untrusted code must be programmed in that interface

Interact with style and image This gadget displays a random image with hyper-link each time it is loaded. A image is created by using `document.createElement(..)` method and is appended to an anchor element also created by `createElement(..)` method. The anchor element is displayed to the page by appending it to a div element, which is accessed by `document.getElementById(..)`.

Policies for this gadget include: (1) allow the gadget to create *ony one* image and *only one* anchor element by using `createElement(..)` method, the assignment to the *src* properties of the image and the *href* properties of the anchor element is allowed if they are in a corresponding pre-defined whitelist; (2) allow the gadget to call `document.getElementById(..)` if the ID is the div element, and the returned element only contain `appendChild(..)` method to assign the anchor element to the hosting page.

5.3 A case study: context-sensitive advertisement

We develop a simple context-sensitive ad web application simulating real world online text ad services such as AdBrite, or Google Adsense. We only focus on the context-sensitive functionality of the service which displays ad text based on the content a user is viewing. The hosting page is a simple dynamic content page displaying different text each time a user visits. There is an fixed area in the page for the ad script to display ad content. The ad script loaded in the page scans the text of the dynamic content to find matched keywords in order to display relevant ads based on the keywords. We implement this ad service by storing ad text, link and keywords in a large table. To access the main content of the page or to display ad content, the ad script must use DOM methods such as `getElementById`, `createElement`, `createTextNode`, however, only with some concrete parameter value, and depending on each specific parameter value, extra policy on returned object is needed.

5.3.1 Policies

In a sandbox, calling to a method provided by an API has full capability of the returned object provided by the API. In our two-tier architecture, modular and fine-grained policies should be defined for a specific application. In this ad scenario, each method of the API and its returned object are enforced by specific policies depending on nodes with which the ad script interacts. For example, the object returned by invoking `createTextNode` is just used to be appended to the hosting page without any further operations. Therefore, an appropriate policy for this object is keep the reference to the object without any methods and properties. `getElementById`, and `createElement` methods need more fine-grained policies as we motivate below.

There are only two elements in the hosting page that the ad script can access by calling `document.`
`getElementById(..)` method: the fist element, which is has ID '*main*', is the main content of the web page which is defined in a div element; the second element, which has ID '*ad*', is also a div element assigned for the ad area. The ad script must only have read access to the element '*main*' and write access to the element '*ad*', therefore, each returned object for these two elements must be enforced with further different policies. This scenario is illustrated in Fig. 2. We define a policy allowing the ad call the `getElementById` method for only the '*main*' and '*ad*' but for each particular element, we need to enforce further specific policies:

- For the DIV element '*main*', the ad script is allowed to fetch the content by reading the '$innerHTML$' property and to get the style of the element by accessing '*style*' property; no write access is allowed for these two properties. The other methods and properties of the elements are disabled.

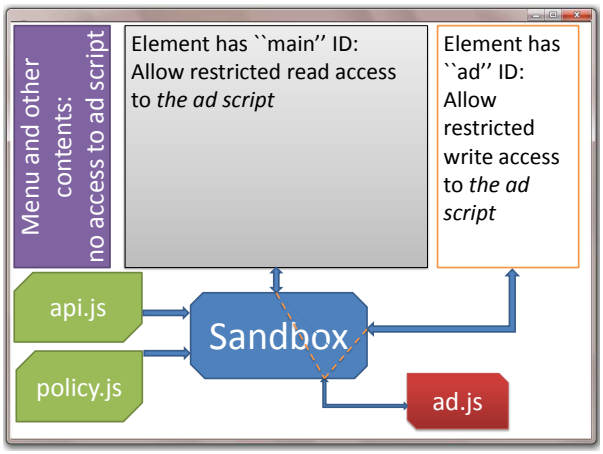

Figure 2: Policy and enforcement model for an ad application

- For the DIV element '*ad*', the ad script is allowed to display its content by using `appendChild` method. Our policy allows the ad script to call this `appendChild(child)` method if the `child` argument is a paragraph element node, and the method is allowed to call at most 3 times (to limit the number of text ad to 3) . Moreover, the ad is allowed to set the style, width, height of the ad area by writing to corresponding properties. The policy also restricts the maximum value of width, height so that the ad can not display an oversize area.

The ad script needs to display ad content including ad text and a corresponding anchor with a link to a URL. These functionalities can be performed by creating anchor and paragraph elements using `document.createElement`. To restrict access to the `document.createElement` method, we define a policy for this method that only allow to be called with tag '*a*' and '*p*'. A newly created anchor element is allowed to set the '*href*' property, which must be checked by a function to ensure the link is safe and allowed. The above fine-grained policies allow the ad script to function properly while prevent other accesses that might be sources for attacks. A real advertisement application may need more functionalities

5.3.2 Security test

We code the ad script and policies in two separate text files and deploy the script and policy enforcement into sandbox environment as described in Section 3.2. The ad script with enforcement mechanism run successfully on Mozilla Firefox 4.0.1 and Google Chrome 12.0.742.91 on Windows 7 platform.

We develop a malicious ad script attempt to violate the defined policies to test the security features of the policies and the policy enforcement mechanism. Moreover, the script also try to break security assurance by exploiting some known vulnerabilities as described in e.g. [14, 15, 13]. We first execute the script in the hosting page to ensure that the attacks are successful. We then deploy the ad script into a sandbox environment with an API without any enforcement to ensure the API provides adequate functionalities and the attacks are successful. Finally, we deploy the malicious script into the sandbox environment with an API enforced by above defined policies. We elaborate the results as follows.

Execute code not provided by policies Our enforcement is implemented in whitelist approach: only primordials defined in policies are accessible for untrusted code. This attack attempts to execute functions not defined in policies. For example, the malicious code tried to create an script element using `createElement('script')`, but this was denied by the policy. Attempting to access e.g. `document.cookie` is prohibited by the sandbox because this field is not defined in the policy.

Changing unallowed content The ad script allows to get the '*main*' element reference but does not allow to change its content or style, and so on. The malicious code tries to violate this policy by setting e.g. '*innerHTML*' field of the element object but it failed because the policy does not define such a capability for the element.

Setting parameters violating policies The ad script allows to create an '*a*' element and this element can be set to its 'href' field but only links in a predefined whitelist is allowed. The malicious code attempted to set an arbitrary link or a link with "`javascript:code`" to the field but these values are disallowed by the policy at runtime.

Exploiting vulnerabilities in current JavaScript The malicious code tries to launch malicious argument object attack but this is prevented by our declarative arguments. Prototype poisoning attack failed because the enforced objects are protected by being sealed and built-in objects are frozen.

5.4 Microbenchmark

We evaluate microbenchmark our enforcement architecture by measuring the slowdown of a number of individual JavaScript operations. Each operation loops in 1000 times which has been run 10 times to get average value. We first run the code without sandbox, then run the code inside a sandbox without policy enforcement, and finally run the code inside the proposed two-tier sandbox architecture with a policy. The average overhead of sandbox is 3.89 times slowdown while that of the two-tier architecture is 4.18 times slowdown.

6 Related work

Solving security issues for untrusted JavaScript has recently received wide attention both in industry and in the research community. However, most of the recent work concern the context of current version of JavaScript (ECMAScript 3). Proxy [24] is a recent approach in ECMAScript 5 to construct robust APIs. Although this approach does not allow to specify modular and flexible policies, it can be used to construct a robust API as a baseline API library for our approach, providing a complete framework for the DOM access for untrusted code. To the best of our knowledge, our work is the first study in enforcing fine-grained security policies for untrusted JavaScript in ECMAScript 5. In [21], the authors have reviewed current security mechanisms for untrusted JavaScript in the literature. In this section, we only review recent work related to fine-grained policy enforcement and sandboxing mechanisms. We divide the related work based on whether it requires browser modification.

Browser-level implementations have access to the lower-level implementation of the JavaScript interpreter, and therefore has possibility to modify or extend the semantics of JavaScript to provide greater security. However, this approach also has down side from an immediate practical perspective. It requires the browser users to be proactive to protect themselves. From a technical point of view, modifying a browser requires much effort. Moreover, the implementation is likely to change more frequently since the codebase of browsers e.g. Firefox normally change rapidly. JCShadow [17] is a recent work (and closest to our work) that also motivates for fine-grained policy enforcement for untrusted JavaScript, and proposes a reference monitor within a JavaScript

engine to enforce policies. The mechanism is implemented by modifying the JavaScript engine in Firefox 3.5. ConScript [15] modifies Internet Explorer 8 to provide aspect-oriented programming constructs for JavaScript in order to enforce fine-grained security policies. ConScript offers a mechanism to define and enforce policies for untrusted JavaScript code loaded within a `<script>` tag. Policies defined in ConScript are validated by a type system to ensure no vulnerabilities in policy code. CoreScript [26] proposed instrumentation for untrusted JavaScript by an add-on proxy in the browser which requires modifying the browser. CoreScript can enforce edit automata [10] policies which is essentially the same class covered by our policies. There are also several other approaches such as [6, 7] using browser modification to enforce policies. However, these methods can only enforce coarse-grained access control policies.

On the other hand, approaches to enforcing security policies without modifying browser have advantage in themselves. The enforcement can be provided as a library by a server or a proxy and the policies are enforced at runtime at the browser. One branch in this area is to modify the original program while the other deploys *non-invasive* approach to original code. BrowserShield [20], Caja [16], and Facebook JavaScript [5] are examples of the approaches using code modification or filtering. BrowserShield [20] is an approach using code transformation dynamically to enforce security policies. The idea of BrowserShield is further developed at Microsoft Live Labs as a Web Sandbox framework [9] which rewrites untrusted JavaScript to run it inside an isolated virtual machine which mediates access to the real JavaScript environment. Caja [16], an open-source project at Google, is another approach to enforcing policies of a web page on the client side. Caja defines a safe JavaScript subset based *object-capability* model and access control rights in Caja is represented as *capabilities*. Untrusted JavaScript code is transformed into a safe version with isolated modules by a rewriting process. The transformed code is provided APIs by libraries such as Domita to have indirect access to the DOM. Similar to Caja, Facebook JavaScript (FBJS) [5] is an another industrial approach to sandboxing untrusted JavaScript application embedded into Facebook. Untrusted code written in FBJS is also transformed in a separate namespace so that it is isolated to the other. Maffeis et al proposed another approach [11] for untrusted JavaScript which uses filtering, rewriting, and wrapping to isolate the untrusted code. Although these mechanisms have proved the soundness by semantics or automated tools [12, 22], they limit untrusted code into a subset of JavaScript and do not allow developers to specify application-specific and fine-grained policies as we investigate in this work. ADsafe [2] is another safe subset of JavaScript to allow untrusted advertisements executing on a trusted hosting page. The safe subset is an interface that mediates access to the DOM and other global variables to ensure that the untrusted code cannot perform malicious behaviors. Before placing an untrusted ad code into a hosting page, the ad code must be validated by an static analysis tool called JSLint to ensure that the untrusted code only has access to the interface provided by the ADsafe library. The soundness of API confinement of ADsafe has been shown in [19, 22]. Although this mechanism do not allow to define fine-grained policies, the ADsafe subset could be provided as a baseline API in our architecture so that the untrusted code can be loaded and executed dynamically without code validation by an off-line tool.

Non-invasive approach to enforcing security policies is exemplified by the lightweight self-protecting JavaScript method [18]. This method defines a wrapper library in aspect-oriented programming [8] style to intercept built-in functions with a security policies. The library is placed at the header of a page so that it can execute first to wrap sensitive function call and therefore to make the web page self-protecting. The implementation of this method faces several challenges which have been addressed in a later work [13]. However, the implementations in [18, 13] focus on enforcing policies on built-in objects which is at page-level while our architecture is to enforce policies on API objects for untrusted code. In our work, we revisit the implementation in the context of ECMAScript 5, and adapt and revise the implementation by the new advantage

features of ES5. ObjectViews [14] is a similar approach to our work which provides wrappers as a library in JavaScript to share objects among principals in the browser. In untrusted code context, ObjectViews [14] focuses on safe sharing of objects (in ES3) between privileged code and untrusted code. However did not discuss how to load and execute untrusted code as we investigate in this paper.

Similar to our case study of context-sensitive advertisement application, AdJail [23] is an approach to isolating an ad script into a hidden iframe (shadow page) which is enforced by the same-origin policy. The ad script interact with the hosting page through tunnel scripts in both frames which can enforce to confidentiality and integrity policies. However, because the ad script is not enforced by security policies on security-relevant events as in our work, it can still perform abuse actions that our system disallows such as displaying a new window.

7 Conclusion

We have presented an architecture to define and enforce modular and fine-grained security policies for untrusted JavaScript in SecureECMAScript (SES) environment in the context of ECMAScript 5. Instead of implementing security policies within an API, our mechanism is to intercept a baseline API with modular and fine-grained policies before providing the API to a sandbox environment. Thus the untrusted code running inside this architecture can interact with outside environment through two layers of enforcement: the interaction is first enforced by fine-grained policies on a baseline API, and then enforced by the confinement of the API itself. The policy enforcement mechanism is also executed within a sandbox so that the policy code cannot expose unprotected resources. The method calls and property accesses on a baseline API are intercepted and monitored to ensure that the untrusted code does not violate the defined policies. The implementation of the architecture is based on client-side JavaScript libraries so that it does not require browser modification. The untrusted code can be loaded and executed dynamically in the sandboxed environment without run-time checking or transformation. The architecture can be used in untrusted scenarios such as third-party gadgets, web mashups as we have demonstrated in our empirical studies.

Our future work may investigate on applying the two-tier architecture to real world mashup or advertisement applications with concrete and sensitive policies. Security test could be improved by employing automated tools such as ENCAP [22].

References

[1] Dasient Blog. Q1'10 web-based malware data and trends . `http://blog.dasient.com/2010/05/q110-web-based-malware-data-and-trends.html`. May 10, 2010.

[2] Douglas Crockford. ADsafe – making JavaScript safe for advertising. `http://adsafe.org/`.

[3] Ecma International. Standard ECMA-262: ECMAScript Language Specification. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`. 5th edition (December 2009).

[4] Mark S. Miller et al. Secure EcmaScript. `http://code.google.com/p/es-lab/wiki/SecureEcmaScript`. Accessed in March 2011.

[5] Facebook. Facebook JavaScript. `http://developers.facebook.com/docs/fbjs`.

[6] Oystein Hallaraker and Giovanni Vigna. Detecting Malicious JavaScript Code in Mozilla. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[9] Microsoft Live Labs. Web Sandbox. `http://www.websandbox.org/`. Accessed in May 2011.

[10] Jay Ligatti, Lujo Bauer, and David Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[11] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *ESORICS*, pages 505–522, 2009.

[12] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy'10*. IEEE, 2010.

[13] Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *AppSec10: Proceedings of the OWASP AppSec Research 2010, LNCS*. Springer-Verlag, 2010. To appear.

[14] Leo Meyerovich, Adrienne Porter Felt, and Mark Miller. Object Views: FineGrained Sharing in Browsers. In *WWW2010: Proceedings of the 16th international conference on World Wide Web*, New York, NY, USA, 2010. ACM.

[15] Leo Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *SP '10: Proceedings of the 2010 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010.

[16] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[17] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards Fine-Grained Access Control in JavaScript Contexts. In *Proceedings of the 31st IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2011.

[18] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight Self-Protecting JavaScript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, Sydney, Australia, 10 - 12 March 2009. ACM.

[19] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *20th USENIX Security Symposium*, 2011.

[20] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.

[21] Philippe De Ryck, Maarten Decat, Lieven Desmet, Frank Piessens, and Wouter Joosen. Security of Web Mashups: a Survey. In *AppSec10: Proceedings of the OWASP AppSec Research 2010, LNCS*. Springer-Verlag, 2010. To appear.

[22] Ankur Taly, John C. Mitchell, Ulfar Erlingsson, Jasvir Nagra, and Mark S. Miller. Automated Analysis of Security-Critical JavaScript APIs. In *Proc of IEEE Security and Privacy'11*. IEEE, 2011.

[23] Mike Ter Louw, Karthik Thotta Ganesh, and V.N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *19th USENIX Security Symposium*, 2010.

[24] Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM.

[25] W3C Working Draft. Uniform Messaging Policy. `http://www.w3.org/TR/UMP/`. Level One.

[26] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM.