A Two-Tier Sandbox Architecture for Untrusted JavaScript

Phu H. Phung*
ProSec Security Group
Chalmers University of Technology
Gothenburg, Sweden
phu.phung@chalmers.se

Lieven Desmet
IBBT-DistriNet Research Group
KU Leuven
3001 Leuven, Belgium
lieven.desmet@cs.kuleuven.be

ABSTRACT

The large majority of websites nowadays embeds third-party JavaScript into their pages, coming from external partners. Ideally, these scripts are benign and come from trusted sources, but over time, these third-party scripts can start to misbehave, or to come under control of an attacker. Unfortunately, the state-of-practice integration techniques for third-party scripts do not impose restrictions on the execution of JavaScript code, allowing such an attacker to perform unwanted actions on behalf of the website owner and/or website visitor.

In this paper, we present a two-tier sandbox architecture to enable a website owner to enforce modular fine-grained security policies for potential untrusted third-party JavaScript code. The architecture contains an outer sandbox that provides strong baseline isolation guarantees with generic, coarse-grained policies and an inner sandbox that enables fine-grained, stateful policy enforcement specific to a particular untrusted application. The two-tier approach ensures that the application-specific policies and untrusted code are by default confined to a basic security policy, without imposing restrictions on the expressiveness of the policies.

Our proposed architecture improves upon the state-of-the-art as it does not depend on browser modification nor preprocessing or transformation of untrusted code, and allows the secure enforcement of fine-grained, stateful access control policies. We have developed a prototype implementation on top of a open-source sandbox library in the ECMAScript 5 specification, and applied it to a representative online advertisement case study to validate the feasibility and security of the proposed architecture.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers*; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

JavaScript, Security, ECMAScript 5, Sandbox

Keywords

Web Application Security, Fine-grained Security Policy, Web Mashups, Untrusted

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. Typically, a web master of a hosting page needs to trust the external JavaScript code before inserting it 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 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 [24], which is not under the control of the hosting page.

There have been a number of solutions proposed dealing with untrusted JavaScript code, such as Google Caja [17], Facebook JavaScript [4], ADsafe [2] and Conscript [16], and so on. However, we identified some important limitations with the current state of the art as will be discussed in more detail in Section 2. In particular, most of the approaches require (1) browser modifications or server-side pre-processing e.g. static validation, filtering, or transformation of the untrusted JavaScript, limiting the deployment capabilities, or (2) restrict the expressiveness of the enforcement to coarse-grained access control policies.

To achieve both the expressiveness of security policies as well as easy of deployment of our secure integration architecture, we propose a client-side security architecture that enforces fine-grained, stateful security policies for untrusted JavaScript code but does not require pre-processing of the code nor browser modification. This client-side security architecture is realized by means of a two-tier sandbox architecture: a generic, coarse-grained sandbox provides strong

*Part of this work was performed while the author was visiting Stanford University hosted by Prof. John Mitchell.

JSTools'12, June 13, 2012, Beijing, China

baseline isolation guarantees, whereas a second sandbox enables fine-grained, stateful policy enforcement specific to a particular untrusted application. The application-specific policy enforcement code is executed within an outer sandbox environment which guarantees that it – even if subverted or badly written – will still adhere to some general security policies provided by the baseline isolation of the outer sandbox, and e.g. does not give unauthorized access to sensitive resources or unintentionally leak unprotected references.

We have developed a prototype implementation of the proposed two-tier sandbox architecture, leveraging on recent developments in ECMAScript 5, a new JavaScript specification. For the sandboxing technology, we have built upon an existing open-source JavaScript library [3]. This library applies the strict mode of ECMAScript 5, and allows to load and execute untrusted code dynamically in a sandbox environment. The fined-grained policy enforcement is a modified version of our lightweight self-protecting JavaScript mechanism [19, 14], and improves upon the earlier work by achieving strong security guarantees and being able to deal with untrusted code. The policies express access control restrictions for both method calls as well as property accesses, and are expressed in pure JavaScript so that a policy writer can easily express stateful and fine-grained policies.

Our proposed architecture improves upon the state-of-the-art since it does not depend on browser modification nor transformation of client-side code, and allows the secure enforcement of fine-grained, stateful access control policies for untrusted JavaScript. The solution enables application-specific policy enforcement, even if multiple third-party applications are loaded on the same page. We have applied the prototype implementation to a set of mashup components, and a representative online advertisement case study to validate the feasibility and security of the proposed architecture.

Organization.

The rest of this paper is written as follows. The next section briefly sketches the challenges for securely integrating third-party JavaScript, and expresses the requirements for a client-side security architecture. Section 3 proposes the two-tier enforcement architecture for policy enforcement of untrusted code on an API in sandbox compartments. We present the prototype implementation of the architecture in Section 4. In Section 5 we present our design for specifying fine-grained policies. The validation of the prototype are presented and discussed in Section 6. Discussion and future work are given in Section 8.

2. PROBLEM STATEMENT

The state-of-practice technique to integrate third-party scripts in webpages is via script inclusion [22]. By doing so, the browser will execute the code as if it part of the original webpage, without any restrictions of the Same-Origin Policy. As a side effect, the third-party code executes in the same JavaScript context, and has access to all the code and data of the integrating webpage.

This is clearly not desirable in case the third-party JavaScript is malicious or can not be trusted. But even if the external party is trustworthy at the moment of website construction, the partner can become malicious over time, or be the victim of an attacker by itself. As a result, if an attacker controls the integrated script, he has full control over the hosting website and can perform unwanted actions on behalf of the website owner as well as website visitors.

Therefore, several countermeasures have been proposed [21, 13, 11, 19, 16, 12] and some of them are widely adopted [17, 4, 2]. Section 7 discusses various solutions in more detail, but in this section we briefly highlight two important shortcomings in the current state-of-the-art: the ease of deployment and the expressiveness of security policies.

Ease of deployment.

There are several modes of deployment, each with their own characteristics. The most invasive set of solutions require client-side modifications [18, 16, 25], which strongly limits the adoption of the solution, and requires web users to install additional extensions or custom-build browsers. Other solutions requires server-side pre-processing (such as filtering, transforming or wrapping untrusted code) [11] or impose restrictions to the third-party code to adhere to safe subset of JavaScript [4, 2, 12].

Both browser modification and pre-processing approaches make the deployment more difficult, and differ from the state-of-practice setting in which a legacy browser fetches JavaScript code directly from the external partner.

Expressiveness.

Most of available solutions only allow coarse-grained access control policies to be specified. Facebook JavaScript [4], ADsafe [2], and Google Caja [17] for instance only allow to enforce general coarse-grained policies for untrusted JavaScript. Alternative solutions providing fine-grained policy enforcement rely however on client-side modifications [18, 16, 25] or server-side pre-processing [11].

Another key issue in policy enforcement is that writing fine-grained, stateful policies can be hard and error-prone, and attackers can subvert defective policies to bypass the enforcement [16, 14]. One approach to tackling this issue is to constrain the way that policies are written, however it limits the expressiveness of policies. For instance, WebJail [25] constrains policies to whitelists to avoid '*inverse sandbox*' attacks. Designing a specific policy language which can construct a suitable static type system to ensure the correctness of policies [16] but this typically also requires modifications to the browser.

2.1 Requirements

As stated before, the current set of secure integration techniques for third-party JavaScript either rely on browser modification or code pre-processing, or strongly limits the expressiveness of the security policies. To achieve an expressive and easy-deployable client-side security architecture, we therefore propose the following requirements:

R1. The client-side security architecture should be able to cope with fine-grained security and/or stateful security policies.

For instance, a policy developer should be able to express that (1) only a subset of security-sensitive operations can be used, (2) that certain functions only can be called if the arguments satisfy additional constraints (e.g. appear in a whitelist), and (3) that no cross-origin requests may be sent out after user-supplied input has been received (e.g. as part of a form).

R2. To ease the deployment of the security architecture,

the solution should not require any modification of the browser, nor should depend on server-side preprocessing of the untrusted JavaScript code.

By targeting mainstream browsers, the solution can be deployed without additional effort of the end-user (such as downloading a specific browser extension or custom-build browser). Without the need for code preprocessing, the state-of-practice integration technique can be preserved to directly embed external scripts in the browser.

R3. The client-side security architecture provides complete mediation in accessing security-sensitive operations. Irrespectively of how the security-sensitive operation gets called, the security policies is always applied.

R4. The client-side security architecture must be robust to potential flaws in security policies.

Since the process of writing fine-grained and/or stateful security policies can be error-prone [15, 16, 14], the security architecture must be able to limit the elevation of privileges in case a security check can be circumvented due to a policy flaw.

3. TWO-TIER SANDBOX ARCHITECTURE

The main purpose of the client-side security architecture is to build a sandbox environment, in which untrusted scripts can run securely. In such a sandbox environment, all accesses to security-sensitive operations are mediated, and are controlled via a security policy.

3.1 Two-tier approach

Since writing fine-grained and/or stateful security policies can be hard and error-prone, especially for rich environment such as client-side web scripting, we propose a two-tier sandbox architecture. The architecture protects security-sensitive operations by nesting two sandbox environments, as depicted in Fig. 1: a generic, coarse-grained outer sandbox provides strong baseline isolation guarantees, whereas a second sandbox enables fine-grained, stateful policy enforcement specific to a particular untrusted application.

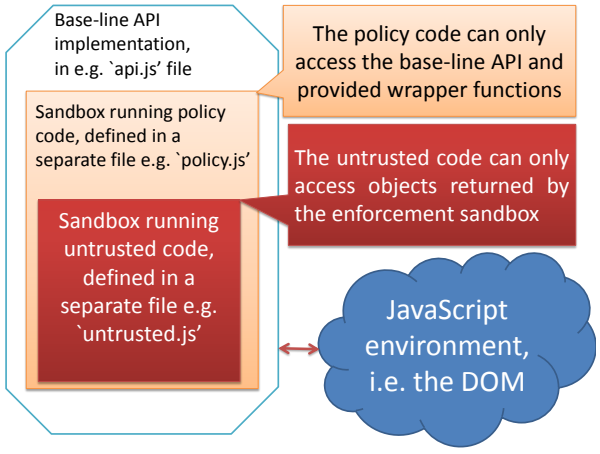

Figure 1: The two-tier sandbox architecture

Our motivation for the architecture is that giving a limited set of allowed operations as coarse-grained policy, a policy developer can define additional, fine-grained restrictions. Hereby, the policy developer can focus on the application-specific policy, rather that coping with the technicalities of achieving the basic set of restrictions and complete mediation.

Outer sandbox.

The outer sandbox provides a reusable sandboxing layer that enforces coarse-grained security policies, i.e. the outer sandbox limits the set of security-sensitive operations and properties available to the inner-sandbox.

By separating this functionality as a separate and reusable sandbox, the two-tier architecture is able to provide strong baseline guarantees, irrespectively of eventual policy flaws in the inner sandbox.

Also the quite challenging effort of achieving complete mediation in a JavaScript environment [19, 14] can be easily reused over multiple instantiations. Once the outer sandbox achieves complete mediation to a particular security-sensitive operation, this complete mediation is automatically inherited by any inner sandbox.

Inner sandbox.

The inner sandbox enforces additional application-specific constraints upon access to security-sensitive operations or properties. Since the baseline guarantees are already provided by the outer sandbox, the inner sandbox can provide an expressive policy environment to the policy writer, allowing him to express fine-grained and/or stateful security policies specific to an untrusted application.

In the remainder of this paper, we will focus on fine-grained access control policies, but the proposed security architecture can also be interesting to enforce usage control policies or information flow control policies.

Loading the security architecture.

Loading the security architecture works as follows. First, the outer sandbox, mediating access between the inner sandbox and the hosting page, loads itself and makes a limited set of security-sensitive operations available to the inner sandbox according to the coarse-grained policy. Next, the inner sandbox loads the untrusted JavaScript code into the second sandbox environment, in which accesses to security-sensitive operations are mediated by the application-specific security policy.

By doing so, accesses from untrusted JavaScript to security-sensitive operations first pass the application-specific policy enforcement of the inner sandbox. Upon approval of the application-specific policy, the call is delegated to the coarse-grained policy enforcement of the outer sandbox. This defense-in-depth approach ensures that the two-tier sandbox architecture, even if application-specific policy get subverted, can always guarantee coarse-grained isolation provided by the reusable outer sandbox, and therefore cannot unintentionally provide accesses to security-sensitive operations or properties on the hosting page.

4. PROTOTYPE IMPLEMENTATION

The prototype implementation of the two-tier sandbox architecture consists of two parts: the security architec-

ture realizing the sandboxing of untrusted code, and the fine-grained policy enforcement mechanism. For realizing the sandbox architecture purely in JavaScript, we employ a sandbox library namely Secure ECMAScript (SES) [3], developed in ECMAScript 5 by the Google Caja Team.

In this section, we briefly introduce the SES sandbox library and then present our prototype implementation of the architecture which is built on top of the SES library. The fine-grained policy enforcement mechanism is discussed in more detail in Section 5.

4.1 The Secure ECMAScript 5 sandbox library (SES)

Features of the current JavaScript language conspire to make sandboxing nontrivial, and sandboxing untrusted JavaScript code normally requires complex filtering, transforming and wrapping untrusted code to restrict the code to a manageable subset [12]. The ECMAScript 5 specification, released by the ECMA committee in December 2009, has been modified to make sandboxing easier and more widely applicable. ECMAScript 5 (ES5) 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) to emulate platform objects, or providing new APIs, ES5 provides more robust programming to 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. The strict mode creates a restriction on ES5 language to archive two isolation properties: *static lexical scope*, and *no encapsulation leak*. Strict mode provides complete and static 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. In ES3 and 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 to ensure *no encapsulation leaks in a closure*. These restrictions can plug encapsulation leaks that happen in ES3 so that make it possible to implement safe closure-based encapsulation.

Secure ECMAScript 5 (SES) is a sandbox library, developed in ES5 strict mode, that enables the construction of a sandbox without the need for server-side transformation or pre-processing. The main goal of the library is to make untrusted code run inside an isolated environment so that the untrusted code cannot access global variables and the global object but can only have access to the whitelist built-in objects, a provided API (which is essentially mediating access to security-sensitive operations) and the objects created by itself. We refer the readers to [23] for the full detail of the library together with its semantics and soundness. Giving a piece of untrusted code represented as a string variable `untrustedCodeSrc` with an API `api`, once the SES library is initiated in a web page (frame), a sandbox environment can be constructed in the frame as illustrated as follows.

```
var api = {...}; //constructing
var makeSandbox =
      cajaVM.compileModule(untrustedCodeSrc);
var sandboxed = makeSandbox(api);
```

4.2 The two-tier sandbox architecture prototype

Built on top of the SES sandbox library, our two-tier sandbox architecture consists of two nested sandbox compartments. As shown in Fig. 1, we assume that the baseline API is developed in a separated file 'api.js', the policy code is specified in 'policy.js' file, and the untrusted code is retrieved from a third-party site and stored at the hosting server in 'untrusted.js' file[1]. Listing 1 illustrates the deployment of the architecture. In this listing, the `api` variable is the baseline API, constructed in the 'api.js' file (included in the hosting page by a `<script>` tag as normal); the 'policy.js' and 'untrusted.js' files are loaded (using `XMLHtmlRequest`) into the `policyCode` and `untrustedCode` variables, respectively.

Listing 1: The structure of two-tier sandbox architecture

```
var outerSandbox =
      cajaVM.compileModule(policyCode);
var enforcedAPI = outerSandbox(api);
load_untrustedCode(enforcedAPI);
function load_untrustedCode(api){
   var innerSandbox
    = cajaVM.compileModule(untrustedCode);
   innerSandbox(api);
}
```

Creating the baseline API.

The outer sandbox relies on a baseline API being provided, and we assume that such a baseline API is available (and verified as in e.g. [23, 20]). Given such a baseline API, our architecture realizes modular and fine-grained policy enforcement on top of such a coarse-grained API in a secure manner. Constructing such an API is not a simple task and out of scope for this paper, but we like to refer to ongoing efforts such as the DOMADO library as part of the Google Caja project.

To validate the results in this paper, we opted to construct a proof-of-concept API which mediates selected accesses to the DOM. This API has been realized as follows.

Firstly, 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 (virtualization is a known technique which has been employed in e.g. Domita [17] or ADsafe [2]). 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 or ADsafe to prevent untrusted code access special variables storing original critical objects. We use the WeakMap implementation in the SES library [3] to keep

[1]We use this method to get untrusted code in order to load at runtime using *XMLHtmlRequest*. An alternative method is to use the Uniform Messaging Policy (see: http://www.w3.org/TR/UMP/) to request the code directly from the third-party site (cross domain).

such references.

Secondly, we have built the prototype of the constructors with methods and properties having the same name of those of critical objects. 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.

4.3 Tamper-proofing Arguments

Our architecture is based on sandbox compartments in a SES environment of which the soundness and confinement have been proved [23]. The capability of untrusted code, therefore, is confined by the API provided by the enforced objects and the sandbox environment. It means that the untrusted code within the sandbox cannot access arbitrary references except the enforced 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 an attack vector in which the attacker can compromise trusted code by modifying a global prototype that is inherited by the trusted code [15, 14]. Our enforced objects are protected by using `Object.seal(obj)` in ES5 so that existing properties of the object become *non-configurable*, i.e. no property descriptors can be changed, and no properties can be deleted. This is an important improvement since in Mozilla, deleting a wrapped object recovers the original object [19].

5. FINE-GRAINED POLICY DEFINITION AND ENFORCEMENT

The main goal of our two-tier sandbox architecture is to define and enforce stateful, fine-grained security policies specifically to a piece of untrusted JavaScript code. As mentioned briefly in introduction, we adopt the lightweight self-protecting JavaScript proposed in [19] for policy enforcement in the inner sandbox in our two-tier sandbox architecture. 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. Although the self-protecting JavaScript method provides a way to specify and enforce fine-grained policies, it does consider the whole page as untrusted code, except for the policy code itself[2], 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 sandbox architecture. Moreover, the implementation of [19] is in current JavaScript specification and faced some vulnerabilities which have been patched in later work [14]. We revisit and revise the issues addressed in [14] to fit in the implementation in the new context of ES5.

Similar to [19], policy enforcement mechanism in this work mediate access to security-sensitive methods and fields. A policy for such a mediator defines if the access is allowed, rejected or 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 mediated, and then execute the policy which decides whether to allow access on the original method or property. Differing from [19], this enforcement is executed within a sandbox environment, therefore local variables and functions are protected. Moreover, the enforced object is sealed in ES5[3] so that it cannot be deleted. We present in detail the enforcement for method invocation and property access.

5.1 Policy Definition

A policy for method invocation defines whether or not the invocation may proceed 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 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.

Listing 2: A policy example for an API 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_M_style;},
    write: function(value){return false;}
   },
   args:['*']
 }
//properties and methods' policy definition
}
var div_M_style=..//further policy
```

A *property access policy* includes *read* and *write* policy defined in corresponding functions which returns a boolean value or an object 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 before writing. Note that these values might be further constrained by some general policies e.g. sanitizing HTML content, in the baseline API, i.e. by the outer sandbox. In the *read*

[2]The policy code is injected into the header of the page to ensure that the policy code is executed first in order to wrap the security critical methods before the untrusted (attacker) code can get a handle on them

[3]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.

function, we can define further restrictions on the returned value by returning a policy predefined in an object variable which is enforced further on the value. These policies are stateful in which *security states* can be defined and updated runtime to be used by the policies.

A *method invocation policy* of a specific object is defined in a function with two parameters `function(args,proceed){..}`, where *args* contains the arguments of the invocation, and *proceed* is the function to control the execution of the original method. Calling the *proceed*(..) function will 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 returns 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 [19] because we enforce policies on API objects while the implementation in [19] enforces policies on built-in methods. Listing 2 illustrates a policy example on an API object including a *method invocation policy* (`getElementById` in `document_policy`) that recursively enforces a further policy (`div_Main_policy`) on the return value, and a *property access policy* (`style` in `div_Main_policy`).

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 our recent work [19, 14], arguments in JavaScript are *non-declarative*, thus could be the source of attacks [19, 14, 11, 16, 15] because of implicit type conversion in JavaScript when a policy inspects arguments provided by untrusted code. In our previous work [14], we 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. This declarative argument approach is applied in this work: the types of the arguments are declared in the `args` field in policy code as e.g. in Listing 2.

Only argument elements declared by the type array can be inspected by the policy and the value is explicitly coerced to the defined type. However, we do not have a type for the return value as in [14] since our policy enforcement is on API objects which are assumed to be safe. Instead, we propose more fine-grained enforcement for the returned object as argued above, whereby it is possible to specify a policy for a returned object in order to recursively enforce full mediation.

5.2 Enforcement Method

Our enforcement method is implemented in a 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 to enforce a policy on an object. The key functionality of the interface is to traverse the policy 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. The 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. Listing 3 illustrates part of the implementation of the interface.

Listing 3: Policy enforcement

```
function enforceWhitelistPolicies(object,
                                  policies){
  Object.keys(policies).forEach(
    function(name) {
    //inspect if the element is a method,
    //get the corresponding policy and types
     if(method)
       wrapMethod(object,name,policy,types);
     else  //property
       wrapProperty(object,name,readPolicy,
                    writePolicy,types);

  });
  //iterate the object to make the methods
  //and properties that are not defined in
  //policies unaccessible
  ...
  return Object.freeze(object);
};
```

A method call policy and a property access policy are enforced slightly different. We explain in detail the two enforcement mechanism as below.

5.2.1 Enforcing method call policies

Listing 4: Enforcing a method invocation

```
function wrapMethod(object, method,
                    policy,types){
 //..find function for possible aliases
 var original = object[method];
 object[method] = function() {
   var polArgs=//..clone arguments by
               // the defined types
   var proceed= function(policies) {
     var result=//execute original func
         if(!policies) return result;
         return enforceWhitelistPolicies
                  (result, policies);
   }
   return policy(polArgs, proceed);
 }
 return object[method];
}
```

We adapt our previous enforcement implementation [14] 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, representing the reference to the original function; calling the `proceed` function will execute the original method. We modify the `proceed` function (from [14]) 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. The simplified snippet of this interface is illustrated in Listing 4.

5.2.2 Enforcing property access policies

Listing 5: Enforcing a property access with a policy

```
function wrapProperty(object, property,
                      read, write, type){
  var desc = Object.getOwnPropertyDescriptor
                        (object, property);
  //..assert desc object
  var newdesc = {
   get: function() {
    var readPolicy = read();
    var value = desc.get.call(object);
    if (readPolicy===true) return value;
    if (typeof readPolicy==='object')
      return enforceWhitelistPolicies
                    (value, readPolicy);
   },
   set: function(v) {
    var cloneValue = coerceByType(type,v);
    var writePolicy = write(cloneValue);
    if(typeof cloneValue==='object')
      cloneValue = combine(cloneValue,v);
    if(writePolicy===true)
      return desc.set.call(object,
                        [cloneValue]);
  },
  configurable: false,
  enumerable: true
  }
  Object.defineProperty(object,property,
                                newdesc);
};
```

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 may 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. Listing 5 shows this enforcement mechanism.

6. VALIDATION

To validate the feasibility and security of our two-tier architecture, we have applied our prototype implementation in various application scenarios, in which untrusted third-party JavaScript code get integrated (e.g. gadget integration in web mashups). In this section, we will report on one particular case study, the integration of online advertisements. This case study captures the representative characteristics of context-sensitive text advertisement services such as AdBrite and Google Adsense.

The policy we want to enforce on the untrusted advertisement code is the following. First, we want to restrict the untrusted code to only write to a particular subset of the page (i.e. one particular div element, where the ad will be displayed). Next, we want to restrict the DOM read access to a particular subset of the page, so that only that part of the page is used in the context-sensitive analysis of the untrusted code. Moreover, we want to disable this reading access as soon as the user enters data into the page (e.g. by filling in an input form). These are just simple policies but represent application-specific and stateful policies. For example, we can define specific elements that the ad can read or write, or define a security state to monitor if the user enters the data.

To enforce these policies on context-sensitive advertisement scripts, we first select the basic API of the outer sandbox to which the application-specific policies and untrusted code will be confined. Next, we define fine-grained, stateful policies to enforce on the untrusted code.

The baseline API enables in a coarse-grained and application-independent way the set of features that are accessible within the outer sandbox (i.e. that can be used by the application-specific policy as well as the untrusted code). In our online advertisement validation experiment, we have chosen to only provide a very limited API, namely access to DOM operations. The baseline API is constructed via virtualization as briefly mentioned in Section 4.2. The technique constructs mediator objects by creating virtual objects which provide predefined methods and properties to mediate accesses to the DOM.

Next, we define the application-specific policy described informally above. Similar to the baseline API, we construct mediator objects to mediate read and write access to the DOM. The policy code also subscribes to the keyboard events e.g. *keydown* to capture any user input. The mediator object for read access grants access based on the requested DOM element and whether a user input event was captured, write access is granted solely on the DOM element specific to the ad. The policy allows the ad to set the style, width, height of the ad area and also restricts the maximum value of width, height so that the ad can not display an oversize area. Listing 6 illustrates some of these policies.

The context-sensitive advertisement script and the application-specific policy are deployed in two separate code files, and and loaded into the two sandbox environments as described in Section 3. This case study has been successfully tested on Mozilla Firefox 4.0.1 on a Windows 7 platform.

In addition, various security tests have been performed to assess the security guarantees by our proposed architecture. Based on known attack vectors and vulnerabilities in previous solutions (as described in e.g. [15, 16, 14]), we assessed whether untrusted code could break out of the sandbox environment.

For reference, we have first executed the attack vectors in the hosting page to ensure that the attacks are successful. We then have deployed the attack vectors into a sandbox

Listing 6: Application-specific and stateful policy examples for untrusted ad

```
var data_read = false;//a security state
var div_Main_policy = {
  innerHTML: {
    read: function(){
      if(dataread)
        return fasle;
      return true;
    },
    //... other definitions
};
var user_input = {
  addEventListener : {
    method:function(args,proceed){
      var eventStr = args[0];
      if(eventStr==='keydown'){
        dataread = true;
        return proceed();
      }
    },
    args:['string','function','boolean']
  },
  //... other definitions
};
```

environment with an API without any enforcement to ensure the API provides adequate functionalities and the attacks are successful. Finally, we have deploy the malicious script into the sandbox environment with an API enforced by above defined policies.

We did not success to break out of our proposed two-tier security architecture; the malicious script execution is prevented by two-tier sandbox enforcement. This result was to be expected, since our two-tier architecture relies on the same foundations and security guarantees of the Secure ECMAScript library (SES) [3].

7. 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 [26] 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 [22], 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.

Browser-level implementations have access to the lower-level implementation of the JavaScript interpreter, and therefore have the 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 [18] 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 [16] modifies Internet Explorer 8 to provide aspect-oriented programming constructs for JavaScript in order to enforce fine-grained security policies. ConScript can enforce edit automata [10] policies which is essentially the same class covered by our policies.

Inspired by ConScript, WebJail [25] applies the deep aspect weaving technique to FireFox browser, and introduces a least-privilege composition policy on top of this security architecture. This secure composition policy is based on an analysis of security-sensitive operations in the upcoming HTML5 specification, and provides a whitelist-based approach to nine disjoint categories of sensitive operations (such as external communication and client-side storage).

There are also several other approaches such as [5, 7] using browser modification to enforce policies. However, these methods can only enforce coarse-grained access control policies which are not applicable to untrusted script scenarios.

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 or restrict untrusted code in a safe subset while the other deploys *non-invasive* approach to original code.

Code transformation and safe subsets.

BrowserShield [21], Caja [17], and Facebook JavaScript [4] are examples of the approaches using code modification or filtering. BrowserShield [21] 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.

Google Caja [17] is another approach to enforcing policies of a web page on the client side. Caja defines a safe JavaScript subset based on *object-capability* model. 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. However, Caja does not support fine-grained policies enforcement as we investigate in this work.

Similar to Caja, Facebook JavaScript (FBJS) [4] 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, 23], 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 [20, 23]. 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 approaches.

Non-invasive approach to enforcing security policies is exemplified by the lightweight self-protecting JavaScript method [19]. This method defines a wrapper library in aspect-oriented programming [8] style to intercept built-in functions with a security policy. The library is placed at the header of a page so that it can execute first to wrap sensitive function calls and property accesses, 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 [14]. However, the implementations in [19, 14] 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 [15] 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 [15] 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 [24] 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, the framework only supports limited coarse-grained access control policies.

8. DISCUSSION AND FUTURE WORK

Our two-tier sandbox architecture is built on the specification of ECMAScript 5 and its *"strict mode"* to provide modular and fine-grained security policies for untrusted JavaScript code. The implementation of the architecture is based on an existing technique [19], and a library [3], however, improving on both of these. In particular, the two-tier sandbox architecture allows application-specific and fine-grained security policies be defined and enforced modularly, which is lacking in [3], and allow the enforcement on untrusted code, which is missing in [19]. Moreover, the policy enforcement mechanism is also executed within a sandbox so that the policy code cannot expose unprotected resources. This improves on both [3] and [19] by providing a fail-safe for badly written policies without need for complex policy language development. To the best of our knowledge, our security architecture is the first study in enforcing fine-grained security policies for untrusted JavaScript in ECMAScript 5[4]. In summary, the architecture is unique in the sense that it enforces application-specific and stateful fine-grained security policies for untrusted JavaScript code without browser modification or pre-processing of the code. In addition, the baseline API in the outer sandbox ensures a failsafe fallback in case of badly written policies.

Not all third-party code available in the wild supports yet the strict mode of ECMAScript 5, but we believe that this will be shifting quite rapidly. As ECMAScript 5 becomes available in all major browsers, and multiple content providers (such as Google and Yahoo) already favor the strict mode. Moreover, the API of the SES sandbox library is currently under proposal by the ECMA committee (TC 39) to be included as built-in features in a future version of ECMAScript [23], our work demonstrates the use of SES and provides a step towards a mechanism for executing untrusted code with application-specific and fine-grained security policy enforcement.

In future work we will further contribute to a robust baseline API, and we plan to validate our two-tier sandbox architecture on a broader range of real-life applications, by automatically injecting the client-side architecture for existing untrusted third-party code.

Acknowledgments

This research is partially funded by the EU FP7 WebSand project. Phu H. Phung was additionally supported in part by the Ericsson Research Foundation. Lieven Desmet was also partially funded by the EU FP7 NESSoS project, the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund KU Leuven. Thanks to David Sands and Andrei Sabelfeld for their feedback and helpful comments. The first author also would like to thank John Mitchell, Ankur Taly, and Mark Miller for their helpful discussions on an earlier version of this work.

9. REFERENCES

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

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

[3] Mark S. Miller et al. Secure EcmaScript 5. `http://code.google.com/p/es-lab/wiki/SecureEcmaScript`. Accessed in September 2011.

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

[5] Oystein Hallaraker and Giovanni Vigna. Detecting Malicious JavaScript Code in Mozilla. In *ICECCS '05:*

[4]IceShield [6] is a very recent ECMAScript 5 library inlined to a page to detect and prevent malicious behaviour of the page. However, similar to our lightweight self-protecting JavaScript approach, this library does not separate between trusted and untrusted code

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Mario Heiderich, Tilman Frosch, and Thorsten Holz. IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, RAID'11, 2011.

[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] Sergio Maffeis and Ankur Taly. Language-Based Isolation of Untrusted JavaScript. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *T. Aura, K. Jarvinen, and K. Nyberg (Eds.): The 15th Nordic Conference in Secure IT Systems (NordSec 2010), LNCS 7127*, pages 239–255. Springer-Verlag, 2012. (Selected papers from OWASP AppSec Research 2010, June 2010, Stockholm, Sweden).

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] Philippe De Ryck, Maarten Decat, Lieven Desmet, Frank Piessens, and Wouter Joosen. Security of Web Mashups: a Survey. In *T. Aura, K. Jarvinen, and K. Nyberg (Eds.): Information Security Technology for Applications, The 15th Nordic Conference in Secure IT Systems (NordSec 2010), LNCS 7127*, pages 223–238. Springer-Verlag, 2012.

[23] 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.

[24] 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.

[25] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Webjail: Least-privilege integration of third-party components in web mashups. In *ACSAC*, December 2011.

[26] 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.