Safe Wrappers and Sane Policies for Self Protecting JavaScript

Jonas Magazinius Phu H. Phung David Sands

Chalmers University of Technology, Sweden

Abstract. Phung *et al* (ASIACCS'09) describe a method for wrapping built-in methods of JavaScript programs in order to enforce security policies. The method is appealing because it requires neither deep transformation of the code nor browser modification. Unfortunately the implementation outlined suffers from a range of vulnerabilities, and policy construction is restrictive and error prone. In this paper we address these issues to provide a systematic way to avoid the identified vulnerabilities, and make it easier for the policy writer to construct declarative policies – i.e. policies upon which attacker code has no side effects.

1 Introduction

Even with the best of intentions, a web site might serve a page which contains malicious JavaScript code. Preventing e.g. cross-site scripting (XSS) attacks in modern web applications has proved to be a difficult task. One alternative to relying on careful use of input validation is to focus on code *behavior* instead of code integrity. Even if we cannot be sure of the origins (and hence functionality) of all the code in a given page, it may be enough to guarantee that the page does not behave in an unintended manner, such as abusing resources or redirecting sensitive data to untrusted origins.

One way to do this is to specify a policy which says under what conditions a page may perform a certain action, and implement this by a *reference monitor* [2] which grants, denies or modifies such action requests. In this paper we study this approach in a JavaScript/browser context, where the policy is enforced by using software wrappers. In the remainder of this introduction we review the background of policy enforcement mechanism in protecting web pages from malicious JavaScript code. A number of recent proposals implement policy enforcement by using wrappers to intercept security-relevant events. Here we sample the various approaches to implementing wrappers – each with their own advantages and disadvantages, before focusing in more detail on the approach, *self-protecting JavaScript*, that forms the main focus of this article.

1.1 The Wrapper Landscape

One key dimension for comparing security wrapper and sandboxing approaches is whether they require browser modification or not. Full browser integration offers some clear advantages. For example, the wrapping mechanism has direct access to the scripts as seen by the browser so there can be no inconsistency between the wrapper's and the browser's view of the code. Such inconsistencies are the basis for attacks, as is well known from the evasion attacks on script filters. The wrapping mechanism also has access to lower-level implementation details that would not be accessible at the JavaScript level, and permits modifications and extensions, for the greater good, to JavaScript's

semantics. The state-of-the-art in this approach is CONSCRIPT [21], which modifies Internet Explorer 8 to provide aspect-oriented programming constructs for JavaScript.

Avoiding browser modification, on the other hand, is an advantage in itself. For example it could allow a server to protect its own code from XSS attacks using an application-specific policy. The user would receive this protection without being proactive. Within this area one can roughly divide the approaches into those which transform the whole program (thus requiring the program to be parsed) and those which perform wrapping without having to modify the code. Phung *et al* [25] refer to these styles as *invasive* vs *lightweight*, respectively. The former approach is taken by the BrowserShield tool [28] which performs a deep wrapping of code, requiring run-time parsing and transformation of the code. In more recent work, Ofuonye and Miller [23] show that the high runtime overheads witnessed in BrowserShield can be improved in practice by optimising the instrumentation technique. The *lightweight* approach refers to techniques which do not require any aggressive code manipulation. There are many JavaScript programming libraries which provide this kind of functionality; the lightweight self-protecting JavaScript work of Phung *et al* [25] is the only one of these which is security specific. More details of this approach are given below.

A number of approaches involve using well-behaved subsets of JavaScript. These can be though of as a hybrid of an invasive pass (to check that the code is in the intended sublanguage), followed by wrapping. By syntactically filtering the language, the wrapping problem becomes much simpler, since problematic language features can be disallowed (these invariably must include, among other things, all dynamic code creation features such as `eval` and `document.write`). This approach is exemplified in FBJS [12], a JavaScript subset provided by Facebook to sandbox third-party applications. A principled perspective on this approach is provided in the work of Maffeis *et al*, e.g. [19].

Each approach has potential advantages and disadvantages, and each must both overcome numerous technical problems to be practically applicable.

1.2 Self Protecting JavaScript

In this paper we focus on problems and improvements in the *self protecting JavaScript* approach [25]. Here we outline the key ingredients of that approach.

Policies are defined in terms of security relevant events, which are the API calls – the so-called *built-in* methods of JavaScript. These are the methods which have an intrinsic meaning independent of the code itself. The attacker is assumed to have injected arbitrary JavaScript into the body of a web page. A policy is a piece of JavaScript which, in an aspect-oriented programming (AOP) style, specifies which method calls are to be intercepted (the *pointcut* in AOP-speak), and what action (*advice*) is to be taken.

The key to being "lightweight" is that the method does not need to parse or transform the body of the page at all. This is achieved by assuming that the server, or some trusted proxy, injects the policy code into the header of the web page. Integrity of this policy code is assumed (so attacks to the page in transit are not considered). Injecting the policy code into the header ensures that the policy code is executed first, so the policy code gets to wrap the security critical methods before the attacker code can get a handle on them. This is a strikingly simple idea that does not have any particular difficulty with dynamic language features such as on-the-fly code generation. The price

paid for this is that it can only provide security policies for the built-in methods, and cannot patch arbitrary "code patterns" as e.g. the BrowserShield approach.

Phung *et al* implemented this idea via an adaptation of a non security-oriented aspect-oriented programming library. But in a security context the ability to ensure that the code and policy are tamper-proof, and that the attacker cannot obtain pointers to the unwrapped methods is crucial. In this paper we study and fix vulnerabilities of both kinds in the implementation outlined by Phung *et al*, and propose a way to make it easier to write sane policies which behave in a way which is not unduly influenced by attacker code.

We divide the study into issues relating to the generic wrapper code (Section 2), and issues relating to the construction of safe policies (Section 3). Before discussing this work in more detail below, we summarize the attacks which motivate the present work, most of which are either well-known or based on well-known mechanisms:

Prototype poisoning Prototype poisoning is a well-known attack vector: trusted code can be compromised because it inherits from a global prototype which is accessible to the attacker. We address several flavours of poisoning attack:
- *Built-in subversion* Built-in methods used in the implementation of the generic *wrapper code* can be subverted by modifying the prototype object.
- *Global setter subversion* Setters defined on prototype objects are executed upon instantiation of new objects. This opens up for external code to access information in a supposedly private scope. In the case of the wrapper implementation, inconsiderate use of temporary objects leads to compromise. This issue has been discussed previously in the context of JSON Hijacking [24,8].
- *Policy object subversion* Any object implicitly or explicitly manipulated by the *policy code* is vulnerable to subversion via its global prototype. Meyerovich *et al* [20] provide a good example of this attack in the subversion of a URL whitelist stored in a policy.

Aliasing issues A specific built-in may have several aliases pointing to the same function in the browser. Knowing what to wrap given one of these aliases is imperative for the monitor in order to control access to the built-in. Meyerovich *et al* [20] call this *incomplete mediation*. Also, each window instance has its own set of built-ins but can under some circumstances access and execute a built-in of another instance. This sort of dynamic aliasing needs to be controlled so that one instance with wrapped built-ins cannot not access the unprotected built-ins of another.

Abusing the caller-chain When a function is called, the `caller` property of that function is set to refer to the function calling it. The called function can thereby get a handle on its caller and access to and modify part of the information which is supposed to be local to it e.g. the `arguments` property. This implies that if user code in one way or another is called from either a built-in, the wrapper, or from the policy, it could potentially bypass the monitor. This general attack vector is described in the Caja end-user's guide [13] ("Reflective call stack traversal leaks references").

Non declarative arguments If a policy inspects a user-supplied parameter the parameter can masquerade as a "good" value at inspection time, and change to a "bad" value a the time of use. This is because JavaScript performs an implicit type conversion. This attack was already addressed in [25] where it is credited to Maffeis

(see also [19]). It is also the basis of a recently described attack on ADsafe [18]. (This paper significantly extends the defence mechanism of [25] for this class of attack).

2 Breaking and Fixing the Wrapping Code

Upon analyzing the wrapper implementation by Phung *et al*. [25] (see Listing 1.1), we found that it was vulnerable to a number of attacks. In this section we discuss the attacks, potential solutions and how the attacks apply to other wrapping libraries.

```
var wrap = function(pointcut, Policy) {
    ...
    var aspect = function() {
        var invocation = { object: this, args: arguments };
        return Policy.apply(invocation.object,
            [{  arguments: invocation.args,
                method: pointcut.method,
                proceed: function() {
                    return original.apply(...);
                }}]);
    } ...
}
```

Listing 1.1. The main wrapper function in Phung *et al* [25].

2.1 Function and Object Subversion

Since the header is executed before the page is processed, any malicious code in the page will only have access to wrapped methods. But since wrapped methods are executed in the attacker's environment, the attacker can subvert functions that are used in the wrapping function to bypass the policies or extract the original unwrapped methods. As an example, the wrapper in Listing 1.1 uses the `apply`-function to execute the policy and the original method. The `apply`-function is inherited from the `Function` -prototype, which is part of the environment accessible to the attacker. By modifying the `apply`-function of `Function`-prototype an attacker can bypass the execution of the policy or even extract the original built-in. Suppose that the wrapped built-in is the function `window.alert`. The following code (Listing 1.2) illustrates this attack by extracting the original `window.alert` and restoring it. If the monitor were to rely on

```
var recover_builtin;
Function.prototype.apply = function(thisObj, args){
        if (args[0].proceed) args[0].proceed();
        else recover_builtin = this; };
//call the wrapped built-in, so that the wrapper will execute
window.alert('XSS');
//then recover the built-in
if (recover_builtin) window.alert = recover_builtin;
```

Listing 1.2. Illustration of subverting built-in to recover the wrapped method.

inherited properties of objects it could be influenced in a similar way.

To prevent attacker code from subverting objects we can try to ensure that each object reference used in the policy is a local property of the object and not something inherited from its low-integrity prototype. The built-in function `hasOwnProperty` can be used for this purpose (of course the integrity of the function `hasOwnProperty` must be maintained as well). But this approach requires all object accesses to be identified and checked. This is potentially tricky for implicit accesses, e.g., the `toString`-function is called implicitly when an object is converted to a string.

Since the monitor code is the first code to be executed it can store local references to the original built-in methods used in the advice function. Our solution is to ensure that the wrapper code only uses the locally stored copies of the original methods. As an example, `o.toString()` would be rewritten as `original_toString.apply(o,[])`. To prevent an attacker from subverting the `apply` function of the stored methods, it is made local to each stored function by assignment, i.e. `original_toString.apply= original_apply`. Now even if the prototype of the function is subverted, the `apply` function local to the object remains untouched. Again, this is not entirely foolproof since it could be hard to determine which functions are being called *implicitly*.

A simpler alternative approach (supported in e.g. Firefox, Chrome and Safari, but not in e.g. IE8 or Opera) is to set an object's `__proto__` to **`null`**. This has the effect of disconnecting the object from its prototype chain, thus preventing it from inheriting properties defined outside of the policy code. Since they are no longer inherited, any required properties of the prototype must be reattached to the object from the stored originals. This technique is used in the implementation of the function `safe` in Section 3.1.

2.2 Global Setter Subversion

A special case of function subversion involves setters. A setter is a function for a property of an object, that is executed whenever the property is assigned a new value. Defining a setter on a prototype object will affect all objects inheriting from that prototype, which is our definition of a global setter. If a setter is defined for `Object.prototype`, it will be inherited by *all* objects.

An issue that has been discussed recently [32,29] is that global setters will be executed upon object instantiation. This creates an unexpected behavior where external code is able to extract values from a private scope. When considering the code in Listing 1.1, an attacker could define a global setter for the property *proceed* of all objects. The below snippet illustrates this attack in the wrapper in Listing 1.1.

```
1 var recover_builtin;
2 Object.prototype.__defineSetter__('proceed',
3                 function(o) { recover_builtin = o });
```

When the advice is executed, a temporary argument object for the policy is created. Since this object contains a *proceed*-property, the setter will be executed and the function containing the original method will be passed as an argument. The attacker can now bypass the policy by executing the function in the setter. Note also that the argument object as a whole will be accessible to the setter through the **`this`**-keyword. The same holds for any object created in the execution of the advice or in the policy itself. This vulnerability also applies to arrays and functions.

While the correctness of this behaviour is debatable [32], it is implemented in most browsers (at the time of writing). The exceptions are Internet Explorer (which only implement setters for DOM-objects) and Firefox which have recently [29] changed this behavior so that setters are not executed upon instantiation of objects and arrays (although for functions the problem still remains). This issue has been discussed previously in the context of JSON Hijacking [24,8].

One possibility to protect against this problem would be to prevent the wrapping code from creating any new objects, arrays or functions. This severely restricts how the advice function could be implemented, in such a way that it might not be possible to implement at all. Checking for the existence of setters for every property before creating an object is another alternative, but it would be infeasible in general. The advice code could define its own getters and setters on the object instead of just assigning the property a value. The custom getters and setters would overshadow the inherited ones, making the object safe to use. Again this might be a bit too cumbersome.

As mentioned in the previous section, the chain of inheritance can be broken by setting the `__proto__` property to **`null`**. This is our current solution. Developing a solution which works for platforms not supporting this feature would require very careful implementation and is left for future work.

2.3 Issues Concerning Aliases of Built-ins

Although policies are specified in terms of built-in function *names*, semantically speaking they refer to the native code to which the function points. This gives rise to an aliasing problem as there may be several aliases to the same built-in. This is a problem since a crucial assumption of the approach is that wrappers hold the *only* references to the security relevant original functions. This problem is highlighted in [21] (where it is solved by pointer comparison – something that is not possible at the JavaScript level).

Static Aliases Most functions have more than one alias within the window, and if one is wrapped, then the others need to be wrapped as well. Otherwise, the original function can be restored by using an alias. As an example, in Firefox the function alert can be reached through at least the following aliases: `window.alert`, `window.__proto__.alert`, `window.constructor.prototype.alert`, `Window.prototype.alert`. Enumerating these different aliases for each method is browser specific and somewhat tedious, but we conjecture that in most cases there is a "root" object at the top of the prototype inheritance chain for which wrapping of the given method takes care of all the aliases. For a given `object` and `method` this root object can be computed by:

```
1 while(!object.hasOwnProperty(method) && object.__proto__)
2   object = object.__proto__;
```

Any aliases not captured by this scheme must be handled on an *ad hoc* basis. But the main point here is that this should be the job of the wrapping library and not the policy writer. Thus we propose to extend the wrapping library with a means to compute aliases, and ensure that a policy applied to one function is applied to all its static aliases.

Dynamic Aliases Another class of aliases are those which can be obtained from other window object (window, frame, iframe). In [25], several attempted solutions were introduced to deal with the problem, including disabling the creation of new window,

frame/iframe or disable the access to `contentWindow` property of frame/iframe objects from where references to unwrapped methods can be retrieved. Unfortunately the proposed approach seems both incomplete (does not provide full mediation) and overly restrictive. In this work, we allow window objects to be created, but user code should not be able to obtain a reference to one. If user code gets a reference to a foreign window object, even if it is enforced with the same policies, that window object could be navigated to a new location which would reset all the built-ins. To implement this we provide pre-defined policies which enforce methods that potentially return a window object. This boils down to two cases: static frames that are defined as part of the html code, and dynamic frames that are generated on the fly.

For static frames the problem is that they do not exist at the time the policy code in the header is executed, and there is no way to intervene just after they have been created. This means we have to proactively prevent access to an unspecified number of frames that *might* be created. If we disable `contentWindow` for all frames, the only other way for user code to obtain a reference to the window is through the `window.frames` or `window` array. By defining getters for "enough" indices in this array we can fully prevent inappropriate access. The remaining problem is determining how many indices will be enough – here we must rely on some external approximation.

For dynamic iframes a similar approach is used. By wrapping all actions that may result in the creation of an iframe, we can intervene and replace the `contentWindow` property and the right number of indices in the `window` array.

2.4 Abusing the Caller-chain

Built-in subversion The following core assumption is formalised in [25]: *we are effectively assuming that the built-in methods do not call any other methods, built-in or otherwise.* This assumption does not hold for all built-ins, and its failure has consequences. Specifically, (i) some built-ins run arbitrary user functions by design, such as `filter`, `forEach`, `every`, `map`, `some`, `reduce`, and `reduceRight`, and (ii) some built-ins implicitly access object properties e.g. `pop` which sets the `length` property or `alert` that implicitly calls `toString` on its argument. These property accesses can, in turn, trigger arbitrary code execution via user-defined getters and setters.

Both of these cases are problematic because of a nonstandard but widely implemented[1] `caller` property of function objects. For a function `f`, `f.caller` returns a pointer to the function which called `f` (assuming `f` is currently on the call stack). Thus any user code which is called from within a built-in can obtain a pointer to that built-in using `caller`.

As an example, suppose that the `alert` function has been wrapped. In Listing 1.3 the user defines an object with a `toString` which sets `alert` to the function calling it. Now the user code calls `alert(x)`, thus invoking the wrapped `alert` function. Now suppose that the wrapper eventually calls the original `alert` built-in. The built-in will internally make a call to `x.toString`. The modified `toString` can now obtain a reference to the built-in from the caller chain and restore the original built-in.

Wrapper subversion The caller attack does not only apply to built-ins. In several places the wrapper code must traverse user-supplied objects in order to inspect or assign

[1] Not part of any ECMA standard but implemented in all major browsers.

```
1 var x = {toString: function(){ alert=arguments.callee.caller; }
    };
2 alert(x);
```

Listing 1.3. An example of the caller attack

to properties. This might trigger the execution of getters or setters or other user supplied code which can abuse the caller chain to influence the wrapper, extract information, or dynamically change its behavior upon inspection.

Mitigating the caller attack For type-(i) functions this is not a real problem – we simply ban them from wrapping. From a policy perspective the built-ins are really just a way to get a handle on *behaviours*. Functions like those listed are simply programming utilities and have nothing to do with the extensional behaviour of the system at all, and policies have no business trying to control them.

Type-(ii) functions, on the other hand, do indeed involve built-ins that may need to be wrapped, e.g. `document.appendChild`. For each built-in, the wrapper needs to know (an upper bound on) the properties that it might access directly. Before calling the original built-in the wrapper must unset any user-defined getters or setters for the accessed properties before calling the built-in; to preserve functionality these are restored after the built-in returns.

As for subversion of the wrapper, there is no upper bound on which properties might be accessed. Therefore the wrapper must ensure that user code is not implicitly executed when traversing the object. This could be achieved as for type-(ii) functions above, but a simpler approach works in this case. If there is a recursive function on the stack then the caller operation can never get past it. So by wrapping operations on untrusted data in a dummy recursive function, the caller operation can be prevented from reaching the sensitive context.

2.5 Browser Specific Issues

It seems unlikely that one can come up with a solution which works for all browsers. One thorny problem that is specific to Firefox is the behavior of the delete operator which when applied to the name of a built-in simply deletes any wrappers and restores the original method. This problem is discussed in [25], and also plagues the Torbutton anonymous browsing plug-in, which is unable to properly disable access to the `Date` object for precisely this reason [30]. We are not optimistic that there is a workaround for this in the current versions of Firefox, although future versions supporting object attributes from the recent ECMAScript 262 standard [11] will certainly see an end to this problem.

2.6 Other Lightweight AOP Libraries

As an experiment we tried to adjust the attacks to other AOP-wrapping libraries to see if any of them were more suitable candidates for implementing a reference monitor. The libraries used were jQuery AOP [15], dojo AOP [10], Ajaxpect [1], AspectJS [3], Cerny.js [7], AspectES [4], PrototypeJS [27]. One thing to note is that none of these libraries were designed for security purposes, but rather as general implementations of AOP-functionality. The results were discouraging: all of the libraries were vulnerable to all the attacks described above. In addition the way they are designed opens up for

new attacks which had been considered in the design of [25]. For example, since the the wrapping code (the AOP library) is not in the same local scope as the policy code, the library must export its wrapping functions, thus making them vulnerable to simple redefinition from attacker code.

3 Declarative Policies

Let us suppose that the mechanism for enforcing policies provides full mediation of security relevant events. Then all one needs to do is to write policies which enforce the desired security properties. Unfortunately, due to the complexities of JavaScript, this is not a simple task. It is all too easy to write policies which look reasonable, but whose behavior can be controlled by the attacker (who controls the code outside of the policy).

In this section we describe this problem and propose a method to structure policy code which makes them *declarative*, in the sense that code outside the policy and wrapper library cannot have side-effects on the policy.

As a running example consider a policy implementing a URL white-list which is used to filter calls to e.g. `window.open(url,..)`: calls to whitelisted URL's are allowed, other calls are dropped.

3.1 Object and Function Subversion in Policies

In [20] an additional problem with policy subversion is noted. Let us consider the example given there: suppose that the policy writer models a URL whitelist by an object: `var whitelist ={"good.com":true, "good2.org":true}`. Then for a policy, which also allows subdomains of the domains in the whitelist, the code would involve a check similar to the one in Listing 1.4.

```
1 var l = url.lastIndexOf('.',url.length-5) + 1;
2 if (whitelist[url.substring(l)] === true) { ... }
```

Listing 1.4. Policy sample code

This looks like the desired policy, but unfortunately the attacker can easily bypass it by assigning to `Object.prototype["evil.com"]=true;` this will add an `"evil.com"` field to all objects *including the whitelist*. Alternatively the attacker could redefine `substring` to always return a string that is in the whitelist. The *url* would then pass the check regardless of its actual value.

The solution we adopt here is the same as for the wrapper code. For functions the policy writer must use local copies of the originals, and for objects we can ensure that they cannot access a poisoned prototype by simply removing it from its prototype chain.

Let us refer to such an object as a *safe* object. How can we make it easy for the policy author to work only with safe objects? Our current approach is to provide a function `safe`, which recursively traverses an object, detaching it and all sub-objects from the prototype chain that can be modified by the user. As explained in Section 2.1 detaching the object is done by setting its `__proto__` property to `null`. Since detaching implies that the object will no longer inherit any of the methods expected to be associated with the type of the object, this functionality needs to be restored. Since determining the type of an object is difficult the safe function takes an optional argument to specify the type. Safe versions of the functions associated with this type are added to the object. The safe versions of the functions are stored locally and are detached from the prototype

chain to prevent attacker influence. The format of the object type is similar to the types described in Section 3.2. Programming with a whitelist would then be written as:

```
1 var whitelist = safe({"good.com":true,"good2.org":true});
```

The policy writer must, in general, ensure that any object which is accessed is made safe. But objects are also constructed implicitly – for example a string might get implicitly converted to a string object. When this happens the string object in question will be unsafe. Because of this the policy author should apply the `safe` function to all types, preemptively (and recursively) converting all values to (safe) objects.

The question of how to obtain complete and optimal insertion of the "safe" operation in order to avoid all unsafe objects is left for further work. Note that it is not enough to wrap safe around object literals (as we initially believed). Suppose e is some expression which returns a value of primitive type. Now consider the expression e`.toString()`. This is unsafe because in order to apply the `toString` method the primitive type constructed by e is implicitly converted to an object (e.g. a Boolean object). This object is unsafe and thus an attacker-defined `toString` method could return any value. To fix this we could apply `safe` to e, but this would be redundant if e is already safe (by virtue of having being built from safe objects).

3.2 Non Declarative Arguments

Phung *et al* [25] (following Maffeis *et al* [17]) note a problem with inspecting call parameters. In the case of the whitelist example, note that the argument to such a call might not actually be a string, but any object with a `toString` method. Since this object comes from outside, it can be malicious. In the case of the whitelist example it could be a stateful object which returns `"goodurl.org"` when inspected by the policy, but in doing so it redefines its `toString` method to return `"evil.com"` when subsequently passed to the original e.g. `window.open(url,..)` method. Phung *et al* [25] suggested a solution to solve this problem by implementing *call-by-primitive-value* for all policy parameters using appropriate helper functions to force each argument into an expected primitive type. The idea is that the policy writer decides which arguments of the wrapped call will be inspected, and at what type. These arguments are then forcibly coerced to that type before being passed to the policy code, thus ensuring that what you see (in the policy logic) is what you get (in any subsequent call to the wrapped built-in).

Types for Declarative Arguments The approach of Phung *et al* has some shortcomings: (i) it does not provide a clear declarative way for the policy writer to specify the parameters and their intended types; (ii) it only only applies to primitive types and not objects; (iii) it does not deal with the return value of the wrapped function (iv) it relies on the policy writer not accessing unsanitized parameters; (v) it uses functions such as `toString` for implementing coercion, but leaves this function open to subversion.

We provide a policy calling mechanism which addresses these shortcomings. Here we provide a brief outline of the design. The idea is that the policy writer writes a policy and an *inspection type* for the argument and the result. The policy code can assume that the parameters are declarative and the wrapper library will ensure this using an inspection type. An inspection type is a specification of the types of the call parameters that will be inspected by the policy code.

As an example (listing 1.5) suppose we have a policy for the `appendChild` method of the `document.body` object. The argument of the `appendChild` method should be an HTML node object which has several properties and methods. The policy (function `ipolicy`) intends to check whether the argument is an iframe by looking at the property `tagName` of the argument; if so then it should only proceed if the `src` property of the argument is an allowed URL. If the argument is not an iframe it should just proceed. (It should be noted that `tagName` is not reliable enough for this policy, but it suffices as an example.) Code to install this policy using our wrapper constructor is given in listing 1.5 below.

```
1  var ipolicy = function(args, proceed) {
2      var o = args[0];
3      if (o.tagName == 'iframe') {
4          if (allowedUrls(o.src))
5              return proceed();
6      } else
7          return proceed();
8  }
9  wrap(document.body, 'appendChild',        // object and method
10      ipolicy,                             // policy function
11      [{src:'string', tagName:'string'}]); // arg inspection type
```

Listing 1.5. Example of using the wrapper.

The first two arguments of `wrap` specify the object and method to be wrapped (the "pointcut"). The third parameter is the policy function (the "advice") and the fourth parameter is the argument inspection type – a specification of how the parameters will be inspected by the policy. In the example call above we are specifying that only the first argument to `appendChild` will be inspected by the policy code, and it will do so assuming type `{src:'string', tagName:'string'}`. Not shown in the example is an optional return inspection type. This is needed if the policy will also inspect and modify the return type of the wrapped builtin.

The parameter inspection type is an array of types. The following simple grammar of JavaScript literals represents the types used in our current implementation:

$$\begin{aligned}\mathsf{type} ::= {} & \texttt{'string'} \mid \texttt{'number'} \mid \texttt{'boolean'} \mid \texttt{'*'} \mid \texttt{undefined} \\ \mid {} & \{\mathsf{field}_1 : \mathsf{type}_1, \ldots, \mathsf{field}_n : \mathsf{type}_n\}\end{aligned}$$

The `'*'` type provides a reference to a value without providing access to the value itself. We expect that experience will reveal the need for a more expressive type language, such as sum-types and more flexible matching for parameter arrays – but these should not be problematic to add.

Policies are enforced as follows: the inspection type is used as a pattern to create a clone of the argument array. We will call this the *inspection argument array*. This is the generalization of the idea of *call-by-primitive-value*, except that the cloned parameters also *remove* any parts of the arguments which are not part of the type. The policy logic can only access the inspection argument array. However, when passing the parameters on to any built-in function, we permit the function access to the whole of the argument

array. To do this we *combine* the original argument array with the inspection argument array. Figure 1 illustrates this process and Listing 1.6 outlines the code.

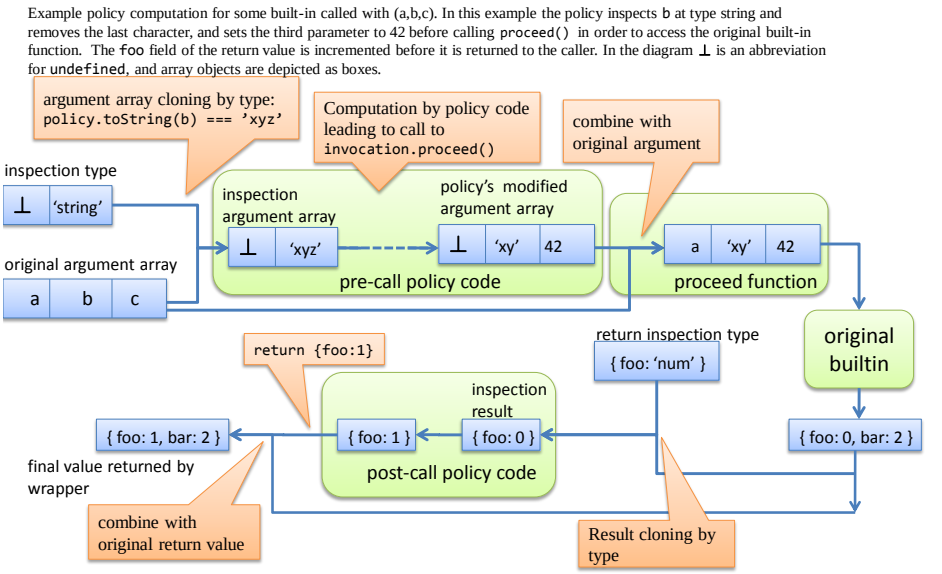

Fig. 1. Illustration of policy parameter manipulation

```
var wrap = function(object, method, policy, inType, retType) {
    // Find function corresponding to alias
    while(!object.hasOwnProperty(method) && object.__proto__)
        object = object.__proto__;
    var original = object[method];

    object[method] = function wrapper() {
        var object = this;
        var orgArgs = arguments, orgRet;
        var polArgs = cloneByType(inType, arguments);
        var proceed = function() {
            orgRet = original.apply(object,
                    combine(polArgs, orgArgs));
            return cloneByType(retType, orgRet);
        }
        var polRet = policy(polArgs, proceed);
        return combine(polRet, orgRet);
    };
}
```

Listing 1.6. Outline of the revised wrapper function supporting inspection types

When cloning, the reference type `'*'` is replaced by a fresh dummy object. When combining, each such object is replaced with original value that it represented. Note that the type language does not include functions. This means that policy code cannot

inspect any function parameters. However, this does not mean that we cannot have policies on built-in functions which e.g. have callbacks as arguments – it just means that we cannot make policy *decisions* based on the behavior of the callbacks. This restriction to "shallow" types does not seem to be a serious limitation, but more experience is needed to determine if this is indeed the case.

The treatment of the return value of the method is analogous to the treatment of the arguments: a return type specifies what the policy may inspect from the return value. If this is not specified then a return type of $\star$ is assumed. The return value of the policy function is combined with the return value produced by the actual built-in.

4 Related Work

This work is based on the lightweight self-protecting JavaScript method [25], which embeds the protection mechanism in terms of security policy into a web page to make the web page self-protecting. Two recent papers [21,20] (concurrent with this present work) also discuss a large subset of the attacks investigated here, but with the purpose motivating a quite different part of the solution space.

Other recent work on JavaScript security includes static and runtime analysis e.g. [5,6,22], code transformation e.g. [28,23,16], wrapping e.g. [21,20], and safe subsets e.g. [19,14]. In this section, we compare our work to more recent related work on enforcing fine-grained security policies for JavaScript execution by wrapping.

Ofuonye and Miller [23] introduced an optimized transformation method to implement wrappers by rewriting objects identified as being vulnerable. Their approach can be viewed as an optimization of the BrowserShield approach [28]. However, it appears that the authors have not considered the vulnerabilities that we discussed in Section 2, and it seems that these attacks can defeat their security mechanism. For example, their transformation method does not protect against Function and Object subversion (cf. Section 2.1). It seems that the solutions described in this paper can be applied directly to their implementation – including not only solutions to function and object subversion, but also e.g. the use of alias-sets to apply a policy consistently across all aliases of a given built-in method.

A similar approach concurrent to our work is *object views* proposed by Meyerovich *et al* [20] that provides wrappers as a library in JavaScript. Object views, however, focus on the safe sharing of objects between two principals in the browser, e.g. between two frames of different origins or privileged code and untrusted code, whereas we focus on controlling the use of built-in methods to mitigate the extensional effects of cross-site scripts. Because policies do not control built-in functions, they need to deal with the flexibility of user defined objects and functions. In order to do so, they provide recursive "deep" wrapping and use reference equality checking of user defined objects to ensure the full mediation of each operation. Meyerovich *et al* also provide a policy system where policy writers can specify policies in declarative rules which is later compiled into wrapper functions.

ConScript [21] is more closely related to our work in the sense that it provides a JavaScript aspect-oriented programming system for enforcing security policies including those studied here. However, as mentioned in the introduction, the realisation of ConScript is different from our work in the sense that it extends JavaScript language with new primitive functions to support aspect-oriented programming and provides safe

methods replacing vulnerable native JavaScript prototype functions. In order to deploy such extensions, the authors have to modify the JavaScript engine (i.e. the browser itself). CONSCRIPT also provides a type system that can be used to validate the defined policies to ensure that the policies do not contain vulnerabilities. This feature is more advanced than our declarative policies since we provide tools for the policy writer to construct sensible policies, but our method does not guarantee the correctness of the policies. A possible extension of our work to include a similar type system is left to further work.

Typed interfaces in JavaScript The use of a typed interface to enable the safe inspection and manipulation of user values is a direct generalisation of the earlier *call-by-primitive value* idea. The use of JavaScript-encoded typed interfaces is not uncommon in Java libraries. For example the Cerny.js library [7] provides a similar type language to the one used here in order to improve code quality and documentation. As mentioned above, the policy language of the ConScript system has a type system that plays an essential role in eliminating a number of security issues such as malicious user objects masquerading as primitive types. But types are only used for type checking. Thus type coercions to primitive types must be added manually to the code where needed in order for type checking to succeed. Our approach is different in that the types themselves are interpreted as coercion operations.

Aspect-oriented programming In the context of aspect-oriented programming for JavaScript, besides the AOP libraries we analysed in Section 2.6 (i.e. jQuery AOP [15], dojo AOP [10], Ajaxpect [1], AspectJS [3], Cerny.js [7], AspectES [4] and PrototypeJS [27]), there have been several AOP frameworks for JavaScript in literature. AOJS [33] is a framework supporting the separation between aspects and JavaScript code where aspects are defined in a XML-based language and then woven to JavaScript by a tool (similar to the proxy-based approach like [28,23,16] reviewed in the introduction). Current implementation of AOJS only support *before* and *after* advice, as the aspect system cannot control the behavior of operations.

Similar to our work (and the self-protecting JavaScript approach), AspectScript [31] is another AOP library for JavaScript that supports richer set and pointcuts in JavaScript. AspectScript also supports stateful pointcuts that is similar to security states in Phung et al [25]. More interestingly, AspectScript provides a library as a weaver tool to transform JavaScript code into aspect-based code and the weaving process is performed at runtime. However, the mentioned libraries or frameworks have not paid attention to securing their aspect systems (see e.g. [9]), thus they are subject to the vulnerabilities that we have presented here.

5 Future Work

Most of the solutions and policy mechanisms presented here have been implemented in JavaScript and a prototype library suitable for the Safari and/or Chrome browsers is available on [26]. A number of more substantial extensions remain to be investigated.

Idiot-proof Policies The current policy language is intended to make it easy for the policy writer to construct sensible policies, but it does not enforce this. A natural extension of this work would be to find ways to guarantee that the policy code does not, e.g. create unsafe objects or use subverted built-in functions. We see two possible directions

to achieve this. One approach would be to provide a proper separation between policy code and attacker code rather than trying to handle this on a per-method and per-object basis as we do here. Another approach is to constrain the way that policies are written, for example using JavaScript sub-languages which can be more easily constrained (see e.g. the ConScript approach and [19]) or by designing a policy language which can be compiled to JavaScript, but for which we can construct a suitable static type system. Recent work by Guha *et al* seems well suited for this purpose [14].

Session Policies Policies should not be associated with pages but with a session and an origin. One issue that we have not addressed in this paper is writing policies which span multiple frames/iframes. This, in general, requires sharing and synchronization of policy state information between frames in a tamper-proof manner.

Acknowledgments This work was partly funded by the European Commission under the WebSand project and the Swedish research agencies SSF and VR. This work was originally presented at OWASP AppSec Research 2010; thanks to the anonymous referees and Lieven Desmet for numerous helpful suggestions.

References

1. Ajaxpect: Aspect-Oriented Programming for Ajax. `http://code.google.com/p/ajaxpect/`. 2008.
2. J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, US Air Force, Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), USA, 1972.
3. AspectJS: A JavaScript MCI/AOP Component-Library. `http://www.aspectjs.com/`. Version 1.1, commercial, 2008.
4. C. M. Balz. The AspectES Framework: AOP for EcmaScript. `http://aspectes.tigris.org/`. Accessed in January 2010.
5. A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52(6):83–91, 2009.
6. A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proc. of the 18th USENIX Security Symposium (USENIX Security 2009)*, 2009.
7. R. Cerny. Cerny.js: a JavaScript library . `http://www.cerny-online.com/cerny.js/`. Version 2.0.
8. B. Chess, Y. T. O'Neil, and J. West. JavaScript Hijacking. `http://tr.im/jshijack`. Accessed in January 2010.
9. D. S. Dantas and D. Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM.
10. dojo AOP library. `http://tr.im/dojoaop`. 2008.
11. Ecma International. Standard ECMA-262: ECMAScript Language Specification . `http://tr.im/ecma2625e`. 5th edition (December 2009).
12. Facebook. FBJS. `http://tr.im/facebookjs`.
13. Google. Attackvectors. `http://code.google.com/p/google-caja/wiki/AttackVectors`. Accessed January 2010.
14. A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. `http://www.cs.brown.edu/research/plt/dl/CS-09-10/`. Accessed in January 2010.
15. jQuery AOP. `http://plugins.jquery.com/project/AOP`. Version 1.3, October 17th, 2009.

16. H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. Javascript instrumentation in practice. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.
17. S. Maffeis, J. Mitchell, and A. Taly. Run-Time Enforcement of Secure JavaScript Subsets. In *Proc of W2SP'09*. IEEE, 2009.
18. S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy'10*. IEEE, 2010.
19. S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *ESORICS*, pages 505–522, 2009.
20. L. Meyerovich, A. P. Felt, and M. 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. To appear.
21. L. Meyerovich and B. 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.
22. Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proc. of Network and Distributed System Security Symposium (NDSS 2009)*, 2009.
23. E. Ofuonye and J. Miller. Resolving JavaScript Vulnerabilities in the Browser Runtime. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 57–66, Nov. 2008.
24. Open Ajax Alliance. Ajax and Mashup Security. `http://tr.im/ajaxmashupsec`. Accessed in January 2010.
25. P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *ASI-ACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, Sydney, Australia, 10 - 12 March 2009. ACM.
26. ProSec Security group, Chalmers. Self-Protecting JavaScript project. `http://www.cse.chalmers.se/~phung/projects/jss`.
27. Prototype Core Team. Prototype - A JavaScript Framework. `http://www.prototypejs.org/`. Accessed in January 2010.
28. C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.
29. The Mozilla Development Team. New in JavaScript 1.8.1. `http://tr.im/newjs181`. Accessed in January 2010.
30. The Tor Project. Torbutton FAQ; Security Issues. `http://tr.im/torsec`. Accessed in February 2010.
31. R. Toledo, P. Leger, and E. Tanter. AspectScript: Expressive Aspects for the Web. Technical report, University of Chile Santiago, Chile, 2009.
32. J. Walden. Web Tech Blog - Object and Array initializers should not invoke setters when evaluated. `http://tr.im/mozillasetters`. Accessed in January 2010.
33. H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: Aspect-Oriented JavaScript Programming Framework for Web Development. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 31–36, New York, NY, USA, 2009. ACM.