Lightweight Self-Protecting JavaScript

Phu H. Phung
Chalmers University of Technology
Gothenburg, Sweden
www.cse.chalmers.se/~phung

David Sands
Chalmers University of Technology
Gothenburg, Sweden
www.cse.chalmers.se/~dave

Andrey Chudnov*
Stevens Institute of Technology
New Jersey, USA
achudnov@stevens.edu

ABSTRACT

This paper introduces a method to control JavaScript execution. The aim is to prevent or modify inappropriate behaviour caused by e.g. malicious injected scripts or poorly designed third-party code. The approach is based on modifying the code so as to make it self-protecting: the protection mechanism (security policy) is embedded into the code itself and intercepts security relevant API calls. The challenges come from the nature of the JavaScript language: any variables in the scope of the program can be redefined, and code can be created and run on-the-fly. This creates potential problems, respectively, for tamper-proofing the protection mechanism, and for ensuring that no security relevant events bypass the protection. Unlike previous approaches to instrument and monitor JavaScript to enforce or adjust behaviour, the solution we propose is *lightweight* in that (i) it does not require a modified browser, and (ii) it does not require any run-time parsing and transformation of code (including dynamically generated code). As a result, the method has low run-time overhead compared to other methods satisfying (i), and the lack of need for browser modifications means that the policy can even be applied on the server to mitigate some effects of cross-site scripting bugs.

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, Programming

Keywords

Language Based Security, Inlined Reference Monitors

*Work performed while the author was at Chalmers

To appear in *ASIACCS'09*, March 10-12, 2009, Sydney, NSW, Australia. ©ACM

1. INTRODUCTION

JavaScript cannot be trusted. The complex nature of the browser architecture, the reflective features of scripting language itself, and the prevalence of dynamic content offers many ways to defeat the built in security mechanisms or to inject malicious code into web pages originating from an otherwise trusted source.

This paper studies a classic approach to imposing a security policy on an otherwise untrusted system via a *reference monitor*. The concept of a security reference monitor [11] is a method to specify and implement secure systems by using a component which intercepts security relevant resource requests and applies a security policy to decide whether to grant such requests. Variants of this idea include the automatic transformation of insecure events - for example to repair known vulnerabilities [25].

In the context of JavaScript one approach to implementing this idea would be to modify the runtime-system of the browser in which the script runs [19]. This gives a great deal of freedom to the implementation, and makes issues such as *tamper proofing* of the monitor mechanism fairly straightforward. On the down side it requires a specific browser architecture to be in place and for the client to actively seek protection.

An alternative approach is to *inline* the policy into the code so that the code becomes *self-protecting*. There has been considerable recent interest in this approach as a way to provide expressive and efficient application-specific security policies for software components e.g. *software fault isolation* [40], and *inlined reference monitors* [38, 13]. There are potential performance advantages to inlining a reference monitor. Certain dynamic checks can be avoided if they can be statically shown to be safe. For JavaScript this is less of an issue since the nature of the language makes it difficult to optimise. However self-protecting script is interesting for a couple of reasons. Firstly it is not dependent on a modified browser technology. This in turn means that the code transformation can take place at any point between the browser and the code producer. For example, a web site delivers a particular piece of dynamic content. The server knows the application and knows that it should only use a fixed number of resources (e.g. dialogs or cookies). Anything beyond this is either due to a programming error or a cross-site scripting (XSS) attack. Using inlining at the server side the server can inline a policy which controls the page's use of resources. Alternatively inlining might be performed by some proxy.

Self-Protecting JavaScript Challenges.

What are the main challenges for implementing self-protecting JavaScript? The key issues are *completeness* and *tamper-proofing*. Completeness – ensuring that all security relevant events are intercepted – is a problem because of the reflection capabilities of JavaScript. Code can be constructed on the fly using *eval*, or written directly to the document itself. Tamper-proofing is the problem of ensuring that the code cannot subvert the monitor mechanism itself – since it lives within the same code base.

The most closely related approach to the one in this paper is the BrowserShield tool [31], which intercepts and transforms JavaScript operations without browser support (related work is discussed more fully in Section 8). Tamper-proofing is handled by dynamic namespace management – ensuring that the namespace for monitor code is disjoint from the underlying program's name space. Completeness is achieved by making the transformation process itself part of the library. Every time an *eval* or a *write* operation is performed on the document the transformer is called recursively on the argument. We call this an *invasive transformation method*: it must traverse and transform all the code – including anything dynamically created. The down side of this – in addition to performance issues – is that attacks are possible by exploiting mismatches between the parser of the transformation method and the parser of the browser itself. See e.g.[21] for a discussion of this, and descriptions of the Samy and Yamanner worms [24] for an infamous example exploiting such a mismatch to evade server-side script filtering mechanisms.

Contributions.

In this article we present a lightweight approach to building self-protecting JavaScript. It is lightweight in the sense that

- it does not require browser modification, so the protection could be applied at the server side or in a client side proxy – or anywhere inbetween,
- it is non invasive: the original code (and any dynamically generated code) is not syntactically modified. The only modification is new code loaded in the header of the page, and
- its implementation is a small and simple adaptation of an aspect-oriented programming library which permits interception of method calls in a clear and direct fashion without the need for any specialised policy languages. The non-policy-specific part of the code is a mere 89 LOC.

Organisation.

The next section presents attacker assumptions and outlines our method to enforce security policy in JavaScript. Section 3 describes the implementation of our enforcement method. Security policy patterns preventing identified attacks are presented in Section 4. A formalisation of the key ingredients of the method is given in Section 5, where we show that security automata can be soundly encoded. Section 6 discusses the evaluation of our method including experiments, efficiency, and overhead measurement. We give discussions with several practical issues and limitations in Section 7 and discuss the related work in Section 8. Section 9 concludes the paper.

2. SECURITY POLICY ENFORCEMENT FOR JAVASCRIPT

2.1 Attacker Assumptions

We assume that an attacker is a malicious user that attempts to inject potentially dangerous JavaScript code into a web page via data entry in the web page, e.g. forum, wiki, blog content such that the scripts will be stored in a web application database. When a user visits the web page, the malicious scripts are loaded from the database and then executed in the browser of the user by the privilege of the web page such that the user becomes the victim of the attack. We assume that the web pages are securely protected and could not be compromised by the attacker, and the attacker could not modify the content of the web pages, for example, by man-in-the-middle attack. We also assume that the web pages are trusted by the users, thus, they allow script execution implicitly and expect their private information to remain safe, e.g. not to leak to a third party.

2.2 The Enforcement Method

In this section we describe the goals of the method and the particular approach we take. The basic approach to policy enforcement in this work is to intercept security relevant events before they are executed. These events can then be permitted to run, rejected, or modified in some other way according to some policy. This is a variation on the classic idea of a *security reference monitor*, and in what follows we will refer to this mechanism as the "reference monitor".

The realisation of this idea will be through a source-to-source transformation of the code. This is sometimes called the *inlined reference monitor* approach [38, 34, 36] – although similar ideas appear elsewhere e.g. [40]. As we discuss later, the transformation could take place on the server, in a proxy, or in the browser itself.

First we must decide on *what* is being monitored. Here we take the view that it is the methods and fields of the built-in objects that are the security relevant events. These are the fields and methods present in the run-time system and having a semantic meaning independent of the program being executed. For JavaScript running in a web browser this basically means the API methods which directly effect the DOM tree.

The reference monitor, based on some security state information which it maintains, decides whether and how to call the original method. Let M range over the security relevant methods. The basic interception method is to create a *security state* needed to record the relevant parts of the computation history, and to ensure any security relevant events go through the monitor, i.e. for each security relevant method M, we

1. Create an alias to the original method
$$M^{\text{orig}} = M$$
2. Redefine[1] the original method to only call the original

[1]Note that there is a design choice here: to overwrite the original method rather than overwrite the call sites with a new wrapper method (as done in e.g. BrowserShield [31]). This has a number of advantages that we will discuss later.

method via a wrapper method:

$$M(\text{params}) = \text{wrapper}_M(M^{\text{orig}}, \text{params}, \text{SecurityState})$$

Consider, for example, the case of a security automaton [33] (a truncation automaton in the terminology of [25]). Here the security state models a state in the automaton. Pseudocode for wrapper_M would be

if permitted(M, params, SecurityState) **then**
  SecurityState := update(SecurityState, M, params);
  M^{orig}(params);
else
  abort;
end if

where "permitted" determines whether the M call is allowed in the current state, and if so the security state is updated and the call proceeds. Otherwise execution is aborted.

The challenges in implementing this approach in a JavaScript context are that anything in the scope of the program can be redefined, potentially including

(i) M^{orig},

(ii) the security state and any of its helper functions,

(iii) The wrapped functions $M(\text{params}) = \text{wrapper}_M(\cdots)$

The solution to (i) and (ii) turns out to be straightforward through appropriate use of scoping mechanisms: the reference to the original built-in method and the security state information are maintained in a local scope visible to the wrapper code [2]. By contrast, BrowserShield [31] uses a runtime "name resolution management" whereby all other variables in the program (including those determined dynamically or defined via reflection at runtime) are renamed so as not to clash with the functions of the protection mechanism library.

How about part (iii)? Although we can make any helper-functions part of a local scope, the new version of M must be globally visible. But JavaScript does not provide any language-level mechanism for making this new definition immutable. Our solution here is to do nothing. We allow the script to freely redefine the any such method M – either within a local scope or globally. This may seem surprising at first. In Section 5 we formalise this approach and show that it soundly enforces the desired policy for one specific class of reference monitors. For the present we explain the rationale informally as follows.

Our policies are concerned with preventing (or modifying) certain behaviours of the built-in methods. The policy is not concerned with methods that happen to have the same name as the built-ins. For example, if we were to monitor the built-in `window.open` method to ensure that there are no more than 2 popups are permitted. The policy refers to the actual native code which performs the `window.open` operation, and is not concerned with the *name* of the method per se. If the program redefines `window.open` then the policy is still only concerned with the native code for the *original* method, and not this new method. This new method is free (under this particular policy example) to be called any number of times.

[2]An alternative idea we tried was to choose obscure random names for the security state. Even if these are chosen after any malicious code is inserted, the reflection capabilities of JavaScript allow the code to determine the obscure names dynamically by code self-inspection.

The key to making this approach safe is the fact that the actual native code for the built-in method in question is *only* accessible via the wrapper method. Although wrapper methods can be overwritten, the reference to the original built-in method is held uniquely by the wrapper method.

3. REALISATION

Here we describe the implementation of the method. The code transformation itself is based on a simple transformation which inserts the policy code and some library code into the header of the page. The exact architecture of this process is not the main focus, and it could be performed by the code producer, at the server, in an application-firewall or proxy, or even as a browser plug-in.

```
1 <html><head>
2   <script src="/policies.js"></script>
3   <!-- file policies.js contains weaving library
4     -- and security policies -->
5 </head>
6 <body>
7 ... <!-- the content of page (unaltered) -->
8 </body></html>
```

Listing 1: The structure of a webpage containing policy enforcement code.

The concrete structure of a web page with an embedded policy is illustrated in Listing 1. The process to embed policies is performed statically at some point between code developer and browser. This static method has advantage since it is simple and does not require any HTML and JavaScript code analysis. The possibility that this could be performed statically on server-side code (or code stubs) has some potential advantages. In particular there is no additional computation for each HTTP request on the server at runtime. This advantage is notable compared with similar approaches such as BrowserShield [31], or CoreScript [44] in which a computation is performed for each HTTP request at a proxy between client and server, and the computation overhead observable by each client depends on the size of web page.

The core issues that must be addressed to make this scheme viable are the requirements that the original methods are "saved" but that these must not be accessible outside the wrapper functions. Furthermore, any additional state variables that are needed should not be modifiable by the rest of the program. These requirements are solved easily by standard JavaScript scoping techniques. Firstly the policy code must live inside a local scope. This can be achieved by placing them in an *anonymous function* which is made to be executed automatically. Secondly we achieve the required privacy of the reference to the original method and the security state by making them local variables (defining by *var*). The lifetime of these variables extends beyond the scope because they are (most likely) used within the wrapper method. The code sketch in Listing 2 illustrates a suitable structure for wrapping a single method called "BuiltIn".

The end result of this structure for the protection initialisation code is that the only variables visible to the attack code are the new versions of the built-in functions – which we freely allow the code to overwrite.

Another concern might be that the attacker attempts to inject code (*e.g.* via `document.write`) into the page that

```
(function() { // Begin local scope
   // Save ref to orig method
   var OriginalBuiltIn = BuiltIn;
   // Define local state
   var SecurityState = ...
   // Redefine the BuiltIn:
   BuiltIn (..) = ... OriginalBuiltIn(..) ...
})();// End local scope
```

Listing 2: Protection Initialisation Code

preempts the execution of this protection initialisation code. In our model, policies defined in aspects are located in the first `<script>` tag within the `<head>` element of the document. Assuming that web server is not compromised, attackers cannot inject scripts into the `<head>` element.

Aspect-Oriented Programming.

Instead of implementing the above schema directly, an elegant and robust implementation is provided the aspect-oriented programming paradigm [23]. In this programming style, an *aspect* comprises a *pointcut*, which defines the point and the condition under which the aspect modifies the behaviour of an application, and an *advice*, which defines what modifications should be applied. The process of combining the pointcut with the advice is known a *weaving*. Using this terminology, our pointcuts correspond to the built-in method calls, and our advice is the policy code. Following this correspondence, our implementation, illustrated in Listing 3, is an adaptation the so-called "around" advice from an off-the-shelf library [1]. Modulo some slightly more elaborate nesting of scope, our code uses essentially the schema defined in Listing 2.

```
var wrap = function(pointcut, Policy) {
  //Override the prototype of object if available
  var source = (typeof(pointcut.target.prototype)
                != undefined)?
      pointcut.target.prototype : pointcut.target;
  var method = pointcut.method;
  //Save reference to the original method
  var original = source[method];
  //Weave the policy with the original method
  var aspect = function() {
    var invocation =
                {object: this, args: arguments };
      return Policy.apply(invocation.object,
        [{arguments:invocation.args,method:method,
         proceed : function() {
          return original.apply(invocation.object,
                                invocation.args);} }]);
  }
  //Redefine the method
  source[method] = aspect;
  return aspect;
}
//Interface to weave an object method call with
//a policy function
var enforcePolicy= function(objectMethod, Policy){
    return wrap(objectMethod, Policy);
}
```

Listing 3: The main wrapper function

3.1 Enforcing Policies on Methods

We define the `enforcePolicy` interface (illustrated in Listing 3) to weave object method calls with a policy function:

enforcePolicy(Map pointcut, Function policy(invocation));

`Map pointcut` is a map to define the built-in method call which we intend to monitor and/or modify. The map consists of target object and method (name of the method to be monitored by the policy). `Function policy(invocation)` is the definition of the policy function. This is the wrapper code that will replace the built-in method. Note that the arguments to the method call are not mentioned by the interface `enforcePolicy`. This is because they can be accessed via JavaScript's `arguments` property. Thus the parameters of the call (via `invocation.arguments`) can be used by the policy to decide the security response. Inside the policy function, the execution of the target method can be controlled. The method execution can be allowed to proceed by calling `invocation.proceed()`. Listing 11 provides an example policy function that can monitor and tune popup parameters at runtime to ensure that new popup window contains a location bar and status bar in order to prevent potential forgery attacks.

3.2 Policy Definition: the Security State

Suppose, for example, that a web application developer knows what behavioural properties a given script should satisfy. Based on that, to define a security policy, the developer first identifies the *security relevant built-in methods and properties* which are the subject of the policy. Then he must define the security policy (as a function) and weave the policy with the target method by calling the weaving function `enforcePolicy(..)` as mentioned in Section 3.1.

JavaScript variables might be used to store security states for security decisions in security policies. For example, a policy like *"the application should never raise more than two popup windows"* concerns the built-in method `window.open` and needs a security state variable to count the number of popups so far.

If the execution is in a safe state (i.e. the popupCount < 2 in this policy), the method is allowed, otherwise the method is suppressed (an alternative could be to abort the whole program). This policy is defined in Listing 4. This policy function invokes the function `antiForgeryPolicy` in Listing 11 which prevents potential forgery attacks when opening a new popup window.

Security states are stored in an array variable. As mentioned previously, the security state variable is not accessible outside the scope containing the policy functions. Similarly any helper functions that are used to process the security state variables must be offered the same protection. For example, the policy in Listing 4 uses a supporting function `AllowedURL(url)` which returns `true` if the provided URL is in the URL whitelist allowed by the policy. If a malicious code redefines this function so that it always returns `true`, the policy is always bypassed. To prevent this problem, supporting functions must be located within the anonymous function.

3.3 Policy Parameter Integrity

In addition to the security state, the parameters of a given call can be used to determine the appropriate policy action. Those parameters which are potentially inspected by the policy code will be referred to as *policy parameters*. From the perspective of the policy code, the remaining parameters are generic: they are simply forwarded in some way or discarded.

One reasonable assumption is that policy parameters are primitive types – primarily *Strings*, *Numbers* and *Booleans*.

```
1  ( function(){
2     // init and assign security states
3     SecurityStates.init();
4     SecurityStates.setState('popupCount',0);
5     //define security policies
6     enforcePolicy({target:window,method:'open'},
7       function(invocation) {
8         antiForgeryPolicy(invocation);
9         var url = stringOf(invocation,0);
10        var popupCount =
11            SecurityStates.getState('popupCount');
12        if ((popupCount < 2) && AllowedURL(url)){
13          SecurityStates.updateState('popupCount',
14                                     popupCount+1);
15          invocation.proceed();
16        }
17        else policylog('Popup suppressed');
18       }
19     );
20   }
21 )();
```

Listing 4: A popup security policy example: allow at most 2 popup windows that its URL must be in the whitelist and the new popup window must has location bar

However, treating them if they were primitive types is risky[3].

Consider, for example, the function `AllowedURL(url)` mentioned above. The `url` is a policy parameter. The policy checks whether the provided URL is in a whitelist. However, although we treat `url` as a string, it could well be any object with a `toString()` method. The problem here is that the `toString()` method is potentially impure: it can return a different value each time it is called. In particular, it could be defined so as to return a "good value" when being checked by the policy code, but to return a "bad value" when passed to the actual built-in. This problem is exemplified in the below example:

```
1 var url={toString: function (){
2                      this.toString =
3                        function (){return "bad";};
4                      return "good";}};
5 alert(url); // alerts with "good"
6 alert(url); // alerts with "bad"
```

The idea to avoid this problem is to implement *call-by-primitive-value* for all policy parameters. We have implemented helper functions to force the arguments into appropriate primitive types. An example of such a function forcing the parameter into string type is implemented as follows:

```
1 function stringOf(inv,argIndex){
2    var tmp = inv.arguments[argIndex].toString();
3    inv.arguments[argsIndex] = tmp;
4    return tmp;
5 }
```

The parameters are the invocation object and the argument index of the intended string parameter. The `stringOf` procedure grabs the current string from the argument and modifies the argument to be that primitive value.

For example, in Listing 4, we make sure that the *url* variable in line 9 is passed by string primitive type by calling `stringOf(invocation,0)`.

3.4 Monitoring Object Properties Access Events

So far we have only discussed policies involving built-in methods. Another important class of security relevant events are *property accesses*. We monitor the property access events using the fact that property access (read/write) of each object can be defined by `__defineGetter__(...)`, and `__defineSetter__(...)` methods of the prototype of the object [5][4].

Our library defines a helper function `MonitorProperty(..)` that represents the new wrapped definitions for the function *getter* and *setter* for the corresponding objects. The interface of the function is provided as following:

```
MonitorProperty( object, property,
                 Function policyforGetter,
                 Function policyforSetter);
```

The functions `policyforGetter` and `policyforSetter` are embedded to the getter and the setter respectively so that each function will be invoked when corresponding property access event is executed. Listing 5 illustrates the use of a property access monitoring. This might be used in implementing a policy which e.g. prevents certain network sends after a cookie has been read.

```
1 SecurityStates.setState('cookieread',false);
2 MonitorProperty('document','cookie',
3             function(){updateCookieState();},
4             function(){}
5 );
6 var updateCookieState = function(){
7  SecurityStates.updateState('cookieread',true);
8  SecurityStates.updateState('sensitiveread',true);
9 }
```

Listing 5: Monitoring cookie read and write event.

The argument for the soundness of this approach is essentially the same as for the treatment of methods. In practice, the attacker could redefine these functions for any properties of any objects, thus, the property access weaving might be overwritten and replaced. Just as for built-in methods, the attacker can never access the original (real) value of the property without going through the monitored getter or setter, and by the same reasoning as for methods, if the getter and setter methods get redefined then this does not matter – the actual property accesses will still be mediated by the policy wrapper.

For properties `document.location` and `window.location`, this monitoring method is inapplicable in practice since they can not be controlled via the setter and the getter. In this case, the method `watch(property, handlerfunc)`, inherited by every object descended from `Object` [5] and adding a watchpoint to a property of the object, could be used to control the property access event. A policy using `watch(...)` to prevent leakage of sensitive data by changing the location of the window is illustrated in Listing 8.

3.5 Attacks to the Unique Reference Property

The key requirement for the correct functioning of our method is the property that the wrapper method holds a unique local reference to the original built-in method. However, this property can be broken using two techniques, one

[3]Thanks to Sergio Maffeis for alerting us to this issue.

[4]This feature is Mozilla-specific, however, it has been specified in the current draft proposal for the next generations of JavaScript (ECMA-262 3.1 [3, 42]).

general and one platform-specific.[5] We present these issues below and discuss methods to prevent such attacks.

3.5.1 Restoring built-in methods from another page

By creating a new `window`, `frame` or `iframe`, the script can manufacture a pointer to the original built-in methods by copying them from their child contexts. Code in Listing 6 illustrates this attacks.

```
1  //create the pointer to a new window context
2  var win = window.open("");
3  //and restore the built-in method window.alert
4  window.alert=win.window.alert;
5  //create the pointer to a new iframe context
6  ifrm = document.createElement("iframe");
7  document.body.appendChild(ifrm);
8  //and restore the built-in method window.alert
9  window.alert = function(s) {
10    return ifrm.contentWindow.alert(s) };
```

Listing 6: Examples of restoring built-in methods.

There are a couple of alternatives to prevent this attack, and one of these must be enforced as part of any policy.

The first solution is fairly heavy handed, and involves a policy which simply disables creation of new `window`, `frame`, or `iframe` contexts. An example of such a policy is defined in Listing 7.

```
1  enforcePolicy({target:document,
2                 method:'createElement'},
3    function(invocation) {
4       var str = stringOf(invocation,0);
5       if(str.indexOf('iframe')>=0) {
6            return;
7       }else invocation.proceed();
8     }
9  );
```

Listing 7: Policy to disallow iframe creation

The second alternative solution is more liberal. Since the way to recover the original built-in method of `frame/iframe` objects must be via `contentWindow`, we can also can prevent the attack simply by using a policy which disables these calls. It seems that this may allow many typical uses of frames/iframes. However, at the time of writing we have not performed any empirical investigations to determine the true impact of disabling `contentWindow`.

3.5.2 Mozilla's delete operator

An implementation specific problem is caused by one particular implementation of the *delete* operator. The delete operator was implemented in JavaScript 1.2, NES3.0 and specified in ECMA-262 [3, 42, 5]. The operator can be used to delete an object, an object's property or an element at a specified index in an array. In Mozilla's implementation of this operator there is a twist. Deleting removes a method (and all previous versions of a method) from an object *except* for methods of built-in objects. In that case the Mozilla flavour of delete *restores the original built-in method*. Thus since our wrapper methods are not built-in, they are deletable and the deletion exposes the original built-in.

[5]The authors are grateful to Jonas Magazinius for pointing out these attacks.

Unfortunately our method cannot prevent this attack; we cannot simply disable delete since it is an operator of the language and not a method. However, we believe that the problem is both Mozilla specific and likely to be a short-term one. In the current draft proposal for the next generations of JavaScript (ECMA-262 3.1 [3]), there is a strong case made for user definable attributes. One such attribute is the *Configurable* attribute (the converse of what was previously called *DontDelete*). By setting Configurable attribute to *false* for our wrapper methods, we can prevent their deletion. Note that the more "obvious" attribute, *Writable* (previously known in converse form as *ReadOnly*), is less suitable for our purpose. Making a wrapper method read-only will break many programs which wish to redefine built-in methods. More importantly, it would no longer permit us to use our method compositionally to apply multiple policies to the same page. This might happen if e.g. policies are applied at different points in time. The *Configurable* attribute, on the other hand, is no more and no less than we need.

4. SECURITY POLICY PATTERNS

So far we have described the bare-bones idea of how to write policies. In this section we flesh this out with a number of examples and patterns of use.

As any security reference monitor, our mechanism is capable, amongst other things, of ensuring any safety property of program execution [33]. This means that the policy can allow or suppress security-relevant actions based on the execution history. Given that the set of such actions is the whole DOM API [6] our vocabulary of policies is rather large. The ability to write policies in JavaScript also adds to the expressiveness in writing policies.

Ideally it is the (non-malicious) application developer who should specify the policies. Knowing the intended functionality of the application is the best starting point for restricting the usage of certain JavaScript features without adversely affecting the function of the application. As it is common in security, there is always a trade-off between functionality and security. We believe that the developer is in the best position to specify meaningful security policies that would maximise both functionality and security of the application.

It turns out that writing sensible policies for JavaScript programs is not an easy task, in particular, because DOM API is large and contains a lot of methods that could be abused and lead to compromising application security. We have found that, instead of trying to come up with a single policy for an application, it is often simpler and more reliable to synthesise it from several small and understandable policies. It is important to make sure, though, that these policies have different *areas of concern*, i.e. the access each DOM API property or method is restricted by at most one such policy. Our mechanism makes it natural to specify the simple policies at the property/method granularity. The resulting application policy would be the product of these policies, hence we call it a *product policy*.

Here we have identified common attacks involving malicious JavaScript code and presented several *security policy patterns* that would help to prevent these attacks. We note that these patterns are not universal, and the actual pol-

[6]Document Object Model Application Programming Interface

icy may and should be specific to the web application in question.

Leakage of sensitive data.

One of the great threats of malicious JavaScript code is leakage of sensitive data. By design JavaScript programs running in the browser do not have access to the file system or other system resources. Still there are several sources of data in the browser itself that could contain sensitive or private information. Although we cannot directly encode information flow policies (and see no "lightweight" way to do so), we can at least control information flow at the endpoints. The general policy to prevent data leakage is to monitor reading from sensitive data sources and prevent operations that could send information to a third party. The sources of sensitive information are:

- the browser cookie, stored in the `window.cookie` and `document.cookie` properties, as it may contain user specific information such as user name, identifier, session identifier etc.,
- the history object, stored in the `window.history` property, as it allows to browse the history of visited webpages thus affecting user privacy,
- the values of `window.location` and `document.URL`, as well as `document.referrer` containing the URLs of the current and previous pages visited since leaking this data may too affect user privacy,
- the values of form elements, potentially containing private data entered by the user, and
- the web page contents.

The means for leaking information to a third party include:

- redirecting to another website by changing the `document.location` and `window.location` properties. The sensitive information to be leaked could be encoded in the URL, e.g. in the query part of the URL `http://evil.com/leak.cgi?data=password:qwerty`,
- changing the source location of the instances of `Frame`, `IFrame`, `Image` and `Form` classes to a new location, similar to described above.

```
1  document.watch('location', locationPolicy);
2  window.watch('location', locationPolicy);
3  function locationPolicy(id, oldloc, newloc){
4    var loc = newloc.toString();
5    if (sensitiveRead()){
6       policylog("Redirection is suppressed:
7                  potential data leakage");
8       abort();
9    }
10   if (!AllowedURL(loc)){
11      policylog(loc+ " is forbidden");
12      abort();
13   }
14   return loc;
15 }
```

Listing 8: Policy controlling the redirection of a webpage.

The general policy that helps preventing leakage is structured as follows:

- reading from the sensitive sources should be monitored. The wrapper function would set an appropriate security state when such a property is accessed to indicate that the JavaScript program has performed a read operation on sensitive data. Above sensitive properties would be monitored by invoking the function `MonitorProperty(...)` as illustrated in the same task in Listing 5 in Section 3.4.

```
1  var onLoadPolicies = function() {
2    var IMGs = document.images;
3    if (!IMGs){ policylog('no images'); return; }
4    for (var i=0;i<IMGs.length;i++){
5      IMGs[i].watch('src', IMGPolicy);
6    }
7  }
8  var IMGPolicy =function(id, oldsrc, newsrc){
9    var src = newsrc.toString();
10   if (sensitiveRead()){
11      policylog("Image changing is suppressed:
12                 potential data leakage");
13      abort();
14   }
15   if (!AllowedIMG(src)){
16      policylog(src+ " is forbidden");
17      abort();
18   }
19   return src;
20 }
21 window.addEventListener("DOMContentLoaded",
22                         onLoadPolicies, true);
```

Listing 9: Policy preventing leakage of information through loading of new images

- writing to the `document.location`, `window.location` and the `src` property of the instances of `Frame`, `IFrame`, `Image` and `Form` classes should only be permitted if the sensitive data fields have not been previously read or the new location is in an URL whitelist allowed by the policy. However, these issues are treated differently. For example, we monitor the change of the location by invoking the method `watch` for the properties `document.location`, `window.location` as mentioned in Section 3.4. Listing 8 gives a policy that prevents redirection attack by changing the location property after sensitive data sources have been read and the new location is not on the allowed list. The function `sensitiveRead()` is a supporting function that monitors security states corresponding to sensitive property read events and returns `true` if one the security states is `true`. For changing image source event, we also use method watch to monitor changes to the document location. Listing 9 defines this policy.

Impersonation attacks.

Another attack that became common with the emergence of AJAX web applications is the impersonation attack. The idea behind the attack is for the malicious JavaScript code to use `XMLHttpRequest` – the very same tool used to build AJAX applications – for sending HTTP requests to the server on behalf of the user. An attacker can therefore mimic a legitimate user and generate requests that appear to be from that user. This could lead to undesired actions to be performed in the context of the web application on behalf of the user. The idea behind the policy preventing such a scenario should involve co-operation with the server side as

```
var XMLHttpRequestURL = null;
enforcePolicy({target:XMLHttpRequest,
     method: 'open' }, function(invocation){
     XMLHttpRequestURL = stringOf(invocation,1);
     return invocation.proceed();}
);
enforcePolicy({target:XMLHttpRequest,
    method: 'send'},
    function(invocation){
      XMLHttpRequestPolicy(invocation);}
);
var XMLHttpRequestPolicy =
    function(invocation){
      //allow the transaction if the
      // URI is in the whitelist
      if (AllowedURL(XMLHttpRequestURL))
          return invocation.proceed();
      policylog("XMLHttpRequest is suppressed:"+
            "potential impersonation attacks");
    }
```

Listing 10: Policy preventing impersonation attacks using XMLHttpRequest object.

well. In particular, if the web application conforms to a W3C recommendation [20] that requires the *GET* request handlers to be pure, i.e. have no side effects, with respect to user data, we can require the application to do *POST* requests only using HTML Forms. Prohibiting *POST* requests prevents malicious scripts from impersonating the legitimate user.

Additionally the web application developer could filter the URL parameter when the `XMLHttpRequest.open` gets called. For example, he can allow for only a certain subset of web-application services to be accessible with `XMLHttpRequest`. Listing 10 defines a policy that can prevent the potential impersonation attacks using `XMLHttpRequest` object. This policy allows `XMLHttpRequest` object to open and send data to an URI if it is in the whitelist of the policy.

Forgery attacks.

Forgery attacks are a subcategory of phishing attacks [2] where the attacker tries to lure the user into believing that either the attacker's website belongs to a legitimate company and/or that all the interactions are performed securely with the legitimate website. These attacks often involve some JavaScript tricks (but in general not necessarily JavaScript-based). For example, attackers can use JavaScript to open a new window without the location bar. To prevent these attacks, we could define a policy to enforce invariants, e.g. always enable the location bar and status bar on the user interface of a new open window. The function defined in Listing 11 illustrates this policy. This function has to be woven with the open new window method call (`window.open(...)`) as we have illustrated in Listing 4.

Another example of a forgery attack is to inject a malicious hyperlink. This can lead to a phishing attack by using IP verses hostname or encoding the link to evade server filtering (c.f. [7]). To deal with this problem, we can disable links in the document that are not in the whitelist link of the policy as in the `checkLinks` function illustrated in Listing 12. This function is invoked by the function `onLoadPolicies()` in Listing 9 to make sure it will be executed before the document is loaded.

```
var antiForgeryPolicy = function(invocation){
  var opts = stringOf(invocation,2);
  if (opts.indexOf("location=no") >= 0){
    opts.replace("location=no","location=yes");
  }else{
    if (!(opts.indexOf("location=yes") >= 0)){
        opts = opts + ",location=yes";
    }
  }
  if (opts.indexOf("status=no") >= 0){
      opts.replace("status=no","status=yes");
  }else{
      if (!(opts.indexOf("status=yes") >= 0)){
          opts = opts + ",status=yes";
      }
  }
  invocation.arguments[2] =opts;
}
```

Listing 11: Policy preventing potential forgery attacks.

```
var checkLinks = function(){
  var links = document.links;
  if (!links){ policylog('no links'); return; }
  for(var i = 0; i < links.length; i++) {
      if (!AllowedLinks(links[i].href))
         links[i].href ="javascript:"+
                        "alert('disabled link')";
  }
}
```

Listing 12: Policy preventing potential forgery attacks by injecting bad links.

Resource abuse.

Client-side resource abuse in JavaScript might seem rather harmless compared to the attacks already mentioned, but still they can adversely affect user experience to the point that the application becomes unusable. A simple example is a program displaying alert messages in an infinite loop. There are certain methods in the DOM API that allow manipulation with the browser window size and location[7], as well as creation of new windows[8] and displaying pop-up alert messages (`window.alert(..)`, `window.confirm(..)` and `window.prompt(..)`). We regard these methods as outdated since modern web application provide consistent user interface without using these methods. However, if the application developer needs to employ some of these methods, this policy could be relaxed and restrict the number of times a certain method is called, or restrict its frequency relative to other events. For example no web application would need to display more than two pop-up alerts in a row, as illustrated in Listing 4 in Section 3.2. Listing 13 illustrates the policy that disallows the above mentioned methods if they could cause resource abuse problems.

5. SOUNDNESS

In this section we attempt to address, from a formal point of view, the correctness (soundness) of our method. What are the formal properties of the method described here? This is a difficult question to answer with any rigour since the

[7]Examples are `window.moveBy(..)`, `window.moveTo(..)`, `window.resizeBy(..)`, `window.resizeTo(..)`, `window.scrollBy(..)` and `window.scrollTo(..)`

[8] `window.open(..)` and `window.createPopup(..)`(only specific to Internet Explorer)

```
var deniedPolicy = function(invocation){
    debug('Method is disabled');
    return null;
}
enforcePolicy({target:window,method:'alert'},
    function(invocation) {
      deniedPolicy(invocation);
    }
);
enforcePolicy({target:window,method:'prompt'},
   function(invocation) {
     deniedPolicy(invocation);
   }
);
//...
```

Listing 13: Policy to disable methods that might cause resource abuse.

problem deals with two complex artifacts, the web browser and the JavaScript programming language. Formalising the semantics of JavaScript is itself a research problem. Nevertheless we can ask about what security policies we are trying to enforce here. The natural class of security policies that we can enforce are the *edit automata* properties [25]. But in order to be concise, in this section we will instead study the simpler class of *security automata properties*, which correspond to the classic notion of *safety properties* [33]. Since our formalisation is rather lightweight its main contribution is to clarify the assumptions that we make about the implementation and platform.

We will proceed as follows. We define an abstract semi-formal model of program execution and prove that the proposed technique, under certain implementation assumptions (that we believe we have satisfied in our particular implementation), is able to soundly inline any security automata over the language of built-in methods. Here the notion of soundness means that if the transformed program exhibits a particular behaviour, then that behaviour is indeed one of the permitted behaviours of the automata specification.

5.1 Computation Model

We begin by defining an abstract computational model. We will not formalise any programming language details (for JavaScript that is a nontrivial task [26]), and thus the connection between simple commands and their (obvious) effect on the computation state will be left informal.

We assume a set of runtime machine *configurations*, Config, ranged over by C, C_1, C_2 etc., which represent the complete information about the computation state of the interpreter and program at any point in time during the computation. Computation is modelled via two kinds of transitions:

- $C \xrightarrow{M} C'$ meaning that from configuration C the native code for a built-in method M is executed, leaving the machine in configuration C'.
- $C \rightarrow C'$ meaning that the machine makes an internal computation step from C to C'.

We define $\xrightarrow{*}$ to be the reflexive and transitive closure of $\rightarrow$, and for any sequence of built-in method names $\vec{M} = M_1, \ldots, M_k$, we write $C \stackrel{\vec{M}}{\Rightarrow} C'$ to mean that

$$C \xrightarrow{*} C_1 \xrightarrow{M_1} C_1' \xrightarrow{*} \cdots \xrightarrow{*} C_k \xrightarrow{M_k} C'$$

By assuming that the transition (1) is a *complete* execution of some native code we are effectively assuming that the built-in methods do not call any other methods, built-in or otherwise.

Furthermore, we suppose that it is possible to identify when a given configuration C is just about to execute the body of a particular non-built-in method d. in that case we say that C *calls* d.

We will also assume that we can extract the value of a particular variable x in C and we will denote this by $C.x$. This notation is somewhat ambiguous but we will use it only to refer to a particular local variable – the representation of the security state.

For a program P let $init(P)$ denote the initial configuration for execution of program P.

Definition 1. We say that P has a run $\vec{M}$ if $init(P) \stackrel{\vec{M}}{\Rightarrow} C$ for some C.

5.2 Security Automata

Security automata [33] are a means to specify safety properties of computations. We can view such an automaton as a reference monitor which determines which transitions should be allowed in any given state. Here we introduce security automata specialised to the alphabet of built-in method names: a security automaton $\mathcal{A}$ is a triple $\mathcal{A}(Q, q_0, \delta)$, where Q is a set of states, $q_0 \in Q$ is the initial state, and $\delta \in Q \times M \rightarrow Q$ is the (partial) transition function. If the automaton is in state q and is willing to accept symbol M and evolve to state q' then $\delta(q, M) = q'$. We view every state of the automaton as an accepting state, and write $\vec{M} \in \mathcal{A}$ to mean that word $\vec{M}$ is accepted by $\mathcal{A}$.

It is convenient to extend the transition function $\delta(q, \cdot)$ to words of by defining, inductively, that $\delta(q, \epsilon) = q$, where ϵ is the empty word, and $\delta(q, \vec{M} \cdot N) = \delta(\delta(q, \vec{M}), N)$. Note that $\vec{M} \in \mathcal{A}$ if and only if $\delta(q_0, \vec{M}) = q$ for some q.

5.3 Self Protected Programs

For simplicity's sake we will assume that the states and transition functions of an automaton are directly representable in a program. In practice there may be a nontrivial encoding layer. See for example [10] for a more formal proof involving a bigger representation distance between the automaton and the encoding.

Definition 2. We say that P is *self-protected with respect to automaton* $\mathcal{A}$ if P begins its execution with the following steps:

(E1) P defines a local alias M^{orig} for each built-in M.

(E2) P initialises a security state variable q to value q_0.

(E3) P then redefines each built-in method M to be
$M(args) = q = \delta(M, q)$;
if $q \neq$ undefined **then** M^{orig}(args) **else** abort;

where abort is any program which prevents further built-ins from being called , and P satisfies the following runtime properties:

(I1) References M^{orig} are the unique references to the built-in methods throughout the execution of the program;

(I2) The security state q is only ever changed by execution of the definitions constructed in step 2 above, and the transition function δ referred to in the body of the definition is unmodified throughout the execution.

Note that the implementation properties I1 and I2 could be proven rather than assumed if we had a suitable model of JavaScript and a precise definition of the code corresponding to E1–E3 (as described in Section 3).

Now we are in a position to state the soundness of self-protecting programs.

THEOREM 1 (SOUNDNESS). *Let P be self-protected with respect to automaton $\mathcal{A}$. If P has run $\vec{M}$ then $\vec{M}$ is accepted by $\mathcal{A}$.*

PROOF. See the tech report version of this article [29] □

One might hope to prove more than just soundness. For example if the program always aborts then it is sound. One notion that has been used in the context of reference monitors in general is *transparency*. To talk about transparency we need to compare the "original" program with the self-protected version. Transparency means that if any run $\vec{M}$ of the original program is accepted by $\mathcal{A}$ then $\vec{M}$ will be a run of the self-protected program. We cannot prove this based on the assumptions we have made, and it turns out that transparency does not hold in practice. We discuss this further in Section 7, where we argue that the lack of transparency is only through programs exhibiting pathological behaviour.

6. EVALUATION

To evaluate our method, we have conducted experiments to verify the effectiveness of our mechanism against known script injection attacks. Experiments attempting to break policy enforcement have also performed to demonstate the policy protection. We have also measured the overhead of the security policy enforcement mechanism. As expected, our mechanism can prevent most of known script injection attack vectors.

Our uncompressed base library contains only 89 LOC in JavaScript with the size of 2.5KB. The code of general security policies we have identified in Section 4 has 236 LOC with the size of 9.3KB, thus in this experiment the total code embedded into each page for policy enforcement has 325 LOC. These code, benchmarks, and test cases are available at [29].

6.1 Effectiveness

We have conducted the experiments in several approaches. First, a test suite of XSS attack vectors introduced in [7] was implemented and run to select successful attacks for different web browsers. Since the method to weave property access events with policies is specific to Mozilla's version of JavaScript, most of the tests were performed in Firefox. As expected, our policy enforcement solution can defeat most of the identified XSS attacks. Out of the over one hundred attack vectors published on `ha.ckers.org` [7], 38 vectors can be launched successfully in Firefox 3.0.1. Our policy can detect and prevent 34 of these vectors. The remaining vectors employ `meta` and `iframe` tags and could not be detected and confined by the policy. The reason is that these attacks could force the browser to load an external HTML document, and all the JavaScript code in that document would be executed in a context of a different DOM tree. This issue is discussed in Section 7.

Besides the simple tests, we have also conducted experiments on real world web applications to verify that our solution is also suitable for defeating real world script injection attacks. We installed a version of a well-known web-based bulletin board system phpBB 2.0.18 [12] known to be vulnerable to cross-site scripting attacks and launched the known attacks. We also experimented with WebCal 3.04 [14] (a web-based calendar application). The bulletin board system is most vulnerable to *stored XSS attacks* where injected scripts are stored in the database of the application and the attack could launched at a later time whenever the user loads the attacked page. In addition to studying attacks identified in [12], we allowed script tags in messages of the bulletin board and successfully mounted the attacks identified in Section 4.

Attacks on WebCal [14] fell in a different category of *reflected XSS attacks*. With reflected attacks it is possible to directly manipulate the contents of the web-page by providing certain inputs (think of a search engine diplaying the search query at the top of the search results). This type of attack does not need a persistent storage to be involved (see e.g. [16] for real examples). We have applied our solution to these web applications by embedding the desired security policy file in an appropriate place[9] to prevent against these two types of attacks. For the phpBB web application, since the description of the board is always displayed, we managed to embed the security policy file in the description of the site via the administration tool and, thus, without modifying the web application. For the WebCal web application, we used the subroutine `start_html()` (in Perl) in the file `webcal_shared.pm` which is invoked to print the start of a html page whenever the application generates and displays a page. We modified this subroutine and embedded the security policy file in the first `<script>` tag within the `<head>` tag. The success in applying our security mechanism to these real world applications gives us confidence in that it could be widely used to counter modern cross-site scripting attacks.

6.2 Overhead

We evaluated the overhead of our policy enforcement mechanism by measuring the overhead of a number of individual JavaScript weaving operations at runtime and page render time. All experiments were performed on a PC PENTIUM D 820 (2.80GHz) with 1.0GB memory.

6.2.1 Weaving overhead

We selected 10 operations to measure their weaving overhead. Table 1 shows the operations and their weaving slowdown ratio. We used JavaScript time functions to calculate the execution time of each operation test, which is a loop lasting over 500 milliseconds. Each operation test was performed locally from the local file system. The execution time of each operation test is averaged over 10 browser loads. For each operation, we weave the execution event with an assignment and proceed with the execution. The slowdown ratio was measured by comparing the average execution time of each operation test with the weaving process and that of original code.

Overall, the slowdown is quite small since our enforcement mechanism is so lightweight. The average slowdown for the 10 operation tests is 6.33 times. Compared to similar ratio overhead of BrowserShield [31], which was average 66.03 times, our enforcement method overhead is very light. The two greatest slowdowns in our operation tests are `concat`

[9]It depends on the structure of each web application

	operation	slowdown
1	`document.getElementsByTagName(..)`	4,89
2	`document.createElement(..)`	3.16
3	`eval()` with an assignment	6.09
4	`eval()` with if ... else	3.50
5	`toString()` object method	9.19
6	string split (`split()` method)	4.61
7	string replace (`replace()` method)	11.16
8	string concat (`concat()` method)	19.54
9	`document.cookie` read (property read)	0.69
10	`document.cookie` write (property write)	0.09

Table 1: Weaving slowdown ratio for operations in self-protecting JavaScript

and `replace` operation of the `String` object. Interestingly enough, the execution time of monitored property read and write operations is faster than that of the original one: reading from `document.cookie` is 45% and writing is 11.58 times faster than reading/writing from an unmodified property. This can be explained by the fact that reading/writing to properties is manipulated through a property cache in our library and does not involve expensive XPCOM calls to the DOM engine.

6.2.2 Render Overhead

The overhead is calculated as percentage difference of the load time of an original page and its modified version with policies embedded. We used a data set of the front pages of 22 sites retrieved from the Internet (18 of which are slowest pages from top 100 popular web sites on `alexa.com`, and the remaining 4 are websites frequently visited by the authors). Then, the 22 pages were copied and the security policies were inlined in each. We uploaded all the pages to our web server to perform the test. Each web page was loaded in the browser 20 times. We measured the total time taken by the browser to load each page using YSlow plugin[9] in Firefox version 3.0.1 on Windows XP SP2 platform. The total load time of 22 unmodified pages in 20 times was 1408.82 seconds, while the total time to load the modified pages was 1484.54 seconds, thus the average overhead is 5.37%.

7. DISCUSSION

In this section we discuss the main limitations of the approach.

Frame, iFrame, and refresh.

The approach we have described is page-specific. All scripts which are part of the page are subject to the policy. This includes, for example, scripts that are loaded via image tags, tables, body background or event handlers, e.g. `onClick()`, `onLoad()`, etc. However the policy does not apply to scripts loaded via `<frame>` or `<iframe>`. These force loading the documents in a *new* context, unreachable for our mechanism. For security reasons these pages are intended to be isolated from the parent page, which means they execute in a fresh context and are, thus, not subject to the policy of the parent. The same applies to the HTML *refresh* directive: this loads a fresh page, whereas our protection is limited to the present page only.

If we adopted a proxy-based approach to our policy injection then one could argue that such pages would also receive their own self-protection. However this has several limitations. If we have application-specific server-side protection then this approach is not applicable. A more important problem is if we want policies which span several pages. This implies, amongst other things, the need for a shared security-state.

Implementation-specific issues.

We touched upon two issues which relate to Mozilla-specific features, one positive and one negative. Firstly, our treatment of JavaScript *properties* of built-in objects is essentially the same as our treatment of methods, but relies on using Mozilla-specific getter and setter methods in order to monitor and modify the behaviour of property accesses. While we benefit from this feature, we suffer at the hands of another. As described in Section 3.5, the behaviour of Mozilla's implementation of the delete operator is a major problem. In the latter case we discussed the solution "just around the corner" in the form of the current ECMAScript 3.1 draft [3, 42]. This language revision proposes to give the programmer control over the delete operation. It turns out that the proposal for ECMAScript 3.1 also includes Mozilla-style user-definable getter and setter methods, thus making our approach to dealing with property access potentially more widely applicable.

Transparency.

A classic property expected of a security reference monitor is *transparency*. This is the property that systems which are already well-behaved are unaffected by the monitor.

As discussed at the end of Section 5, our approach is *not* transparent even when restricted to the implementation of security automata. In fact it is not transparent even when restricted to a trivial policy which does not attempt to do anything. The issue here is JavaScript's reflection capabilities, and we argue that the lack of transparency exhibited here is pathological.

The problem is that any JavaScript code can inspect the source of the page in which it is embedded. Thus it is possible for the script to detect the presence of the policy code. If it detects it, it can modify its behaviour accordingly. In particular we can have a script which acts like an attention seeking toddler: well-behaved when not being watched, but behaving badly when monitored. Our method would clearly not be transparent for such a script. The question is whether we care about transparency in such cases. We would argue that in practice this is not an issue.

Interestingly the BrowserShield approach [31] goes to considerable lengths to achieve some transparency. This is done, at some cost, by maintaining a runtime shadow of the original document. Every document read or write must use and maintain, respectively, the shadow copy. A similar approach may be possible in our setting by suitably modifying all methods which read and modify the document, although we have not attempted to construct such a policy. However, even BrowserShield's method is not transparent to a determined attacker. Since the method has high cost there are easy timing channels which can be used to signal the presence of the monitor. We believe that it would be more practical to settle for a weaker transparency goal which promises transparency for non-self-inspecting programs. A precise formalisation of this property remains to be made.

Perhaps a more important security implication of JavaScript's self-awareness is that we can have code which behaves badly only when it is *not* being watched. Trans-

parency is not the issue here. This means that a malicious script might survive undetected for longer, by attacking only clients unequipped to detect their actions. This, however, is not a transparency issue but a general issue for all self-protection approaches.

8. RELATED WORK

Web application security has recently received wide attention both in industry and in the research community. Most web browsers provide security protection for JavaScript such as sandboxing, same-origin policy and signed scripting [32, 5]. XPCNativeWrapper [6] is a security extension library that wraps up objects so that it can limit access to the properties and methods of the wrapped object. Unfortunately, these mechanisms only provide coarse-grained protections; attacks such as XSS, phishing, or resource abuse could defeat these protections.

8.1 Static Analysis

Few existing solutions use static analysis of the source code of server programs in order to detect violations of data integrity policies that could lead to potential cross-site scripting or other code injection attack [30, 4, 41, 35, 22, 43]. The basic idea is to declare all the input data of the program logically "tainted". The programmer is then forced to sanitise the data obtained from untrusted sources in order to produce an output depending on the these inputs. Some of the analyses are quite elaborate [43] and allow to precisely capture the flows of information even in such dynamic languages as PHP ([43]). However, as practice has shown [24], input sanitisation is hard to get right[10] and is, probably, the weakest point in the solution. The solution presented in [28] achieves the same goal as the above, but using dynamic code analysis. The fact that the analysis is being dynamic provokes a question on the soundness of the method ([33]). The path taken in [28] also requires modification of the language interpreter.

8.2 Monitoring

In contrast to static analysis approaches there are mechanisms that offer protection on the client side by monitoring the execution of the JavaScript programs. Here the type of the enforced policies could be different. One paper ([19]) enforces access control policies, controlling script access to DOM API methods and properties. Somewhat in the spirit of our work, the solution presented in [19] requires quite an invasive modification of the browser (Firefox). Another work [39] also relies on a modification of Firefox, but concerns data integrity policies. Although the monitoring mechanism does not raise immediate soundness concerns, it is, however, quite restrictive when it comes to the most interesting (dynamic) parts of JavaScript. It is also unclear whether data integrity policies enforced on client side can prevent any cross-site scripting attacks.

Tahoma web browsing framework uses virtualization and sandboxing techniques to contain the client part of the web application [15]. Although the emphasis of this work is on mitigating the vulnerabilities of the web browser there is a mechanism that allows web application developers to restrict the communication of the client side by specifying a list of allowed URLs. In addition, it restricts the web application access to local files. The approach, however, requires using special versions of the OS, the hypervisor and the browser.

The common feature of these three solutions is that they require the modification of the basic software on the client machine. Introducing new security mechanisms, for example, in the browser potentially allows more transparency than in our solution. But, from an immediate practical perspective such modifications are certainly time consuming, error prone and, most importantly, short lived: the rapidly changing codebase of the main branch of, e.g, Firefox (or any other open source project) is likely to make the changes obsolete rather soon. Maintaining the patches for the current versions of software would most likely require much effort. From a general perspective methods requiring browser modifications offer discretionary protection. In contrast, protection mechanisms not involving browser modifications do not require the browser user to be proactive.

In [21], the authors proposed a solution for preventing script injection called Browser-Enforced Embedded Policies (BEEP). In this solution the code developer plays an active role (as we envisage for our approach). The idea of this mechanism is that web pages contain embedded policies describing which scripts are allowed to run. When executing scripts in the web pages, the browser enforces the policies. Although the BEEP provides a lightweight security policy enforcement, web browsers are needed to *modify* in order to enforce policies at runtime. Moreover, the mechanism can enforce *only access control policies* that allow script methods in a whitelist to be executed and deny script methods execution not in the whitelist. Another weakness of this solution is that attackers can use script methods in the whitelist to launch attacks, for example, using the `XMLHttpRequest` (that is allowed by Web 2.0 applications [8], thus it should be in the whitelist) to send `cookie` to adversary's site. The paper shows the security hook function to enforce policies is tamper-proof but did not discuss the problem that attackers can overwrite the whitelist or add more methods in the whitelist to break the policy enforcement. Compared with BEEP, our solution could enforce more fine-grained security policies without browser modification and provides method to protect policy enforcement.

8.3 Code transformation

CoreScript [44] uses program instrumentation for JavaScript where untrusted JavaScript code goes through a rewriting process according to a security policy before executing them in the browser. CoreScript policies are specified in term of edit automata [25], which is essentially the same class covered by our policies. The rewriting process is deployed in an add-on proxy in the browser[11], thus it requires the modification of the browser. A disadvantage of this approach is that it only consider script code in `<script>` tags, consequently, other ways embedding script code e.g. using tags such as `<img src=...>` `<body background=...>` or event handlers such as `onClick()`, `onLoad()`... are escaped from the rewriting. Moreover, scripts dynamically generated at runtime the rewriting method in CoreScript are treated by string analysis, therefore, it might face the problem that attackers can evade the analysis by using script encoding. In

[10] One ends up either too restrictive or misses obscure attack vectors.

[11] In fact the main focus of the work is on the theoretical framework as described for a simple language; our description of their "system" is an extrapolation.

our method, we intercept execution events at runtime, thus our method can overcome these challenges.

The system called BrowserShield [31] which we have discussed in more detail in Section 1 is also based on code transformation. We consider this to be the closest related work since it is browser independent. BrowserShield is a general framework that rewrites web pages to enforce policies at runtime. As mentioned previously BrowserShield applies transformation dynamically. The main weaknesses here are script obfuscation evading BrowserShield's parser, and the run-time overhead of such invasive transformation, both in terms of the size of the library, and in terms of the runtime parsing and transformation of code. Advantages over our approach are the possibility to make the instrumentation mechanism transparent to the code being monitored (modulo other covert channels such as timing) and the fact that the code can monitor and modify arbitrary code patterns – something which might be useful to repair other forms of code vulnerabilities (e.g. (hyperthetical) buffer overruns in the runtime system triggered by specific coding patterns).

The approach of Erlingsson in [37] also relies on dynamic code transformation, but is distinct from the vast majority of other research and is, potentially, capable of enforcing a wide range of policies. However it requires features not currently present in any of modern browsers, i.e. so called mutation event transformers.

Another interesting approach to access control on the client side is taken in [27]. Instead of employing a reference monitor, the authors show that a certain subset of JavaScript is an *object-capability language* and represent access control rights as *capabilities*. The enforcement model is static analysis.

Other subsets of JavaSript, like FBJS ([18]) and ADsafe ([17]), are used as standard languages for untrusted scripts in Facebook and Yahoo applications respectively. FBJS enforces sandboxes the untrusted code by enforcing the usage of a separate namespace. ADsafe puts restrictions on some of the most dangerious features of JavaScript. Both languages are backwards-compatible with JavaScript and allow programs to run in the browser unchanged. Although these approached do not allow the developers to specify flexible policies for their code, they could be used in combination with our approach, providing more rigid foundation than unconstrained JavaScript.

9. CONCLUSIONS

We have described an approach to control and modify the behaviour of JavaScript by transforming the code to make it self protecting. Unlike previous mechanisms for achieving similar goals our method does not require browser modifications, and is non-invasive, avoiding the need for extensive run-time code transformation in order to handle dynamicly generated scripts or scripts loaded on the fly. The flexibility of self protecting code is that the self-protection can, on the one hand, be applied at the code source (server side) in order to embed application specific policies describing the intended behaviour of the code. Even if the code is compromised by a XSS attack then the developer's policy still applies. On the other hand the method can be used as a lightweight way to patch generic vulnerabilities by inserting the self-protection using a proxy or plugin.

Acknowledgements.

We are indebted to Jonas Magazinius and Sergio Maffeis who both provided valuable contributions in the form of attacks to the method presented. Thanks to Andrei Sabelfeld and the anonymous referees for their feedback and helpful comments.

This work was financially supported by the Swedish funding agencies Vinnova (The Swedish Governmental Agency for Innovation Systems), VR, and SSF, and by the European IST-2005-015905 MOBIUS project.

10. REFERENCES

[1] Aspect Oriented Extensions for jQuery. `http://code.google.com/p/jquery-aop/`.

[2] CNN: 'Phishing' scams reel in your identity. `http://www.cnn.com/2003/TECH/internet/07/21/phishing.scam`.

[3] ECMAScript 3.1 Language Specification. `http://wiki.ecmascript.org/doku.php?id=es3.1:es3.1_proposal_working_draft`. Working Draft as of 01 Dec 2008.

[4] IBM Rational Web application security software (former AppShield). `http://www-01.ibm.com/software/rational/offerings/websecurity`.

[5] Mozilla Developer Center: Core JavaScript 1.5 Reference. `http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference`.

[6] Mozilla Developer Center: XPCNativeWrapper. `http://developer.mozilla.org/en/docs/XPCNativeWrapper`.

[7] RSnake: XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html`.

[8] Web 2.0. `http://en.wikipedia.org/wiki/Web_2`.

[9] Yahoo Developer Network: YSlow for Firebug. `http://developer.yahoo.com/yslow/`.

[10] I. Aktug, M. Dam, and D. Gurov. Provably Correct Runtime Monitoring. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, pages 262–277, 2008.

[11] 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.

[12] M. Arciemowicz. phpBB 2.0.18 XSS and Full Path Disclosure. `http://securityreason.com/securityalert/269`.

[13] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM.

[14] S. Bubrouski. XSS in WebCal (v1.11-v3.04). `http://securityreason.com/securityalert/267`.

[15] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 350–364, Washington, DC, USA, 2006. IEEE Computer Society.

[16] D. Danchev. HSBC sites vulnerable to XSS flaws,

could aid phishing attacks. http://blogs.zdnet.com/security/?p=1365. Posted on June 29th, 2008.

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

[18] Facebook. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

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

[20] I. Jacobs. URIs, Addressability, and the use of HTTP GET and POST. http://www.w3.org/2001/tag/doc/whenToUseGet.html. Accessed May 16 2007.

[21] T. Jim, N. Swamy, and M. 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.

[22] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (short paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[24] E. Levy. Worst-case scenario. *IEEE Security and Privacy*, 4(5):71–73, 2006.

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

[26] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, pages 307–325, 2008.

[27] M. S. Miller, M. Samuel, B. Laurie, and I. A. M. Stay. Caja: Safe active content in sanitized JavaScript. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

[28] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, 2005.

[29] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. Technical Report 2008:24, Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden, December 2008. Project URL: http://www.cse.chalmers.se/~phung/projects/jss. ISSN:1652-926X.

[30] T. Pietraszek, C. V, and E. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[31] 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.

[32] J. Ruderman. Same origin policy for JavaScript. http://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.

[33] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[34] F. B. Schneider, J. G. Morrisett, and R. Harper. A Language-Based Approach to Security. In *LNCS 2000, Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101, London, UK, 2001. Springer-Verlag.

[35] A. Stevens. KaVaDo InterDo: A useful tool in the fight to keep your server secure. http://www.vnunet.com/pc-magazine/software/2133317/kavado-interdo. PC Magazine, 08 Jul 2002.

[36] Úlfar Erlingsson. *The Inline Reference Monitors Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Ithaca, New York, 2004.

[37] Úlfar Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[38] Úlfar Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Washington, DC, USA, 2000. IEEE Computer Society.

[39] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, and C. Kruegel. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS '07: Proceeding of the 14th Annual Network and Distributed System Security*, San Diego, CA, 2007. Internet Society.

[40] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, New York, NY, USA, 1993. ACM.

[41] J. Wells. Protect sensitive Web data with Teros-100 APS. http://articles.techrepublic.com.com/5100-10878_11-1060422.html.

[42] A. Wirfs-Brock. Proposed ECMAScript 3.1 Static Object Functions: Use Cases and Rationale. http://wiki.ecmascript.org/lib/exe/fetch.php?id=es3.1%3Aes3.1_proposal_working_draft&cache=cache&media=es3.1:rationale_for_es3_1_static_object_methodsaug26.pdf. Revised August 26, 2008.

[43] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2006. USENIX Association.

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