Flexible Access Control for JavaScript

Gregor Richards[1] Christian Hammer[2] Francesco Zappa Nardelli[3] Suresh Jagannathan[1] Jan Vitek[1]

[1] Purdue University [2] Saarland University [3] INRIA

Abstract

Providing security guarantees for systems built out of untrusted components requires the ability to define and enforce access control policies over untrusted code. In Web 2.0 applications, JavaScript code from different origins is often combined on a single page, leading to well-known vulnerabilities. We present a security infrastructure which allows users and content providers to specify access control policies over subsets of a JavaScript program by leveraging the concept of delimited histories with revocation. We implement our proposal in WebKit and evaluate it with three policies on 50 widely used websites with no changes to their JavaScript code and report performance overheads and violations.

1. Introduction

Many popular Web applications mix content from different sources, such as articles coming from a newspaper, a search bar provided by a search engine, advertisements served by a commercial partner, and included third-party libraries to enrich the user experience. The behavior of such a web site depends on *all* of its parts working, especially so if it is financed by ads. Yet, not all parts are equally trusted. Typically, the main content provider is held to a higher standard than the embedded third-party elements. A number of well publicized attacks have shown that ads and third-party components can introduce vulnerabilities in the overall application. Readers of the New York Times online version were subject to a scareware attack originating from code provided by a previously trustworthy ad agency.[1] Similarly, German newspapers were attacked by malware from a legitimate advertisement service which delegated some ads to second-tier agencies. Other attacks leverage the extension facilities of some popular web sites. Facebook, for example, encourages third party extensions to be delivered as JavaScript plugins. Taxonomies of these attacks are emerging [19]. Attacks such as *cross site scripting, cookie stealing, location hijacking, clickjacking, history sniffing* and *behavior tracking* are being catalogued, and the field is rich and varied.[2] Barth *et al.* even proposed a name for this threat model: the *Gadget Attacker* [2, 5].

What makes the JavaScript platform challenging is that applications that run in a single client-side browser are composed on the fly, their source code is assembled from different sources and run in the same environment with little isolation. Moreover JavaScript is very dynamic; text can be turned into code at any time and very few properties can be statically guaranteed [30]. Our study of real-world JavaScript behavior demonstrated that this dynamism is widely used [31]. Web browsers offer two lines of defense for end-users: The first is a sandbox that protects the operating system from JavaScript code. The second is known as the *same origin policy* (SOP); it segregates components into trust domains based on their origin (a combination of host name, port and protocol) and enforces access control restriction on elements of the web page with a different origin. However, the SOP is not uniformly applied to all resources (e.g. images may come from different origins, potentially leaking information in their URLs), and it is too coarse-grained. While the SOP prevents scripts in one frame from accessing content in another, many web sites choose not to use frames as this form of isolation is highly restrictive, and far from foolproof [5]. Thus much third-party code is simply included in the body of the web page [26].

The majority of attempts to strengthen the security of Web 2.0 applications rely on isolating trusted from untrusted code and limiting the dynamism of JavaScript, either through static analysis techniques which reject programs that do not meet certain static criteria or by defining a subset of the language that is easier to verify [12, 13, 23–25]. These approaches have been adopted by the industry as exemplified by Facebook JavaScript, Yahoo's AdSafe, or Google Caja. However, these techniques can be circumvented [34] and can

[1] `www.nytimes.com/2009/09/15/technology/internet/15adco.html`, `www.h-online.com/security/features/Tracking-down-malware-949079.html`.

[2] `www.webappsec.org/projects/threat`, `www.owasp.org/index.php/Category:Attack`.

OOPSLA '13, October 29–31, 2013, Indianapolis, IN, USA.

http://dx.doi.org/10.1145/2509136.2509542

miss attacks due to peculiarities and leniencies of different browsers' JavaScript parsers, and they are so restrictive that many valid legacy programs would be rejected.

This paper proposes a novel security infrastructure for dealing with the Gadget Attacker threat model. We extend JavaScript objects with *dynamic ownership* annotations and break up a web site's computation at ownership changes, that is to say when code belonging to a different owner is executed, into *delimited histories*. Subcomputations performed on behalf of untrusted code are executed under a special regime in which most operations are recorded into histories. Then, at the next ownership change, or at other well defined points, these histories are made available to user-configurable *security policies* which can, if the history violates some safety rule, issue a *revocation* request. Revocation undoes all the computational effects of the history, reverting the state of the heap to what it was before the computation. Delimiting histories is crucial for our technique to scale to real web sites. While JavaScript pages can generate millions of events, histories are typically short, and fit well within the computation model underlying Web 2.0 applications: once the history of actions of an untrusted code fragment is validated, the history can be discarded. Histories allow policies to reason about the impact of an operation within a scope by giving policies a view on the outcome of a sequence of computational steps. Consider storing a secret into an object's field. This could be safe if the modification was subsequently overwritten and replaced by the field's original value. Traditional access control policies would reject the first write, but policies in our framework can postpone the decision and observe if this is indeed a leak. While policies of interest could stretch all the way to dynamic information flow tracking, we focus on access control in this paper.

Targeting Web applications means that we must deal with the idiosyncrasies of modern web browsers and propose a security model that could be deployed without disrupting the ecosystem. Our main design constraint was thus backwards compatibility with the web. For this reason our design carefully avoids extending the syntax of JavaScript and does not require changes to code that is already well-behaved. Our proposal can be integrated into a web browser with no modifications of the implementation of existing web sites. We leverage the existing SOP policy and use it as a building block in our infrastructure. Our secondary goal was to demonstrate acceptable performance. While many of the overheads of a proof-of-concept implementation can be optimized away, massive slowdown would make adoption unlikely. We address this by an in-browser implementation and a careful selection of the properties being recorded. This paper makes the following contributions:

- *A novel security infrastructure:* Access control decisions for untrusted code are based on delimited histories. Revocation can restore the program to a consistent state. The enforceable security policies are a superset of [33] as revocation allows access decisions based on future events.
- *Support of existing JavaScript browser security mechanisms:* All JavaScript objects are owned by a principal. Ownership is integrated with the browser's same origin principle for backwards compatibility with Web 2.0 applications. Code owned by an untrusted principal is executed in a controlled environment, but the code has full access to the containing page. This ensures compatibility with existing code.
- *Browser integration:* Our system was implemented in the WebKit library. We instrument all means to create scripts in the browser at runtime, so if untrusted code creates another script we add its security principal to the new script as well. Additionally, we treat the `eval` function as untrusted and always monitor it.
- *Flexible policies:* Our security policies allow enforcement of semantic properties based on the notion of security principals attached to JavaScript objects, rather than mere syntactic properties like method or variable names that previous approaches generally rely on. Policies can be combined, allowing for both provider-specified security and user-defined security.
- *Empirical Evaluation:* We validated our approach on 50 real web sites and two representative policies. The results suggest that our model is a good fit for securing web ad content and third-party extensions, with less than 10% of sites' major functionality broken. Our policies have successfully prevented dangerous operations performed by third-party code. The observed performance overheads were between 11% and 106% in the interpreter.

Our work builds on and combines ideas from the literature. History-based access control [1] extends the stack inspection security model of Java to include a history of methods called. Inline reference monitors [33, 37] dynamically enforce a security policy by monitoring system execution with security automata. We build on their insight and record a wider selection of operations. The design of our policy language is informed by Polymer [6]. While the notion of revocation is inspired by research on transactional memory [16], we avoid the transactional memory terminology because delimited histories depart from transactions in that they do not guarantee isolation, they do not perform conflict detection, they do not re-execute aborted histories, do not support nesting of histories, and record different sets of operations.

2. JavaScript and Security

JavaScript is a prototype-based object-oriented language which is extremely dynamic. Objects can be (and, as shown in [31], are) modified in arbitrary ways after their creation. Moreover, text can be turned into executable code by the `eval` function (which is more frequently used than we would

like [30]). A JavaScript object is a set of properties, a mutable map from strings to values. A property that evaluates to a closure and is called using the context of its parent object plays the role of a method in Java. Each object has a prototype, which refers to another object. As a result, it is difficult to constrain the behavior of any given object, as either it or any of its prototypes could be modified at any time. A JavaScript program running in a browser executes in an event-driven fashion. The browser fires events in response to end-user interactions such as cursor movements and clicks, timer events, networks replies, and other predefined browser happenings. Each event may trigger the execution of a JavaScript function. When the function returns, the system is able to handle the next event. The timing and order of events depend on the particular browser, network latency, timer accuracy, and other environmental factors. The JavaScript code interacts with the browser, the network, and in a very limited way, persistent storage through native operations.

2.1 Threat Model

Web browsers have standardized on the same origin policy to isolate content from different providers. Figure 1 illustrates the situation where a single web page is built out of a mixture of trusted and untrusted components kept at bay by the browser's SOP. Web pages are served by a provider. Each party has its goals. The host provider's interests are to retain the user's trust and to maximize ad revenues. Users want access to the content while restricting, as much as possible, the behavior of ads and other untrusted elements. For the purposes of this paper, we focus on threats originating from third party scripts such as ads and widgets that are embedded in an otherwise trusted page. While there are plugins that block undesired content, this practice hurts web pages' funding, and it is also based on syntactic properties (a blacklist of resources not be loaded) instead of a semantic security policy.

A typical attack is easy to construct. Imagine a host which uses a third-party ad service. The ads are loaded by including a dynamically-created script into the host page. If the ad service is malicious or has been corrupted then it may do much more than simply display ads. For instance, in this scenario there is no built-in security mechanism in the browser which prevents the script from installing a handler for keypress events and silently intercepting and logging everything the user types, such as login credentials for the host site. Although the same origin policy is intended to prevent the script from then communicating this data to an untrusted host, there are ways to work around it, such as encoding the logged data into a source URL for an image tag. Section 5 reports on some real-world attacks. For concreteness, consider Figure 2. The host, mysite.com, uses a third-party ad-supported login service, happylogins.com, with ads from evilads.com. The login service is loaded in an iframe in hopes of isolating it from the host. mysite.com also stores private information of the user, in a variable secret. The ad is able to leak the secret by taking advantage of subtle flaws:

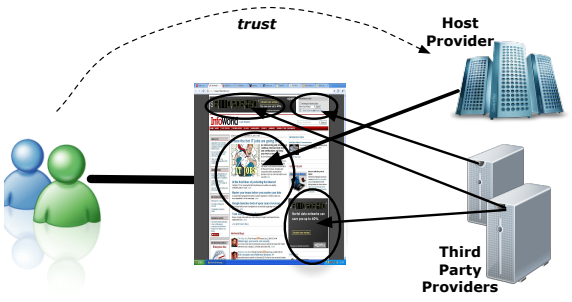

Figure 1: Web applications are made up of components of multiple origins. End-users typically trust the main provider, but do not have a relationship with ad providers. The browser's same origin policy attempts to isolate the different components of a web page.

```
<script>
var secret = "supersecret";
document.addEventListener("message",
  function(e) {
    var resp = eval(e.data);
    // handle the response
  }, false);
</script>
Please log in:
<iframe src="http://happylogins.com/login">
</iframe>
```

(a) http://mysite.com/

```
<script src="http://evilads.com/ad.js">
</script>
<script>
var okMsg = "({loginOK: true})";
function login(u) {
  if (loginOK(u))
    window.parent.postMessage(okMsg, "*");
}
</script>
<input type="text" id="name">
<button onclick="login(this.value);">
Log In</button>
```

(b) http://happylogins.com/login

```
window.addEventListener("load", function() {
  okMsg = "new Image().src = " +
    "'http://evilads.com/evil?p='" +
    "+secret;";
}, false);
```

(c) http://evilads.com/ad.js

Figure 2: okMsg is a vulnerability; it allows any component loaded on the login service's page to run code on the host page.

- The host trusts the login service to provide JSON.
- The login service trusts the ad not to corrupt its heap.
- The ad is not loaded in a frame, and as such is not subject to the SOP.

The host code adds an event listener for "message" events, which are triggered by the postMessage command. This is the only means of communication between frames of different origins, and messages can only be strings. In this case, the message is expected to be an object serialized in JSON. A widely used mechanism for deserializing JSON is eval, which the host assumes is safe because the message can be verified to have originated from the login service's origin. The login function checks whether the login information is correct, and sends back a canned message that will set the property loginOK to true. With these two pieces of code in isolation, the communication is secure. The ad code, however, is able to create an event that fires when the frame fully loads, which replaces the canned JSON login OK message with a string of JavaScript code which will, when executed, send the secret variable in the host to evilads.com. Although the secret could not be sent via an asynchronous web request, images are allowed to come from any source, and so the request is allowed.

2.2 Out-of-scope Threats

We do not consider covert channels, as we believe that users would not adopt a technology that would be overly restrictive. We do not consider flaws in the browser; attacks that trigger buffer overflows or heap-spraying attacks are not covered by our infrastructure. They are orthogonal to the ideas we are exploring. Attacks based on other browser technologies, the browser's layout engine, plugins, social engineering etc. are not in the scope of this work. We assume that the host page is trustworthy and do not prevent potentially malicious behavior stemming from the host, including most forms of cross site scripting. Other research has targeted these issues.

3. Delimited Histories with Revocation

The security infrastructure we propose applies security policies to portions of a program's execution. We start with a high-level description; we will discuss later how this integrates in a JavaScript engine.

DEFINITION 1. *The execution state of web application consists of the state of a JavaScript engine P and an environment E. A step of execution is captured by a transition relation $P|E \xrightarrow{\alpha} P'|E'$ where α is a label.*

A label denotes either an action α^E initiated by the environment or an action α^P performed by the JavaScript engine.

α^E	EVT n	External event of type n
	REP v	Return value v from a native call
α^P	APP f v	Call function f with arguments v
	RET v	Return value v from a call
	GET v w p v1	Member v.p found in w holds v1
	SET v p v1 v2	Update v.p from v1 to v2
	NEW f v	Create an object with constructor f
	INV f v	Invoke native operation f

The latter set comprises function calls (including calls to eval) and returns, properties reads and writes, object allocation, and calls to native functions. The former set comprises external events (for some set of events ranged by n) and returns from calls to native functions.

DEFINITION 2 (Trace). *A trace $T = \alpha_1 :: \cdots :: \alpha_n$ corresponds to an execution $P|E \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} P'|E'$. We write $P|E \vdash T, P'|E'$ when execution of a configuration $P|E$ yields trace T and configuration $P'|E'$.*

Security policies are applied at *decision points* and *suspension points*. Decision points represent the end of an untrusted subcomputation, suspension points denote calls to native functions (e.g. XMLHttpRequest.send). Native functions potentially have irreversible side effects: suspension points enable policies to catch these early, preventing information leaks before they become irrevocable.

Programs are run with a pair of policies $(\mathcal{P}_S, \mathcal{P}_D)$ such that $\mathcal{P}_S$ is applied at suspension points and $\mathcal{P}_D$ is applied at decision points. Policy decisions are based on the state of the computation P and a subset of trace T which we call a *delimited history*, denoted $[\![T]\!]$. We represent the outcome of applying a policy as a label α^S in the set {OK, REV}.

DEFINITION 3 (Policy application). *Given a policy $\mathcal{P} = (\mathcal{P}_S, \mathcal{P}_D)$ and a computation $P|E \vdash T, P'|E'$ with a delimited history $[\![T]\!]$, applying the policy can yield either the trace $P|E \vdash T::$ OK, $P' \mid E'$ if $\mathcal{P}(P', [\![T]\!]) =$ OK, or the trace $P|E \vdash T::$ REV, $P \mid E'$ if $\mathcal{P}(P', [\![T]\!]) =$ REV.*

If the policy returns REV, we say that the delimited history $[\![T]\!]$ has been revoked. This has the side effect of reverting the JavaScript state of the computation P' to its original state P, the state before the call was made. None of the changes to the memory and internal state of the JavaScript program are retained. On the other hand, the environment E' is not rolled back (as there is no practical way to undo network traffic). As mentioned above, a policy may use suspension points to prevent external effects from happening at the cost of having to make access control decisions early.

We tie the start of a delimited history and the associated decision point to the principal on whose behalf a script is run. The SOP policy defines a notion of principal based on three components of a web page – the application layer protocol, the domain name, and the TCP port of the URL the file originated from. Since any call to eval may, potentially,

take as an argument an untrusted string and be subject to distinguished policies, we extend the definition of ownership to include the string passed to eval; for normal code execution this field is empty and our ownership coincides with the definition of principal of SOP.

DEFINITION 4 (Ownership). *Every JavaScript object is associated with an ownership record o, which is a quadruple* `(protocol, domain, port, eval)`.

We call *host page* the web page obtained from the URL in the location bar of the browser. In JavaScript, functions and scopes are objects, thus they are naturally tagged with an ownership record. The default ownership record is that of the host page. It would be reasonable to start a delimited history whenever code originating from any page other than the host page is executed. One could simply start recording when the browser encounters a <script> tag from a different origin and place a decision point at the matching end tag. Unfortunately many ads install callbacks (via setTimeout or an event handler) or install functions in the global object which can be called by the host page's code. Calls to eval must also be taken into account. This justifies the following definition.

DEFINITION 5 (Recording). *For a page with ownership $o = (p, d, r, \epsilon)$, recording of a history $[\![T]\!]$ starts at a call APP f v or NEW f v if no recording is in progress and any of the following holds: (i) the owner o' of function f is not o, (ii) the previous label was a EVT script o' and $o' \neq o$, or (iii) the function f is eval, and $o' = (p, d, r, v)$. The history is said to be owned by o'. A decision point occurs at the matching return unless the history was revoked at a suspension. Every object created during recording is tagged with the owner of the history, o'.*

A delimited history thus starts at either a script tag, the invocation of function object (either directly or through an event handler), or a call to eval. The owner of the function is used to tag all objects created while the history is active. Ownership is invariant during the course of execution, so calling into another function with a different owner does not affect the tag associated by objects that the function creates.

The history $[\![T]\!]$ contains the following: $last([\![T]\!])$ is the last label in the trace which, in case of a suspension point, is of the form INV f v. $rd([\![T]\!])$ is the sequence of all property accesses GET v w p v1 to all locations (v,p). $fwr([\![T]\!])$ is the sequence of writes SET v p v1 v'2 derived from the *first* write SET v p v1 v2 and *last* write SET v p v'1 v'2 to any (v,p) pair. Only the original value and most recently written value are necessary for updates; in practice, the last value may simply be read from the live heap and need not be recorded. Observe that GET and SET are used for both property accesses on objects and variable access within scopes. These notions are conflated in JavaScript: the global scope is also an object, window, there thus is little distinction between the two.

As is customary with JavaScript, there are subtleties. When reading a property, such as x.foo, lookup starts with the object referenced by x, but may potentially traverse the prototype chain. In a label GET v w p v1, v is the target of the property lookup, and w is the object where the property was found. The object NONE is used when the property was not found. For writes SET v p v1 v2, JavaScript semantics does *not* specify prototype traversal[3], i.e. the only object that may change due to a SET is v. There are three cases to consider: (a) the property p is found in v and its old value v1 is updated to v2, (b) the property did not exist in v, the property will be added to the object (this case is denoted by v1 = NONE), (c) the property was deleted from v by the JavaScript delete operation (v2 = NONE). With this information, revocation of a history entails going through fwr to revert properties to their original values and, in the process, create and delete properties as appropriate.

Suspension points must also be added to property accesses which may be rerouted to a native getter or setter method. For example, setting the src property of an image tag in the DOM will trigger the download of the image at that URL. For pragmatic reasons the implementation maintains a whitelist of functions that are deemed safe and do not introduce a suspension point. Any other native function is a suspension point.

In both the SOP and our delimited history model, each frame is considered an independent entity, and so code running within it is owned by the frame's origin, and not the original page's origin.

3.1 Example

To make this design more clear, reconsider Figure 2. We will step through this example assuming an empty policy that always yields OK. When mysite.com loads, its script tag (lines 2-7) will be executed. Since the source comes from the same origin as the site itself, no recording of history needs to be started. The owner of the callback function it produces will therefore also have the same owner. The iframe tag will cause happylogins to load, but in an isolated environment protected by the browsers SOP. As such, the host owner for that code is happylogins, not mysite. Its first script tag refers to evilads, so history recording starts.

The file ad.js has only one statement, and it will result in a GET window addEventListener f1, followed by NEW Function f2, the second argument to addEventListener. Because the function is created while within a history owned by evilads, its owner will be evilads. Then a APP f1 event is produced with the provided arguments which installs f2 as a callback function. The execution continues with the second script tag on happylogins. As its owner is the host of the iframe, it is not recorded, and the function login is owned by the happylogins.

When happylogins has finished loading, it will fire a load event. In this case, evilads has registered a load event listener,

[3] Setter functions are represented as a GET followed by an APP.

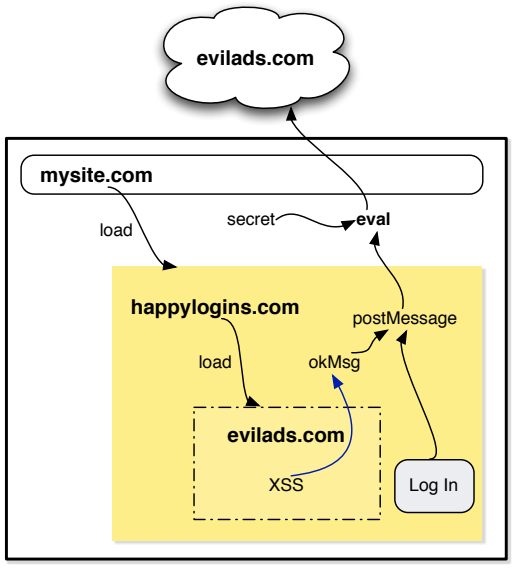

Figure 3: Data flow through the web site detailed in Figure 2.

f2. Because f2 is owned by evilads, its execution will be recorded in a history. Its execution consists of a single event (line 2), SET window okMsg v v' with the previous string v' (set by happylogins) and its new string v, containing a cross-site scripting attack.

When a user clicks the "Log In" button, the login function set by happylogins will be executed. Since its owner is the host origin of that frame, it will not be recorded. Assuming the login is OK, the message it sends, okMsg, will be the one set by evilads. Until this point, the primary host, mysite, has been idle. Having been sent a message from the login frame, its message handler (lines 4-7) will be executed, with the message as an argument (in e.data). Because the message handler's owner is mysite, it will not be recorded. It will retrieve the attack string that evilads produced, and use it as an argument for eval. eval'd strings are always recorded. In this case, the code yields the following behavior:

- NEW Image ()
- SET v1 src undefined with the new URL, where v1 is the image produced previously.
- As setting src calls a native function and performs I/O, the execution suspends for policy checking.

Since an empty policy was applied in this example, the secret will in fact be leaked. This attack could have been stopped in several points, which will be discussed in Section 4.

4. Security Policies

We now turn to the policies that can be expressed in our infrastructure. Policies are written in C++, the implementation language of WebKit. A library of policies is linked dynamically to the instrumented browser at startup. Using C++ gives policy developers the opportunity to write low-level code that will have predictable performance. A clear demarcation line between the policy language and JavaScript makes it somewhat easier to ensure that policies will not be tempered with. We considered expressing policies in JavaScript but were discouraged by concerns of efficiency and the challenges of enforcing isolation. To improve readability, we present policies in idealized pseudo-code.

4.1 Policy API

Our infrastructure includes a number of simple data structures used by security policies. Pseudo-code for these can be found in Figure 4. The class Owner encodes the ownership information required by our variant of the SOP. The enumeration Suggestion contains three values: IGNORE to denote cases where a history is irrelevant to a policy, OK if a history abides by a policy, and REVOKE for policy violation which should be revoked. The abstract class Op has subclasses for all operations recorded in a history, these include GetOp, SetOp, CallOp, and DownloadOp among others. A delimited history is an instance of the History class which holds a suggestion (by default IGNORE) and has methods to return the last operation in the history, an array of read operations, an array of writes, and method originalValue(op) which, for a Get or Set operation, will return the value of the field at the start of the history.

A security policy is represented at runtime by an instance of a subclass of Policy created at page load time and re-

```
1  class Owner {
2    var port : number, domain : string,
3        protocol : string, evalstr : string;
4  }
5  enum Suggestion {
6    IGNORE, OK, REVOKE
7  }
8  class History {
9    var suggestion : Suggestion;
10   fun last() : Op;
11   fun reads() : Op[];
12   fun writes() : Op[];
13   fun ops() : Op[];
14   fun originalValue(op : Op) : any;
15 }
16 class Policy {
17   var owner : string;
18   Policy(args : string[]) {}
19   fun setOwner(o : string) : void {owner=o;}
20   fun querySuspend : History (history : History, op : Op) {
21     return queryEnd(history);
22   }
23   fun queryEnd(history : History) : History { return history;}
24   fun cleanup(history : History) : void {}
25 }
```

Figure 4: Pseudo-code for the Policy APIs. Class Policy must be extended to define security policies. Class History is the delimited history recorded while executing an untrusted script.

claimed when the page is destroyed. Every policy has a reference to the immutable Owner record describing the origin of the code being executed. Policy classes have methods: querySuspend() is called at each suspension point and is passed a history and the reason for suspension. It must reply by returning a history with a suggestion. queryEnd() is called when the end of a history is reached. cleanup() is called with the final history and suggestion to request that the policy clean up any state variables it may have used, unless the suggestion was IGNORE. cleanup() is merely a simplification; querySuspend() could clean up whenever it returned REJECT, and queryEnd() could clean up universally. Policies are composed by building policy combinator objects with sub-policies embedded within them. We will now proceed with some examples of policies that have been implemented for our experiments.

4.1.1 Controlling changes to the Global Object

In JavaScript, the global object is a dumping ground for variables defined at the top level and serves as a communication channel between scripts. We show a policy that lets untrusted scripts extend the global object, but not change existing values or objects of a different owner. While this does not prevent breaking the host page, it does prevent subverting the host's functionality. The ad could install a new function tricking the host into executing it. However, the function would be run in a delimited history due to its ownership. The policy in Figure 5 extends the base policy class by defining a queryEnd() method which iterates over the operations in the write set of the history (line 3). For each of the write operations it tests if the owner of the object differs from the owner of the history (line 4). If the write updates an existing property in the global object (lines 5-6), then the suggestion is set to REVOKE. The querySuspend function is not distinct from queryEnd, as the writes need to be checked in the same way.

4.1.2 Hygienic Policy

One might want to adjust the add-only policy above to allow writes as long the original value is restored before the decision point. The policy in Figure 6 checks if all properties of objects that are not owned by the history's owner are restored to their original values by the end of the history. This policy is an example that semantically looks into the future to assess whether an event needs to be prevented.

```
1  class AddOnly : Policy {
2    fun queryEnd(history) {
3      foreach op in history.writes()
4        if (differentOwner(op) &&
5              op.target.isGlobalObject() &&
6              history.originalValue(op) != NONE))
7            history.suggestion = REVOKE;
8      return history;
9    }
10 }
```

Figure 5: Restricting global object property updates.

```
1  class SameValue : Policy {
2    fun querySuspend(history, op) {}
3    fun queryEnd(history) {
4      foreach op in history.writes()
5        if (differentOwner(op) &&
6              op.value() != history.originalValue(op))
7              history.suggestion = REVOKE;
8      return history;
9    }
10 }
```

Figure 6: Ensuring that values of objects belonging to other owners are returned to their original state.

4.1.3 Script blocking

One possible use of our infrastructure would be to define a trivial policy that blocks either all scripts or those from a selected group. The policy of Figure 7 is created with a user supplied argument that lists disallowed sites, any query will check the history's owner against the list.

4.1.4 Send After Read Restriction

The lifetime of a policy is tied to that of the page, encompassing possibly multiple invocations of untrusted operations. The policy can retain some security state across multiple suspension points and across multiple histories. This is needed to prevent leakage of confidential information via HTTP requests. Pseudocode of this policy appears in Figure 8. It disallows events which transmit data over the Internet after read events have been performed on data owned by another principal. It also prevents transmission after enabling event listeners, as knowing when events fire is a leak of potentially-private information. The policy maintains some internal security state. The variable pos tracks how much of the history has already been checked, which is updated at every query. The default behavior of querySuspend() is to call queryEnd(). The pos variable is reset by cleanup(), which is only called if one of the queries returned something other than IGNORE. The hasread variable is a bit of security state that is retained across different histories. This prevents a two stage attack where one script (running with one history) reads information and the second script (running in another history) performs a HTTP request leaking that value. By retaining state across histories we can ensure that the policy will remember that some script has read protected data and prevent the send.

4.2 Composing and Selecting Policies

Policies can be composed by combinators, objects that invoke query and cleanup methods of sub-policies. As combinators are policies themselves, they have access to histories and can, for example, filter histories to hide some

```
class Blocker : Policy {
  var blacklist;
  Blocker(args){super(args); blacklist=args;}
  fun queryEnd(history) {
    if (blacklist.contains(owner))
      history.suggestion = REVOKE;
    return history;
  }
}
```

Figure 7: Blocking scripts from specified servers.

```
class SendAfterRead : Policy {
  var hasread = false;
  var violation = false
  var pos = 0;
  fun queryEnd(history) {
   foreach op in truncate(history.ops(), pos) {
      if (op.isGetOp()) {
        hasread |= differentOwner(op);
      } else if (op.isCallOp()) {
        if (op.isNative() &&
            op.name.is("addEventListener"))
          hasread = true;
      } else if (op.isDownloadOp()) {
        violation |= hasread;
      }
    }
    pos = history.ops().length();
    if (violation) history.suggestion=REVOKE;
    else history.suggestion=OK;
    return history;
  }
  fun cleanup(history){violation=false; pos=0;}}
```

Figure 8: Preventing send after read.

events deemed safe. Combinators can also override policies to, for instance, take into account user preferences over site-specified policies. Figure 9 illustrates a binary combinator which takes the conjunction of two policies. Policies are specified by name and created by makePolicy.

4.3 Policy Combinators: White Listing

Many sites use secondary servers for storing static content, including JavaScript. For instance, youtube.com's JavaScript is hosted on ytimg.com. For these sites, the same origin policy is too stringent, as it would treat the secondary server as untrusted. The Whitelist class is a combinator which modifies the owner of subpolicies so as to give the same owner to all scripts coming from secondary servers. Figure 10 shows a simplified version of the whitelist combinator. The setOwner method is overridden to tag an object with the main hostname instead of the name of a secondary server. The list of secondary servers is passed to the constructor in the parameter list. findPrimary returns the name of the primary server if the current host is in the list of secondary servers.

```
class Conjunction : Policy {
  var p1, p2;
  var s1, s2;
  Conjunction(name1, args1, name2, args2) {
    p1=makePolicy(name1,args1);
    p2=makePolicy(name2,args2);
  }
  fun setOwner(o) {
    p1.setOwner(o);
    p2.setOwner(o);
  }
  fun join(s1,s2) {
    if (s1 < s2) return s2;
    else return s1;
  }
  fun queryEnd(history) {
    h1 = p1.queryEnd(history);
    s1 = join(s1,h1.suggestion);
    h2 = p2.queryEnd(history);
    s2 = join(s2,h2.suggestion);
    h2.suggestion = join(s1,s2);
    return h2;
  }
  fun cleanup(history){
     if (s1 != IGNORE) p1.cleanup(history);
     if (s2 != IGNORE) p2.cleanup(history);
  }
}
```

Figure 9: A simple Policy combinator.

```
class Whitelist : Policy {
  var p, list;
  Whitelist(name, args, list) {
    p=makePolicy(name,args);
    this.list=list;}
  fun setOwner(o) {
    var o2=findPrimary(o)
    super.setOwner(o2);
    p.setOwner(o2);}
  fun queryEnd(h) {return p.queryEnd(h);}
  fun querySuspend(h) { return p.querySuspend(h);}
  fun cleanup(h){ p.cleanup(h);}
}
```

Figure 10: A simple Whitelist combinator.

4.4 Breadth of Security Policies

A survey of policies found in the research literature appeared in the ConScript paper [25]. We discuss whether these policies fit into our framework. Their first policy disallows all scripts. The related policy that disallows inline scripts (i.e. scripts embedded in attributes of html tags) does not fit our SOP-based model, as inline scripts would have the same owner as the host code and thus are not monitored. We could of course change our notion of principal but this would likely cause more mismatches with legacy code; we believe these first two policies are over-restrictive. Most of the policies collected in [25] are syntactic. For example, all policies that restrict access to potentially hazardous methods (maybe in-

volving some form of black- or whitelisting) can be trivially implemented as a policy in our framework. At suspension points the policy needs to check whether the method to be called matches a given signature, whether the arguments have certain properties like a given type, and whether the signature and/or arguments are valid according some black- or whitelist. For methods which do not trigger suspension points, blacklisting can be implemented by checking the history at a decision point. A strength of our approach is that while we can handle syntactic properties (modulo dead code), we also can reason about the side-effects that untrusted scripts may induce and their potential for putting the trusted host code's security at risk. Therefore, the policies presented in this section have a different quality than those summarized in [25]. As the next section will elaborate, many real attacks can be prevented with a handful of simple policies.

4.5 Rectifying Errors

Ideally, policies would never incorrectly revoke the behavior of non-malicious code, but in practice, policies may be overly restrictive, preventing valid code from proceeding. Consider as a possible example ads on a social networking site such as Facebook. Facebook stores information on the user, and much of it may be accessible in JavaScript objects. With no policy in place, an ad may look into that object, and it may be considered non-malicious for the ad to use non-specific information such as the user's postal code and sex, which the ad could then use to target the user more specifically. If Facebook added a policy which prevented access to all objects with user information, this ad would cease to function.

The ad cannot detect programmatically whether such a policy is in place, and even if it could, it could not anticipate all possible policies, so it is not possible to generally write third-party code to account for such changes in policy.

Because our technique is based on recording histories, it is possible to determine quite specifically what the offending code has done wrong when its behavior is revoked[4]. This could be as easily detected by the ad agency as by Facebook, or indeed any Facebook user. With the specific offense known, the ad could be modified to avoid violating the policy, or the policy could be modified to support the ad. For instance, in the example given, Facebook could modify their policy to disallow all access to user information except for sex and postal code, or even more generally to, for instance, allow ads to read user information so long as they don't then send any data, or allow ads to read no more than a particular number of sensitive fields. We imagine that the precise changes to be made would be negotiated between the ad agency and the site in question.

[4] Being an academic prototype, the current version of the software does not present this information adequately, but it is available.

5. Empirical Evaluation

We evaluate our proposed security infrastructure along several dimensions. Based on an implementation in a production browser, we start by considering real attacks, then we run sample policies on real web sites and lastly we measure performance and scalability.

5.1 Implementation

We implemented our system in the open source rendering engine WebKit, which is used in the Safari browser. To that end, we modified 29 files: 20 in the JavaScriptCore package, which implements the core JavaScript interpreter and is independent of the actual browser integration, and 9 in WebCore, which accounts for the browser-specific JavaScript like the DOM. We added 588 lines of code to these 29 files. Apart from that, new classes added 4697 LOC. The biggest change was in the Interpreter class, which we instrumented to intercept all relevant events. In particular, we instrumented all opcodes where object properties are accessed (read, store and delete), or objects are created. We also intercept the beginning and ending of script execution, calls, returns, and abrupt termination due to exceptions. The call stack is the basis for determining the policy's decision points. An event is forwarded to a filter class that decides whether history needs to recorded based on the call stack and the owners of involved functions. In case we are recording a delimited history, events are relayed to a bookkeeping class that handles all policy-independent tasks. Newly created objects are tagged with the owner of the current history (whereas the owner of objects created outside a history defaults to the owner of the host page). The owner of the current history and the associated global object are recorded for that purpose. Furthermore, the bookkeeping class records operations including the read and write sets and maintains a list of policies to be checked. For each call to a function, the bookkeeping class determines whether this function has a native implementation, and if so, whether that function is contained in a whitelist of side-effect free functions. If not, a suspension point is triggered and we iterate over all installed polices to query if the call is allowed or needs to be prevented. In the latter case, a flag is returned to the interpreter to not call the native function and abort execution of the script. For getter and setter methods invoked as a result of property access we adopt a different strategy[5]. When calling into a setter or getter function results in reaching native code where a side-effect like network access or database storage is about to take place, we generate a synthetic download or storage event and pass that to the filter before triggering the side-effect. If we are currently in a delimited history these events are suspension points that will be passed along to the policy, which then decides whether the side-effect is permitted or not. In

[5] In WebKit, a native getter or setter may be called for optimization purposes, but in the majority of the cases a standard interpreter function is called, which makes it impossible to distinguish the external code.

the latter case the native code will basically throw an exception instead of executing the side-effect that is subsequently caught at the invocation point of the setter or getter.

Revocation is a rather tricky business as there are two separate call stacks maintained by the interpreter, and both need to be popped to the point where history recording started. First, the interpreter maintains a stack modeling the JavaScript call stack which needs to be reverted to the level when the history started, before restoring the program counter and resuming execution from there. However, some native calls like eval alter the underlying C++ call stack in addition to the JavaScript call stack. Therefore, when a call frame contains WebKit's HostCallFrameFlag, which signifies the invocation of the interpreter loop function, we need to return from that C++ function and resume unrolling of the JavaScript call stack of the interpreter loop of the previous C++ call frame. The unwinding of call stacks does not trigger execution of finally blocks.

Once the system determines that a script is going to terminate, be it normally or due to an exception or violation of a security policy, it triggers the decision point check in the bookkeeping class. This iterates over all policies checking whether a violation has happened in the recorded history. If a policy signals a violation, all writes in the write set are rolled-back according to JavaScript semantics, any thrown exception is caught and the undefined value is returned. While the completion value will be ignored for histories started by a script tag, eval expressions might use the return value for further computations, which must be prevented for aborted scripts.

5.2 Attack Vectors

5.2.1 Samy worm

The Samy worm[6] is a Cross Site Scripting/Cross-site request forgery (CSRF) worm developed to propagate through the MySpace social networking site. While MySpace filtered the HTML that users add to their sites, the worm exploited holes in the filtering process to inject code into the profile of each person that viewed an infected page. The main hole in the filtering was that some browsers allow JavaScript within CSS tags. MySpace tried to prevent this but failed due the rather lenient JavaScript parsers found in many browsers. For instance, splitting a keyword (such as javascript) across line boundaries as shown in the following code snippet was enough to defeat MySpace's filters:

```
<div id=mycode style="background:url('java
script:eval(document.all.mycode.expr)')"...>
```

The injection mechanism relies on eval since both single and double quotes are already taken. Writing any meaningful JavaScript code inside the style tag is difficult. But it suffices to add the text of the attack to a property of the div tag, called expr, and access this property in the eval expression. The attack itself consists of reading several pieces of information from e.g. the document's location and the document itself and using them as parameters to subsequent AJAX requests that inject the malicious code into the profile of a viewer. The key steps involved: (1) eval('document.body.inne' + 'rHTML') to access the content of the website in a way that circumvented the filter mechanism; (2) redirect from profile.myspace.com to www.myspace.com, as the SOP would block AJAX calls; (3) html.indexOf('Mytoken') to access the random hash from a pre-POST page for the subsequent AJAX request; and (4) sending an AJAX request.

While our approach does not primarily target code injection attacks, it can prevent this CSRF attack. We are sandboxing JavaScript code originating from an eval expression and any other means to dynamically create code. Thus the SendAfterRead policy prevents reading private data on the page (like Mytoken) and subsequently sending the AJAX request. Blocking the AJAX request prohibits CSRF attacks as they rely on the requests being sent out with the user's credentials. We validated this claim experimentally by embedding the Samy worm in a test page and running it with the SendAfterRead policy. When running in an uninstrumented Safari browser, the page containing the Samy worm sends out all AJAX requests necessary to infect the viewer of the profile. With our system, this AJAX request is prevented as the script had read the document location. This read is considered private data, so the policy prevents subsequent Internet requests.

5.2.2 Clickjacking

A *clickjacking* attack lures the user into clicking on a link of an invisible element. The attacker needs to have access to the current page to create the invisible element, which is usually an iframe, and move that element over a legitimate link on the page. The attack may have a different origin than the host page, and, as the user authorizes the operation with a click, it will be executed with all the user's credentials. For example, if the user is logged into Facebook, the attacker can add an application to the user's Facebook account. The click will send any authorization data (e.g. a cookie) along with the request. Even worse, when automatic password fill-in is turned on in the browser, the user might be tricked into logging in, and a subsequent click will execute the malicious operation without the user even noticing an attack has occurred.

Our approach does not prevent mouse clicks on a link, but as the attack requires addition of an event handler, that script will be monitored. Consider the following as an example:

```
function updatebox(evt) {
 var mouseX=evt.clientX;
 var mouseY=evt.clientY;
 var f = document.getElementById('open');
 f.style.left=mouseX−10;
 f.style.top=mouseY−25;
}
```

[6] http://namb.la/popular/tech.html

If third-party code wanted to inject a clickjacking attack into a host, a policy that prevents updates to objects from a different owner, such as the SameValue policy would prevent this attack, as the callback function will have the third-party owner, such that the updates performed to f.style are revoked by the policy.

5.2.3 History Sniffing

It took more than 10 years[7] until an attack known as *history sniffing* was addressed by browser manufacturers. History sniffing infers the browser's history through the style of links (visited links are displayed in a different color). While newer browsers are immune to the basic form of history sniffing (including the CSS-only variant), this attack is being used in practice [19] and it has been shown that it can be abused very effectively, detecting as many as 30,000 links per second [18]. The SendAfterRead policy would prevent most sniffing attacks. We will return to this with an in depth example in Section 5.3

5.2.4 Key-logging

If third party scripts run unmonitored they may even install key- or mouse-loggers threatening user security. For example, a keylogger might attempt to intercept login credentials or other sensitive data provided to the host page. Figure 11 shows an example keylogger that sends out the log periodically via its reportLog function. The SendAfterRead policy prevents these kinds of attacks, as installing an event handler is considered by the policy to be another reason to reject send events. This is because knowing when events fire is a leak of potentially-private information.

```
1 window.addEventListener("keypress",
2   function(event) {
3     log.push(event.which);
4     if (log.length >= 1024) reportLog(log);},
5   false);
```

Figure 11: Attack by keylogger in untrusted code.

5.2.5 Storage

Recently, a new attack against user privacy has been proposed based on replicating user tracking data in a set of storage mechanisms other than traditional cookies. In particular, HTML5 proposes three more storage containers apart from cookies: Session and local storage and database storage. Session storage is accessible to any page from the same site opened in that window, while local storage spans multiple windows and lasts beyond the current session. Both of these mechanisms provide a key/value storage interface to JavaScript. Even more powerful is the database storage mechanism that provides an SQL interface persisting multiple sessions. All these storage mechanisms represent side-effects that potentially threaten security, as data stored in one session can be accessed and e.g. sent out over the Internet at a later time, which would invalidate policies like NoReadSend. An extension of the SendAfterRead policy in Figure 8 where StorageEvents trigger a violation after reading (in analogy to DownloadEvents) prevents third-party code from changing storage of the host page, as well as reading that storage and sending out information later on.

[7] https://bugzilla.mozilla.org/show_bug.cgi?id=57351

5.3 Case Study: Sniffles

We present a real attack found on zaycev.net, a file and news sharing site in Russia (#1390 on the Alexa list). The host page loads advertisements from a third-party ad server, but included along with the ads is a history-sniffing attack which determines how many of a list of sites the user has visited in the past. It works by setting CSS styles for visited links, then checking if they are active on links that the attacker is interested in. The code reads data it should not have access to (the style of links) and sends the information to a foreign web server. Figure 12 is a reduced version of the attack seen on the site.

```
1  function ucv(c, d) {
2      var a = document.createElement("a");
3      a.href = c; d.appendChild(d);
4      return (a.style.color == "#ff0000");
5  }
6  var d = document.createElement("div"), seen = [];
7  addStyle(d, "a:visited { color: #ff0000 }");
8  if (ucv("qwe.ru", d)) seen.push("qwe.ru");
9  //... same for other servers
10 seen = seen.join(",");
11 // send seen to the ad server
```

Figure 12: Excerpt of a history-sniffing attack

Although the SendAfterRead policy could prevent a history-sniffing attack, we devised an extension to the add-only policy which furthermore restricts what the foreign ad is allowed to read, and what functions it is allowed to call. The functions it is allowed to call are those which add to the page, instead of reading data from the page (e.g. the data that must be read to sniff history), which is consistent with the add-only policy.

The ad is allowed to read from the global state and even from the DOM, but not from the CSS style or from text nodes in the DOM. For certain targets it is only allowed to call certain functions: for document: write, getElementById, createElement, getElementsByTagName; for window: setTimeout, parseInt, open, encodeURIComponent, escape; for other DOM elements: setAttribute, appendChild; all functions are ok for Date; none for all other targets. These functions are sufficient for a conventionally-written ad script to add its advertisement to the page, and in fact the history-sniffing ad in question conforms to this policy if the history-sniffing attack itself is removed. Because none of these functions read information from a page, only adding information, they are

```
class Sniffles : Policy {
  fun queryEnd(history) {
    foreach op in history.writes()
      if (differentOwner(op) &&
          (!op.target.isGlobalObject() ||
           history.originalValue(op) != NONE)))
        history.suggestion = REVOKE;
    foreach op in history.reads()
      if (differentOwner(op) &&
          (!op.target.isGlobalObject()) &&
          (!op.target.isHTMLDOMObject()))
        history.suggestion = REVOKE;
    foreach op in history.calls()
      if (differentOwner(op) &&
          (!isWhitelistedFunction(op.target, op.func)))
        history.suggestion = REVOKE;
    return history;}}
```

Figure 13: Restricting reads and calls.

a reasonably privacy-preserving subset of the DOM API. Using document.write is fine in our system as scripts that are thus created will be monitored by the security policy, as well. Further restrictions, like allowing access to a DOM node only if that was intended by the host [22] are possible. This is important for targeted ads that are allowed to process public parts of the page but not the sensitive information.

5.4 Real-Site Behavior

We applied security policies to unmodified web sites to evaluate how restrictive the policies are for legacy code, code that was not specifically designed to abide by a particular policy. Figure 14 shows the results of applying three policies to the top 50 web sites (as displayed on `alexa.com` on 20-Mar-2013). We report the count of untrusted scripts executed, and thus the number of histories created by our infrastructure and of decision points where policies are evaluated. The number ranges between 0 and 417. One website, wikipedia, did not load any untrusted code. In detail, we recorded how many times the same behavior occurred (Count), the owner of the page or iframe (Host Origin), the owner the code (Script Origin), the reason why a history is being recorded (Cause) which can be one of **e**val, **s**ource, and **f**unction with different owner, the number of suspension points (Suspend), the total number of revocations (Revoke), the size of the read and write set in the history (R/W). For space reason we do not list the entire ownership information but rather part of the URL. Note that for a given site the host for frames may differ from that site's URL and that host would still be a trusted host, as frames are protected by the SOP in the browser.

It is striking to observe the size of the scripts. None are small, ranging from 192KB to 2.8MB of JavaScript code executed during our short runs. On average, 84 delimited histories were recorded per site, but that number and the behavior of the actual histories varies considerably between these sites.

Inspection of the code revealed that Google does not load any third-party code, but twenty-one eval statements were executed. None was suspended or revoked. Manual inspection suggests that Google uses eval to deserialize JSON objects. Facebook uses some alternate domains to store static content, which we had to whitelist. While it ran no third-party code, delimited histories were created due to eval, one of which was revoked. Most of this code do not explicitly read or write properties; three of the evals create and define objects via JSON and pass them back to the host. Since none of the objects owned by the host are changed in that process, all of these are allowed, as well. Youtube needs whitelisting of a static content domain s.ytimg.com. With that in place, the page looks and feels like the original even though some advertisement services and third-party functionality are rejected. For example, we found that a script from 2mdn.net tries to install a global function isValid, however a function with that name had already been defined by the host code in the global scope. Therefore, allowing the untrusted code to install the function may put the functionality of the host in jeopardy; in the worst case a malicious script may subvert the security of the host code. Our policy rejects that script and reverts its side-effects. Yahoo structures its services into several subdomains and maintains a number of hosts for static content, which we whitelisted. We found frames displaying ads from different hosts. Some scripts were revoked due to updates to a field owned by the host. In a realistic setting the frame content provider would specify the hosts to be whitelisted so that their scripts would work as expected. Yahoo also loads scripts from Facebook. These scripts change the href property of a link not owned by the third-party script, which means that its side-effect are revoked at the next suspension point.

We categorized subjectively the behavior of the site with security enabled and report on the results in Figure 15. We first applied, as a sanity check, the Empty policy which

Policy	Empty	SendAfterRead	AddOnly
Functional	50	22	12
No auto-complete	0	8	15
Ads blocked	0	7	8
Partial	0	9	10
Broken	0	4	5

Figure 15: Categorization of 50 benchmarks. **Functional:** The site worked in its entirety. **No auto-complete:** Form auto-completion functionality was broken, but otherwise the site was fully functional. This is a common failure case due to how auto-completion is frequently implemented. **Ads blocked:** The site worked, but its ads did not. Although acceptable to clients, this is not acceptable to servers, since ads are a revenue source. **Partial:** The important or vital features of the site worked, but minor or secondary features other than auto-completion or ads did not work. **Broken:** The site is mostly nonfunctional.

Count	Host / Script Origin	Suspend Cause	R/W	Revoke	
	google		(16 files, 272KB)		
1	google / google	s	0	0	2/0
21	google / google	e	1	0	5172/572
1	google / ssl.gstatic	s	7	0	2745/293
11	google / clients1.google	s	1	0	14476/1708
55	google / clients1.google	f	0	0	14928/1477
1	google / clients1.google	f	1	0	8/2
2	google / ssl.gstatic	s	1	0	2/0
	facebook		(48 files, 2.8MB)		
33	facebook / facebook	e	1	0	15101/1778
30	facebook / facebook	f	0	0	11858/1336
2	facebook / facebook	s	0	0	10/2
2	facebook / facebook	s	1	1	20/4
	youtube		(16 files, 1.1MB)		
5	youtube / youtube	e	1	0	22/4
17	ad-g.doubleclick / youtube	s	0	0	7574/945
1	ad-g.doubleclick / youtube	s	15	0	0/0
1	ad-g.doubleclick / youtube	s	2	0	0/0
1	ad-g.doubleclick / youtube	s	1	1	2745/293
1	ad-g.doubleclick / youtube	s	3	1	10/2
1	ad-g.doubleclick / s0.2mdn	s	17	0	10/2
1	ad-g.doubleclick / youtube	s	4	1	10/2
1	ad-g.doubleclick / s0.2mdn	s	19	0	10/2
3	ad-g.doubleclick / s0.2mdn	s	0	0	2352/262
4	youtube / s0.2mdn	s	3	0	38/8
1	youtube / s0.2mdn	s	18	0	10/2
2	youtube / s0.2mdn	s	2	0	2326/260
1	youtube / s0.2mdn	s	4	0	10/2
7	youtube / s0.2mdn	s	0	0	2418/271
6	youtube / s0.2mdn	s	1	0	2354/268
1	youtube / s0.2mdn	s	5	1	2313/258
	qq		(44 files, 1.1MB)		
4	qq / qq	s	0	0	26/6
1	qq / pingjs.qq	s	0	0	0/0
1	qq / pingjs.qq	f	4	1	0/0
1	qq / adsrich.qq	s	8	0	0/0
1	qq / mat1.gtimg	s	2	0	0/0
2	qq / mat1.gtimg	f	1	1	2745/293
3	qq / mat1.gtimg	f	0	0	2818/408
1	qq / mat1.gtimg	f	5	1	0/0
1	qq / adsrich.qq	s	1	1	10/2
27	qq /	s	0	0	9561/1079
1	qq /	s	2	0	2331/259
3	qq / adsrich.qq	s	0	0	30/6
1	qq /	s	1	1	10/2
1	qq / qq	s	1	1	2313/258
1	qq / qq	s	0	0	10/2
1	soso / soso.qstatic	s	0	0	10/2
1	soso / soso	s	0	0	10/2
1	soso / adsrich.qq	s	0	0	2308/258
	yahoo		(23 files, 928KB)		
2	yahoo / yahoo	s	1	1	2/0
1	yahoo / ucs.query.yahoo	s	0	0	0/0
1	yahoo / sugg.us.search.yahoo	s	2	1	0/0

Count	Host / Script Origin	Suspend Cause	R/W	Revoke	
	wikipedia		(14 files, 800KB)		
	live		(24 files, 1.6MB)		
1	login.live / login.live	s	2	1	2/0
	baidu		(19 files, 192KB)		
2	baidu / s1.bdstatic	s	1	1	2/0
2	baidu / s1.bdstatic	s	0	1	0/0
7	baidu / baidu	e	1	0	5563/701
2	baidu / s1.bdstatic	s	0	0	20/4
	amazon		(34 files, 616KB)		
1	amazon / ad.doubleclick	s	1	1	2/0
7	amazon / amazon	s	1	0	4965/511
1	amazon / amazon	e	1	0	0/0
3	amazon / ad.doubleclick	s	0	0	20/7
52	s3l3lkinz3f56t.cloudfront / amazon	s	0	0	17949/2265
5	s3l3lkinz3f56t.cloudfront / amazon	s	1	1	38/8
132	s3l3lkinz3f56t.cloudfront / amazon	s	1	0	64837/7210
1	s3l3lkinz3f56t.cloudfront / amazon	s	5	1	10/2
9	s3l3lkinz3f56t.cloudfront / amazon	s	3	0	2388/274
1	s3l3lkinz3f56t.cloudfront / amazon	s	17	0	8/2
7	s3l3lkinz3f56t.cloudfront / amazon	s	2	0	2281/261
7	s3l3lkinz3f56t.cloudfront / amazon	s	3	1	575/261
1	amazon / amazon	f	1	0	0/1
1	amazon / amazon	f	0	0	0/1
1	amazon / amazon	f	6	0	0/1
1	amazon / amazon	f	20	0	3/2
1	amazon / amazon	f	5	0	841/74
1	amazon / amazon	f	10	1	29/1
59	view.atdmt / amazon	s	0	0	9820/7661
1	view.atdmt / b.voicefive	s	4	1	179/25
5	view.atdmt / amazon	s	1	1	0/472
2	view.atdmt / amazon	s	3	0	3054/298
93	view.atdmt / amazon	s	1	0	21661/6189
5	view.atdmt / amazon	s	2	0	745/324
3	view.atdmt / amazon	s	3	1	0/9
1	view.atdmt / amazon	s	0	0	3/2
1	view.atdmt / amazon	s	7	0	20/0
13	view.atdmt / amazon	s	6	0	7375/2184
2	view.atdmt / b.voicefive	s	0	0	487/95
	taobao		(48 files, 804KB)		
1	taobao / a.tbcdn.cn	s	2	0	2/0
12	taobao / a.tbcdn.cn	f	2	1	7954/972
1	taobao / z.alimama	s	1	0	0/0
1	taobao / a.tbcdn.cn	f	2	0	0/0
4	taobao / a.tbcdn.cn	f	0	0	10/2
4	taobao / a.tbcdn.cn	s	2	1	2348/264
80	taobao / a.tbcdn.cn	s	0	0	32609/3524
4	taobao / p.tanx	s	0	0	4677/520
4	taobao / p.tanx	s	1	1	40/8
1	taobao /	s	1	1	8/2
1	taobao / taobao	s	2	1	0/0
24	s.taobao / a.tbcdn.cn	s	0	0	7193/817
3	s.taobao / a.tbcdn.cn	f	0	0	2352/262
1	s.taobao / a.tbcdn.cn	s	1	1	10/2
1	s.taobao / a.tbcdn.cn	s	2	1	10/2

Figure 14: Dynamic characteristics of untrusted code for each of the top 10 site.

places no constraints on the code. Not surprisingly, no degradation of behavior could be observed. In our tests, we observed one very common failure case, the elimination of auto-completion features from form fields. In most cases, this was due to auto-completion being loaded either through eval or from a separate domain, and either whitelisting or a more finessed policy could have retained that behavior. For the remainder of this description, we consider that particular failure case as inconsequential. The SendAfterRead policy performed admirably, with 60% of web pages fully functional or with only auto-completion failing, and 14% of web pages functioning but with ads blocked, terminating the execution of the ad without affecting the site. In 18% of the pages the site broadly worked with minor features missing. The AddOnly policy is slightly more restrictive. Only 54% of sites were fully functional or only had auto-completion failures, with 20% having other missing features and 16% have some ads blocked. This shows that even with generic policies and unmodified code, our infrastructure is able to provide an acceptable user experience in 70% of the top 50 sites. In a realistic deployment one can expect customized policies and modification in the JavaScript code to yield significantly smaller false positive rates. For all runs we used a policy combinator to add the Whitelist policy to treat secondary servers as trusted.

Figure 16 is a poster child for global namespace pollution: it uses a variable i as a counter, but since the code executes in global scope, the variable is global. We reject the script as variable i was already defined by the host code.

```
1  var links = document.getElementsByTagName('a');
2  for (var i = 0; i < links.length; i++) {
3    links[i].onclick = function() {
4      if (document.images) {
5        var href = unescape(this.href).split('/article.aspx?');
6        var pic =new Image(1, 1);
7        pic.src ='http://www.af.com/Af.PartnerSite/'
8            + 'Tracker.aspx?event=ararequest&purl='
9            + this.href + '&' + href[href.length−1];
10 } }; }
```

Figure 16: ARAtracker.js at flickr.com.

5.5 Performance

To evaluate overheads, we arranged for our browser to load three web sites and to fire a deterministic sequence of events on each of these. We measured the execution time, both with and without our instrumentation. The sites were cached in a proxy to avoid inconsistency between runs. Our system is currently limited to WebKit's interpreter, so the JIT was not used. Five runs of each site for each mode were performed. The measurements were performed with the empty policy, as it has all of the instrumentation overhead needed to record policies but is semantically identical to running uninstrumented. The machine had a 3GHz six-core AMD Phenom 1075T processor and 16GB 667MHz DDR3 RAM. Our instrumentation is based on WebKit revision 92569, and the same revision was used for the uninstrumented runs. The three measured sites span the spectrum of history usage: MSNBC (www.msnbc.msn.com) runs nearly all of its code through eval, and so 98.41% of the traceable events were run in a history, YouTube (www.youtube.com) substantially less so (only 0.62%), and Google Maps (maps.google.com) has no histories at all.

Figure 17 shows the results. MSNBC which is a worst case scenario with most of the execution being recorded, has an overhead slightly of 115%. The execution records 17 histories accounting for 424 suspensions, 117,328 read events and 22,924 write events. In most sites we expect the amount of controlled code to be much less than the amount of host code as Figure 14 suggests. YouTube and GMaps are more typical with 13% to 15% overheads. Our YouTube benchmark ran 22 transactions accounting for 28 suspensions, 705 read events and 773 write events, and our Google Maps benchmark ran 48 transactions accounting for 3 suspensions, 33 read events and 122 write events.

	Instrumented		**Uninstrumented**		**Over-**
Site	Avg.	Std. dev.	Avg.	Std. dev.	**head**
MSNBC	275.8	7.29	128.2	3.77	115.1%
YouTube	155.2	1.30	137.8	1.64	12.6%
GMaps	224.2	4.32	195.4	1.14	14.7%

Figure 17: Runtimes (ms) with and without instrumentation.

Although the interpreter overhead running the empty policy is sufficient to gauge the overheads of our system, in Figure 18 we additionally provide measurements using the JIT (with no instrumentation, as we have not implemented instrumentation on the JIT) and with our two standard policies, AddOnly and SendAfterRead. Due to an early revocation on MSNBC in both policies, the runtime is actually considerably less with the realistic policies than the empty one; on the other sites, the runtime is not greatly affected by the choice of policy.

	JIT		**AddOnly**		**SendAfterRead**	
Site	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.
MSNBC	27.0	4.24	47.4	0.55	50.0	1.22
YouTube	15.6	0.54	156.4	1.14	154.8	0.84
GMaps	35.2	1.30	199.4	3.65	215.4	1.67

Figure 18: Runtimes (ms) with JIT, AddOnly, SendAfterRead.

6. Related Work

Improving the security of JavaScript programs on modern web browsers has attracted much interest. Many problems come from the pervasive sharing of data and code in JavaScript. While a browser could be viewed as an operating

system for web applications, unlike a traditional OS it is quite common to execute components with little or no isolation. This leads to, amongst others, Confused Deputy attacks [4], where a trusted program unknowingly exercises its authority to perform an action on the behest of an adversary. Many authors have proposed to draw stronger boundaries between components [12, 17, 29]. Library-based approaches tried to limit the interface between components [10]. Intrusion detection techniques were proposed to detecting misbehavior [14, 15, 35]. Considerable effort was invested in restricting the behavior of JavaScript programs. The highly dynamic nature of the language steered research towards a combination of filtering and rewriting [23]. Static analysis can filter programs before execution, while rewriting is used to inline reference monitors [25, 27, 28, 36]. BrowserShield [28] underlines the difficulty of detecting malicious scripts. Differences between JavaScript parsers led to false positives or missed attacks. Carefully crafted subsets of JavaScript [23, 24] allow the separation of programs into statically verifiable components and others that must be checked at run-time. This was refined into the staged information flow verifier of [8] which reduced the false positive rate down to 33%. Defining two subsets of JavaScript, a trivially statically analyzable one and a run-time checked subset, reduced the false positive rate to 22% [13]. As detecting malicious scripts remains tricky, some authors have advocated extending browsers. Security policies can, for instance, by embedded in web pages [20] with a 15% slowdown for a native implementation, and 10x slowdowns for a purely JavaScript implementation. However, it was shown that neither white- or blacklisting approaches are impervious to attacks [3]. Off-loading the execution of untrusted code to another JavaScript engine is another alternative to impose isolation. This has been show to have 69% overhead [22] with the drawback of preventing many of valid interactions present in legacy code.

Dhawan *et al.* [11] look at the use of Software Transctional Memory for similar purposes as our delimited histories. In [11], the authors offered an implementation in the Mozilla Firefox JavaScript engine. While the two proposals share the idea of using histories for access control decisions, we will argue for our proposal. Dhawan's work does not extend objects with ownership, and thus it can not leverage changes of ownership to determine decision points. Instead a new keyword is added to the language and web application programers must change their code to add transaction boundaries. Security, in their approach, crucially depends on programmers adding transactions at all points where untrusted code can run. This is done by encapsulating untrusted code in `transaction {...}`, the ellipsis stands for the code to run transactionally. Then the user must write code to run and check the results of the transaction. This means that to adopt the approach on legacy websites would require refactoring them. Also, their approach may capture trusted code in a transaction which would lead to a policy being applied to code that actually is allowed to perform sensitive operations. Their approach also requires conflict detection for the heap. Performance overhead of their approach is between 1.6x and 6.5x on tiny applications and up to 26x on microbenchmarks. No results are reported on real web sites. This makes the systems hard to compare, and since both proposal are implemented in JavaScript interpreters (and not JITs), perhaps comparing performance is premature. Dhawan's work support transactions on DOM objects, in our work we view these as external operations. The advantage of being able to undo DOM operation is that security policies need not worry about suspension points at DOM operations.

Other uses of similar ideas include: Birgisson et al. [7] base enforcement of authorization policies in concurrent programs on ideas from transaction memory to eliminate race conditions related to security checks. Rudys and Wallach [32] proposed transactional rollback as a way to implement safe termination of misbehaving codelets in Java. Speculative execution [21] takes a similar approach (speculation and rollback) but is built on top of a binary rewriting tool and reports slowdowns of up to $3000\times$.

De Groef *et al.* propose FlowFox, a version of Firefox that enforce security based on information flow control [9]. Their approach is based on secure multi-execution: the idea that the program is executed once for each security level with values that do not belong to that security level stubbed out. We considered dynamic information flow tracking but eventually discarded the idea because of the fundamental limitations of information flow tracking. Basically, information flow tries to detect causal dependencies between actions executed at different security levels. Consider the following example of two instructions in a sequence:

```
1 H.foo.bar = 1; // secret
2 L.fum = 1; // public
```

The first line assigns 1 to field of a secret variable, the second line stores 1 in the field of a public variable. One would expect that there is no information leak in this trivial program. Yet, if `H.foo` is `undefined` then an exception will be thrown and the second line will not be executed. A sound information flow policy would have to reject this program. Information flow control is still a research topic and it is possible that better solutions will be found.

Our policies are closely related to inline reference monitors (IRMs) [33, 37]. IRMs oversee system execution with security automata. IRMs typically apply a security policy $\mathcal{P}$ before the execution of each event and terminate the program if the policy is violated. Decisions cannot depend on future events. In contrast, revocation lets us base access control decisions on events that are yet to happen: having seen only a prefix T of the execution, whose last event α might violate security, we do not need to decide at that point whether the prefix T is benign or malign. Instead, we speculatively execute to the next decision point. Only then do we need to

check the policy, and revocation ensures that no side-effects of α are visible if a policy violation is detected. Decision and suspension points give well defined point in the computation where access control decision are made.

7. Conclusions

This paper presented a security mechanism for monitoring untrusted code in JavaScript based on delimited histories with revocation. When security policies can reflect upon sequences of operations performed by a computation, it is possible to support semantic properties rather than the more limited syntactic checks of traditional rewriting or wrapping approaches. Most prominently, it is possible to write policies to detect side-effects that jeopardize confidentiality or integrity, like sending out private data to untrusted servers or updating sensitive data structures. By extending the same origin policy and tagging objects as well as code with owners we obtain a robust notion of principal which can leverage to add our new security mechanism in a non-intrusive manner. Whenever control is transferred across an ownership boundary, security policies are activated. For that reason, trust boundaries become implicit and we can apply policies to unmodified JavaScript code and have significant success with legacy code. Our evaluation in the WebKit JavaScript engine demonstrates its effectiveness in preventing realistic attack vectors like internet worms, as well as applicability and scalability to realistic web sites exemplified by the 50 most popular pages.

We see promising direction for future work. Performance can be improved by integrating the monitoring mechanism with a trace-based just-in-time compiler and specializing the code generated to the policy. A better treatment of `eval` will greatly reduce the size of histories, especially in the case of pathological web sites that run entirely in `eval`. A second strand of work will investigate a mixture of static and dynamic checking to target information flow policies. In particular, we are interested in seeing if we can get a handle of quantitative flows by reflecting on histories. Lastly, we intend to investigate high-level declarative languages for specifying polices to allow for policy-carrying web pages.

Acknowledgments

This work was supported by in part by Google Research Award "HAJS: High-Assurance JavaScript" and by NSF Grant "CT-ER: Controlled Declassification with Software Transactional Memory".

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Network and Distributed System Security Symp. (NDSS)*, 2003.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF)*, 2010.

[3] E. Athanasopoulos, V. Pappas, and E. P. Markatos. Code-injection attacks in browsers supporting policies. In *W2SP 2009: WEB 2.0 Security and Privacy*, 2009.

[4] A. Barth, C. Jackson, and W. Li. Attacks on javascript mashup communication. In *W2SP 2009: WEB 2.0 Security and Privacy*, 2009.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52(6), 2009.

[6] L. Bauer, J. Ligatti, and D. Walker. Composing expressive runtime security policies. *ACM Trans. Softw. Eng. Methodol.*, 18:9:1–9:43, 2009.

[7] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *Conference on Computer and communications security (CCS)*, 2008.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Conference on Programming language design and implementation (PLDI)*, 2009.

[9] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Computer and Communications Security (CCS)*, 2012.

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

[11] M. Dhawan, C.-c. Shan, and V. Ganapathy. Enhancing JavaScript with transactions. In *ECOOP–-Object-Oriented Programming*, 2012.

[12] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *Workshop on Social Network Systems (SocialNets)*, 2008.

[13] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, 2009.

[14] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Conference on World wide web (WWW)*, 2009.

[15] O. Hallaraker and G. Vigna. Detecting malicious JavaScript Code in Mozilla. In *Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer architecture (ISCA)*, 1993.

[17] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: operating system abstractions for client mashups. In *Workshop on Hot topics in Operating Systems (HOTOS)*, 2007.

[18] A. Janc and L. Olejnik. Feasibility and real-world implications of web browser history detection. In *Proceedings of the 2010 Workshop on Web 2.0 Security and Privacy*, 2010.

[19] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Conference on Computer and communications security (CSS*, 2010.

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

[21] M. E. Locasto, A. Stavrou, G. F. Cretu, and A. D. Keromytis. From STEM to SEAD: Speculative execution for automated defense. In *USENIX Annual Technical Conference*, 2007.

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

[23] S. Maffeis, J. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Computer Security (ESORICS)*, 2009.

[24] S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *Symposium on Computer Security Foundations (CSF)*, 2009.

[25] L. A. Meyerovich and B. Livshits. ConScript: specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Symposium on Security and Privacy (S&P)*, 2010.

[26] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Computer and Communications Security (CCS)*, 2012.

[27] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *International Symposium on Information, Computer, and Communications Security (ASIACCS)*, 2009.

[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] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *European Conference on Computer Systems (EUROSYS)*, 2009.

[30] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do – a large-scale study of the use of eval in JavaScript applications. In *ECOOP–Object-oriented Programming*, 2011.

[31] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[32] A. Rudys and D. S. Wallach. Transactional rollback for language-based systems. In *Conference on Dependable Systems and Networks (DSN)*, 2002.

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

[34] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *Symposium on Security and Privacy (S&P)*, 2011.

[35] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Conference on Computer and Communications Security (CCS)*, 2009.

[36] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Symposium on Principles of programming languages (POPL)*, 2007.

[37] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004.

A. Availability

The prototype implementation described in this paper has been reviewed by the OOPSLA Artifact Evaluation Committee and found to "meet expectations". The software is released in open source and is available at:
`http://dumbo.cs.purdue.edu/webkit-ifc-5f893ccea10c.tar.bz2`

B. Implementation of SendAfterRead

We report below the complete C++ implementation of the SendAfterRead policy; the correspondence between the pseudo-code of Figure 8 and the actual implementation is obvious. We point out that while the pseudo-code relies on the standard implementation of querySuspend which includes a call to queryEnd, the C++ implementation provides optimized code for querySuspend, which does not traverse the whole delimited history but only checks the last event. It is also worth observing that reads to certain classes, in this case, HTMLDocument, Navigator and Location, are not considered as potentially harmful: Checking for members of document and navigator is a common technique to determine what features the browser or its JavaScript engine offers, and reading the current web page's URL is generally considered acceptable for analytics or targeting purposes. Note that these checks are shallow, so, for example, allowing reads to document does *not* allow arbitrary reads within document.body or other sensitive portions of the DOM.

```
1  /*
2   * SendAfterReadPolicy.cpp
3   */
4  #include "config.h"
5  #include "SendAfterReadPolicy.h"
6  #include "Threading.h"
7  #include "InstrEvent.h"
8  #include "AccessSet.h"
9  #include "EventGenerator.h"
10 #include "JSObject.h"
11 #include "JSFunction.h"
12
13 namespace JSC {
14   SendAfterReadPolicy::SendAfterReadPolicy() {
15     hasread = false;
16   }
17
18   SendAfterReadPolicy::~SendAfterReadPolicy() {}
19
20   bool SendAfterReadPolicy::queryEnd(History& history) {
21     if (history.ops().isEmpty()) return false;
22     Vector<Event*>::const_iterator end = history.ops().end();
23     history.addSuggestion(OK);
24     bool violation = false;
```

```
25      ExecState* exec = history.execState();
26      for (Vector<Event*>::const_iterator it = history.ops().begin();
27           it != end; ++it) {
28        Event* ev = *it;
29        if (ev->type() == Event::Read) {
30          violation |= readEvent(static_cast<ReadEvent*>(ev));
31        } else if (ev->type() == Event::Call) {
32          violation |= callEvent(static_cast<CallEvent*>(ev), exec);
33        } else if (ev->type() == Event::Download) {
34          violation |= downloadEvent(static_cast<DownloadEvent*>(ev));
35        }
36      }
37      if (!violation) return false;
38      history.addSuggestion(REVOKE); // violation
39      return true;
40    }
41
42    bool SendAfterReadPolicy::querySuspend(Event *ev, History& history) {
43      history.addSuggestion(OK);
44      bool violation = queryEnd(history);
45      if (ev->type() == Event::Read) {
46        violation |= readEvent(static_cast<ReadEvent*>(ev));
47      } else if (ev->type() == Event::Call) {
48        violation |= callEvent(static_cast<CallEvent*>(ev),
49                               history.execState());
50      } else if (ev->type() == Event::Download) {
51        violation |= downloadEvent(static_cast<DownloadEvent*>(ev));
52      }
53      if (!violation) return false;
54      history.addSuggestion(REVOKE); // violation
55      return true;
56    }
57
58    bool SendAfterReadPolicy::readEvent(ReadEvent* event) {
59      if (!event->getObject().isCell()) return false;
60      Owner owner = event->getObject().asCell()->getOwner();
61      bool read = false;
62      if (owner == Owner()) {
63        if (event->getObject().isObject()) {
64          JSObject* obj = asObject(event->getObject());
65          read = obj->className() == "HTMLDocument" ||
66            obj->className() == "Location" ||
67            obj->className() == "Navigator";
68        }
69      } else {
70        read = m_owner != owner;
71      }
72      if (read) hasread = true; //got field of a different owner
73      return false;
74    }
75
76    bool SendAfterReadPolicy::callEvent(CallEvent* event, ExecState* exec) {
77      if (! event->isNative()) return false;
78      InternalFunction *iff = asInternalFunction(event->getFunction());
79      if (iff->name(exec) == "addEventListener")
80          hasread = true; // callback has access to DOM object
81      return false;
82    }
83
84    bool SendAfterReadPolicy::downloadEvent(DownloadEvent* event) {
85      UNUSED_PARAM(event);
86      return hasread;
87    }
88  }
```