# A User's and Programmer's View of the New JavaScript Security Model


Vinod Anupam  David M. Kristol  Alain Mayer

*Bell Laboratories, Lucent Technologies*
*600 Mountain Avenue*
*Murray Hill, NJ 07974*
*{anupam,dmk,alain}@bell-labs.com*



## Abstract

*In this paper we introduce a new security model for JavaScript in Mozilla, as well as its programming interface. We present important concepts via examples from electronic commerce applications. We also describe our experience of implementing the model in the publicly available Mozilla source code. This model is likely to be integrated into Navigator 5.0, which, as of this writing, is scheduled to be released in late fall, 1999.*


## 1 Introduction

Web browser scripting languages are lightweight, yet powerful procedural languages with rudimentary object-oriented capabilities. Their source code is typically embedded in an HTML page and executed by an interpreter in the browser. As a form of executable content, they add interactivity and automation to browsers. This means that a Web page need no longer be static HTML, but can include transportable programs that interact with the user, control the browser, and dynamically create HTML content. Examples of such languages are Netscape's JavaScript, and Microsoft's VBScript (see [F97] and [L97], respectively).

At the same time, scripting also adds to the power of an adversarial entity. A user might visit a dubious site (`crook.com`) and (unknowingly) download scripts. Indeed, in the summer of 1997 we reported in [CERT97, AM98] an attack against both JavaScript and VBScript that allows a hostile entity to plant a Trojan Horse script in a user's browser. This script subsequently reports back all Web activity – URLs visited, private data supplied by the user in a Web form, *e.g.*, credit card numbers, social security numbers, company passwords, *etc.* Such an attack works even when the user employs encryption (*e.g.*, SSL) or when a user is behind a firewall, because the data is captured from the browser inside the firewall, before it is encrypted. We further learned that other people had discovered a series of security flaws in earlier browser versions (see [L96] for an overview). Unfortunately, the "tradition" of security weaknesses being discovered did not stop with us (see [K98] for an overview).

In March of 1998, Netscape decided to make the Navigator source code available to the public (under the name "Mozilla"). We consequently decided to implement a new security model for JavaScript in Mozilla. A technical description of the security primitives used in our model can be found in [AM98b].

We gave a demo of an early prototype "Bell Labs Mozilla" to Netscape in late fall of 1998. Given the positive feedback, we then went on to complete "our" Mozilla and ship it to Netscape in March 1999. As of this writing, our contacts at Netscape started to integrate our code into Navigator 5.0, currently scheduled for release in late fall, 1999. Check `www.mozilla.org` for updates on the progress of Mozilla.

In this paper we focus on how our new model benefits both the end user who surfs the Web and the JavaScript programmer who designs the Web sites that the end user visits. We also devote some of the discussion on our implementation experience integrating our model into the existing Mozilla source code base.

## 2 Brief Introduction to JavaScript

In this section, we give a very brief introduction to JavaScript; for more details see [F97, KK97]). *JavaScript* is a simple procedural language that is interpreted by Web browsers from Netscape. (*JScript*, Microsoft's implementation, is a clone that is interpreted in Microsoft's Web browsers.) JavaScript is object-based in the sense that it uses built-in and user-defined extensible objects, but there are no classes or inheritance. The code is integrated with, and embedded in, HTML. By default, JavaScript provides an object-instance hierarchy that models the browser window and some browser state information. For example, the *navigator* object provides information about the browser to a script, and the *history* object represents the browsing history in the browser window.

Also, through a process called *reflection*, JavaScript automatically creates an object-instance hierarchy of elements of the script's HTML document when it is loaded by the browser. The *location* object represents the URL of the current document, while the *document* object encapsulates HTML elements (forms, links, anchors, images, *etc.*) of the current document. The reflection process defines a unique *name space* for each HTML page and thus for each collection of scripts embedded in that page. JavaScript is *loosely typed*: variables' data types are not declared. JavaScript uses *dynamic binding*: object references are resolved at runtime.

## 3 Overview of our Security Model

In this section, we give an overview of the security primitives used in our model; for more details see [AM98b]. Our design is based on the following two basic building blocks: (1) *Access Control* regulates what data a script can access on a user's machine and in what mode; and, (2) *Trust Management* regulates how trust is established and terminated among scripts executing simultaneously in different contexts. See [AM98b] for a detailed discussion and justification of the security primitives employed and what common goal is being realized through them.

### 3.1 Security Policy and Access Control

Our JavaScript interpreter accepts as input, in addition to the scripts to execute, a *security policy* from the browser user. Different users may have different requirements with respect to their own privacy or that of the data they submit, and this will be reflected in their chosen security policies. Simply put, a security policy defines a partitioning of the JavaScript name space into inaccessible, read-only, and read-write objects.

A policy also defines the action for the JavaScript interpreter to take when the current script tries to execute an operation that violates the access control specification that the policy's name space partition defines.

A security policy further specifies which external protocols (*e.g.*, loading a `mailto:` or `ftp:` URL) the script is allowed to invoke and the appropriate action to take in the event that a script attempts to invoke a protocol not allowed by the policy.

### 3.2 Management of Trust

We use *access control lists* (ACLs) to regulate access that scripts have to objects in name spaces other than their own (*e.g.*, a page in a different browser window or frame). In a nutshell, a document's ACL is a list of URL paths or hostnames. Only a script whose origin appears in the document's ACL may access the name space of this page. The ACL mechanism allows Web developers to both expand and contract the set of domains that they trust. For example, `store.com` can put `partner.com` on the ACL of an HTML document that it serves to allow scripts from `partner.com` full access to the page. Also, `emall.com/store1` can prevent scripts from `emall.com/store2` from accessing its documents by setting its ACL to `emall.com/store1`.

The ACL provides an all-or-nothing control for access to a name space by other scripts. Another script is *trusted* by a document if the script's origin is listed in the document's ACL. Either every object in the document's local name space is accessible (to a trusted script), or none is (to an untrusted script). To complement this, we introduce a new method, `setPrivate`. If for any object `obj` a script executes `setPrivate(obj);`, then `obj` (and any of its properties) is subsequently inaccessible, even to trusted scripts.

## 4 The End User's View

In current browsers, the user's choice with respect to JavaScript is truly limited. He/she can either turn JavaScript completely off or on for all sites.

We introduce the notion of a *security policy* for JavaScript. From the user's perspective, a security policy is a bundled set of preferences with respect to the following capabilities given to scripts that execute on the user's machine:

- *Access to reflected objects*: A script has access to a number of objects in the JavaScript hierarchy. For example, `document.referrer` indicates the page from which the user arrived at the current page, and `navigator.platform` indicates the operating system on the user's machine. A policy aggregates access permissions to these *property policies* of all reflected objects.

- *Access to external interfaces*: A script also has access to a number of external interfaces, such as ftp (`ftp:` URL) and e-mail (`mailto:` URL) protocols. It also has access to calls in the Java language, through which it can capture the user's IP address, for example (by calling `java.net.InetAddress`). Again a policy aggregates access permissions to all external interfaces.

- *Actions in the event of access violations*: A policy also specifies the action to take by the JavaScript interpreter if a script attempts to violate the current policy.

Most end users will not want to be bothered with such low-level details as the exact specification of a policy. Thus we offer a small number of increasingly strict *predefined policies* from which the user can pick; see Figure 1. The chosen policy, the *global security policy*, will be in effect whenever the user starts visiting Web sites. A user can also pick predefined policies to be in effect only for specific sites (*site-specific security policy*). The user may specify either a hostname or a specific URL for which this policy should be in effect; see Figure 2. For example, it makes sense to allow a more lenient policy when browsing within an intranet than when accessing the external Internet. In fact, as part of an overall corporate security policy, the employees' browsers can be initialized with a strict policy for external sites and a liberal policy for internal sites.

As for many security tools (see, *e.g.*, [WT98, ZS96]), it is hard to design a user interface that, on the one hand, does not restrict the power user from fully exploiting the provided functionality, and, on the other hand, does not confuse the average user, the confusion leading to possible unwanted security implications. For example, [WT98] call the problem of choosing access rules and policies the *abstraction property* and observe that such notions are often alien and unintuitive to a wider user population. Another factor mentioned in [WT98] is that users get little feedback when they make an error in configuring security aspects. Consequently, we think it is very important that reasonable default settings be chosen for the average user and that good representative examples be chosen to explain in which situations a given policy is adequate.

We envision that corporate administrators will want to incorporate policy management (creating and updating policies, installing new policies on each desktop's browser, etc.) via some sort of directory service integration (e.g., LDAP-like solution). Home users might also want to have tools to easily create or download (from certifiably trusted site) and then install new policies on their desktop. Thus, policy management and its tools seem like a fruitful area of further work.

### 4.1 Signed JavaScript

Netscape Navigator 4 and later versions support digitally signed scripts that can request privileges, and, subject to user approval, lift certain security restrictions while executing. A digital signature allows the browser to securely establish the author of a signed JavaScript program (see [N98]). Cryptographically signed scripts are not yet very popular, partly because average users find it hard to grasp the privilege-granting process or the implications of granting a particular privilege.

For future versions of browsers we propose to integrate code signing into our model, by having specific security policies that go into effect if a signed script is downloaded from a particular site. For example, a Fidelity policy for the user's interaction with the brokerage house might allow reading and writing files in a specific directory, so that the user can study his account offline.

## 5 The JavaScript Programmer's View

### 5.1 Domain and URL Based Trust Management

JavaScript executes in the name space defined by both the browser window and the HTML page in which it is embedded. This name space is accessible to all scripts embedded in the same page. Standard JavaScript also grants access to that name space to scripts that run in other browser windows, but that were loaded by an HTML page that loaded from the same server. This *same origin policy* leads to the situation where scripts from, for example, `e-mall.com/pet-shop` can access all data (*e.g.*, credit-card numbers) that the user inputs into a form at

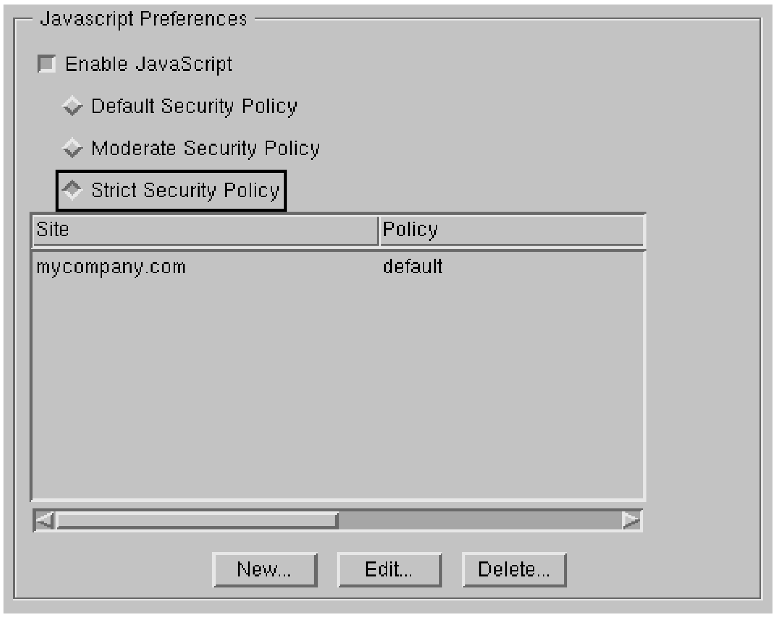


Figure 1: User Interface for Setting Browser Security Policy


Figure 2: User Interface for Setting Site-Specific Security Policies

`https://e-mall.com/toy-store/checkout.html`, given that the user has pages from both shops open at the same time. Note that this access is possible even though `toy-store` uses SSL to secure their client's data.

In our model, the JavaScript programmer can model trust *explicitly* by using ACLs. The `toy-store` programmer can state in the initialization step of scripting in all of the `toy-store` pages:

```
<SCRIPT LANGUAGE="Javascript">
document.ACL =
  "http://e-mall.com/toy-store";
...
</SCRIPT>
```

The above statement indicates that only scripts from a URL that is prefixed by the above element of the access control list (ACL) are allowed to access the page's name space. Thus, if a script embedded in `e-mall.com/pet-shop/snoop.html` executes the following:

```
<SCRIPT LANGUAGE="Javascript">
toy_store_check_out_window =
  window.open("http://e-mall.com
               /toy-store/checkout.html");
...
```

the page will be loaded into a new browser window on the user's desktop, but its name space will be inaccessible to the calling script.

If `toy-store` decides to collaborate with `baby-store` in order to cross-link, then the initialization might look like:

```
document.ACL =
   "http://e-mall.com/toy-store
    http://e-mall.com/baby-store";
...
```

If these two stores want to collaborate further with a site that is not even part of the `e-mall` domain (*e.g.*, the `parentsoup` site), then while the existing browsers do not allow this, our model can accommodate this easily by using the following:

```
document.ACL =
  "http://e-mall.com/toy-store
   http://e-mall.com/baby-store
   www.parentsoup.com";
...
```

The last entry above is a domain name, which includes all pages from that domain in the ACL.

### 5.2 Fine-Grained Trust Management via `SetPrivate`

The ACL-based approach is still an all-or-nothing approach in the sense that by including a page in its ACL, a script makes its *whole* name space available to the other page. This coarse granularity may not always be appropriate. The new security model allows Web developers to prevent access to sensitive information by marking it private by using the `setPrivate` method. We motivate this by using an example.

"Associates programs" are rapidly gaining popularity with e-commerce sites, such as `amazon.com`. In their associates program, `amazon.com` pays each participating site (like `associate.com`) a small percentage of a sale that results from a person's following a link from `associate.com` to `amazon.com`, *i.e.*, by a person who was *referred by* `associate.com`. Often, the associates program is realized by a third-party network, such as `linkexchange.com`. Currently, both `associate.com` and `linkexchange.com` simply have to trust `amazon.com` to provide at the end of each month a statement that accurately reflects the sales; the business relationship is clearly stacked in favor of `amazon.com` — since `linkexchange.com` and `amazon.com` are different domains, the name spaces on their respective pages are inaccessible to each other. The same applies for `associate.com` and `amazon.com`. At best the associate could know that the user clicked on their link leading them to `amazon.com` by appropriately instrumenting their Web page, and the third-party network could know that this happened if referrals were redirected through it. However, there is no way for either of them to verify that the user engaged in a purchase. While this may not a problem with a well-known site such as `amazon.com`, better accountability would boost these programs when other stores (`new-vendor.com`) are involved.

While using ACLs would circumvent the problem of lack of access, `new-vendor.com` may not be comfortable simply setting

```
document.ACL = "www.associate.com";
```

on all of its relevant check-out pages, as that could potentially reveal confidential client data (*e.g.*, credit-card number, etc.). However, `new-vendor.com` might be willing to reveal some of the non-confidential customer data, such as the fact that the referred client did indeed purchase a book and the amount paid. In our model, new-vendor.com would simply add the following code on the check-out page:

```
<SCRIPT LANGUAGE="Javascript">
document.ACL = "www.associate.com";
setPrivate(CreditCardForm, "elements");
....
</SCRIPT>
```

The above code allows access by scripts from www.associate.com to all the elements on the page except the form with the name "CreditCardForm". Assuming that all the client's confidential data is in this HTML form, it is now protected. We note that the statement setPrivate(document.forms[0], "elements"); is equivalent to the above, if "CreditCardForm" is the first form on the HTML page. But we caution that this version is less safe. If during page evolution, a new form is inserted in front of "CreditCardForm", the new form will be protected while "CreditCardForm" is all of a sudden accessible again to www.associates.com. Therefore we strongly recommend the use of names in setPrivate.

Now, when associate refers a potential client to new-vendor, its page can stay resident on the user's desktop in its own window or frame; if the client ever reaches the check-out page at new-vendor, associate will be able to read the relevant data used to verify new-vendor's end-of-month statements. Note that new-vendor.com can generate the necessary JavaScript automatically via server-side logic based on referrer information. The ability to selectively expose information realizes a better balance in the business relationship between a store and its associates. Furthermore, as soon as the user leaves new-store.com's Web site, the associate no longer has access to any information about the user.

A similar scenario exists for sites like shop.com that serve as centralized resources providing information about online stores. shop.com's customers (online vendors) provide it with enough information to organize stores into hierarchies, provide searchable interfaces, *etc.* A user browsing shop.com's Web site eventually clicks on a link on a page from shop.com and is sent to store.com's Web site, which is displayed as a frame on shop.com's page. As in the earlier example, shop.com only knows that the user clicked through to store.com, and is unaware of any activity that subsequently transpires.

The new security model allows shop.com's customers (store owners) to put shop.com on the ACL of pages they serve. store.com appropriately protects the information that it considers sensitive (*e.g.*, the user's credit card number) by marking it private. shop.com thus has access to the information that it needs for pay-per-click, pay-per-lead and pay-per-sale type scenarios.

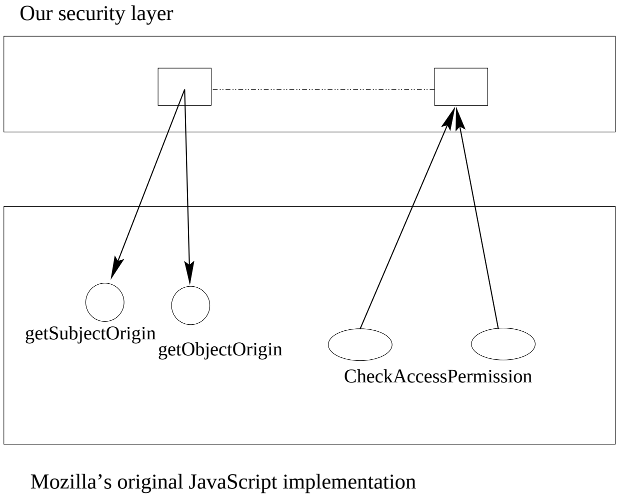


Figure 3: Security Layers

## 6 Security Code Layers and Hardening of the Mozilla Layer

Our implementation adds a new security layer on top of the existing Netscape code, realizing access control, security policies, and trust management as described in Section 3. We created a security layer API and added calls to it from Netscape's code, as depicted in Figure 3. (See Section 7 for further details.)

The robustness of the combined code depends on finding all the right spots in Netscape's code at which to interpose our API calls such that we close all back doors. At the same time, our implementation makes calls to basic functions in the Netscape code and therefore relies on the correct behavior of that code. Much of that code is devoted to identifying the subject and object origin URLs. (The subject origin URL is the place where the executing JavaScript code comes from. The object origin URL is the place where the JavaScript code comes from for the object being acted on.) If our code were to get the wrong information, it could possibly grant access inappropriately, thus opening a security hole. Given the importance of this basic code, we suggest a more methodical approach to realize these two basic functions. (See the subsequent subsections.)

Another area of concern is that object values persist across document loads in a window. Each document is supposed to form a separate context. However, in Netscape's current implementation, `window.name` maintains its value across document loads. A clever intruder could then access the information that was supposed to be destroyed with the object. While we have fixed the case of `window.name`, we chose not to close this hole in its generality, because we believe that the new Netscape document object model (DOM) would do so for us.

## 6.1 Finding the Subject Origin URL

JavaScript has a number of possible ways to pass the control flow or to generate an additional thread of execution:

1. Function/method call: *e.g.*, `v = foo(x);` or `u = otherWindow.foo(x);`

2. Dynamic generation of JavaScript code: *e.g.*, `document.write(foo(x););` or `myWindow.eval(foo(x));`.

3. JavaScript URL: *e.g.*, `otherWindow.location = "javascript:foo(x)";`

4. JavaScript invocation through HTML (browser):

   ```
   <a target=otherWin href="javascript:
      alert(window.location);" >
   ```

   Also, code invoked through installed event handlers, "script" tags, *etc.*

Most "traditional" languages only have function/method calls. In those cases it is usually fairly straightforward to determine which entity originated the call sequence by inspecting the call stack, *i.e.*, stack inspection. The additional methods and flexibility in JavaScript complicate this task. For instance, dynamic invocation has the flavor of a function call, but the passing of control does not take place instantly. In fact, a callee might be executing at a time when the caller is no longer on the stack. Therefore, we propose the use of **proactive forward passing** of the subject origin information. Whenever a passing of the control flow in the code is indicated (and whether it is about to take place right then or not), the interpreter sets the subject domain of the callee to the subject domain of the caller, in such a way that when the callee executes, this value can be easily retrieved.

We distinguish three cases:

- The callee will begin executing immediately. Either its stack frame is loaded right on top of the caller's frame (*e.g.*, regular function call) or its stack frame is loaded onto another document's stack frame (*e.g.*, `javascript` URL). Hence, the interpreter can propagate the subject origin information among stack frames (which logically form a tree structure).

- The callee will execute at a later point (*e.g.*, `document.write`). In this case, the callee will be the top-level stack frame in its document. Hence, the interpreter can store the subject origin information as an attribute of the document. Once the top-level stack frame gets loaded, the interpreter fetches the subject origin information from the document attribute and initializes the subject origin value in the top-level stack frame.

- JavaScript is invoked through HTML. In this case, the callee executes in a top-level frame of the JavaScript stack of its document. Hence, the browser should play the role of a JavaScript caller as far as the passing of the subject origin to the top-level frame is concerned.

Our goal is to arrive at a situation where, for every stack frame loaded, the correct subject origin can be retrieved from a well-defined location and, consequently, there will never be the need to search backward for it or even to fail and declare "unknown origin", as might occur in the current version of JavaScript (4.x browsers).

## 6.2 Finding the Object Origin URL

The object origin is always defined by its static scope. That is, global objects always derive their origin from the origin of their document. Similarly, an object local to a function or method derives its origin from the origin of the method. A reference to a variable $x$ in a document $d$ loaded into window $w$ is really $w.x$ and thus derives its object origin from the origin of $d$. Let $y$ be a local variable to the function `foo()`, located in a document $d$ in window $w$. Even when `foo()` is called from a different window `otherWin`, *e.g.*, when "`z = w.foo();`" is a line in `otherWin`, the object origin of $y$ is still determined by its static scope, the origin of $d$. Note that in "`otherWin.location = "javascript: x = 1;";` ", the "`"javascript: x = 1;`" part is just a string and not code; thus the static scope of $x$ is within the window `otherWin`.

Dynamically created documents (*e.g.*, created by `document.write`) should always inherit origin from the creating window/document.

Here are some examples for code executing in a window $w1$, loaded from origin $o1$:

```
v = foo(x);
  // o1 remains subject origin
  // for function ‘‘foo’’
v = w2.foo(x);
  // o1 remains subject origin
  // for function ‘‘foo’’
  // foo will execute in w2’s scope
w2.location = "javascript:foo()";
  // o1 remains subject origin
  // for function ‘‘foo’’
  // foo will execute within a
  // top-level stack frame
  // of a new document in w2.
w1.document.write(foo());
  // o1 remains subject origin for
  // function ‘‘foo’’
  // foo will execute in a new
  // document in w1, whose origin
  // is the same as the one of
  // the current document
w2.eval(w1.foo(x));
  // subject of this statement is o1 and
  // thus is subject of foo.
  // foo will execute in w2’s context,
  // since w2 owns the eval method.
```

Thus, in this context, the object origin can always be retrieved in one step, by accessing the enclosing scope.

# 7 Our Implementation

Our implementation adds a new security layer on top of the existing Netscape code, realizing access control, security policies, and trust management as described earlier.

## 7.1 Overview

With the exception of user interface code to support site security policies, nearly all the code that was modified directly supports the JavaScript object model.

Property policies are implemented in the respective modules for their objects. They control whether there is read-write, read-only, or no access to the property. External interface policies are handled by the code that sets a URL object’s value. They control whether there is read-write, read-only, or no access to the external interface.

Our implementation depends on correctly identifying subject and object origin URLs. The subject origin URL determines which site security policy to use. The subject and object origin URLs together determine ACL behavior.

When a policy violation of any kind occurs, the implementation always presents an error dialog to the user. Based on the value of a configurable continuation preference setting in the current policy, the JavaScript interpreter may then stop interpreting the offending script, or it may continue, while denying the requested access.

## 7.2 Policy Lookup

To look up a policy $P$ for a property or external interface, our implementation first checks whether there is a site security policy for the executing script’s origin URL. Otherwise it uses the current default global security policy. It then checks whether there is a preference for $P$ in that security policy.

Because we allow site security policies to apply to URLs, and not just to hostnames, the site security policy lookup uses the longest (and, therefore, presumably most specific) policy that matches the subject URL. For example, assume there are site security policies for `e-mall.com` and `http://e-mall.com/store1/`. The subject origin URLs `http://e-mall.com/index.html` and `http://e-mall.com/store2/` would both use the first policy, whereas, `http://e-mall.com/store1/catalog.html` would use the second.

## 7.3 Property and External Interface Policies

The per-property and per-external-interface policies were easy to implement. At the point in the code where a get- or set-property function is implemented, the modified code checks whether there is a corresponding property (external interface) policy in effect, and, if so, whether the requested access violates it (for example, attempting to write a read-only property). Because this check happens frequently, performance optimizations should be considered, such as caching previous results of checks or building a hash table that can be efficiently queried to check whether an object’s access is affected by an existing policy. A vast majority of objects won’t be affected since a typical policy covers only a few security-sensitive objects.

On a violation, the implementation checks the continuation setting for the relevant security policy and either aborts interpretation or continues without granting access. If there is no violation, interpretation proceeds normally.

Access to the new `document.ACL` property is a special case. We unconditionally restrict access so only the script that created the `document` has permission to read or write `document.ACL`. Otherwise a rogue script could attempt to change `document.ACL` and gain access to the objects that ACLs protect.

### 7.4 Access Control List (ACL) Support

Because an ACL is tied to a `document` object, the default ACL is the subject origin URL that created the document. However, as stated earlier, that script may change the value of `document.ACL` to enlarge the set of URLs that can access objects that the ACL protects.

We implement ACL checks in those places in the interpreter code where one object attempts to access an object that might have arisen from a different context. The implementation checks the ACL for the (containing object of the) object being accessed against the accessing script's (subject) origin URL. If the two are identical, or if one of the URLs in the ACL is a prefix of the subject origin URL, then the check succeeds, and access is granted. Otherwise an ACL violation occurs. As above, the continuation policy on ACL violation is controlled by a preference, but access to the protected object is never granted if a violation occurs.

As described above, our current implementation uses only prefix matching for ACL checks rather than fully general regular expressions, which are costly to evaluate. Logically, an ACL check has to be performed for each object access. Note however, that the ACL check actually has to be performed only once for each ACL/subject pair if we cache the outcome of the check (and adjust or flush the cache when there is an assignment to an ACL). Thus under such cache optimization, the use of regular expressions becomes feasible.

### 7.5 setPrivate, unsetPrivate

Even if access is granted according to the ACL, the specific object may have been `setPrivate`. The implementation does a separate check for whether the object is `setPrivate`, and there is a separate continuation policy for failed attempts to access one.

As with `document.ACL`, we only allow the script that created a document to `set` or `unsetPrivate` objects.

### 7.6 Hierarchy of Tests: "Hedging Your Bets"

The various tests outlined above and summarized here must *all* succeed before access to an object or property is granted. The order of checks is something like this:

1. Check whether the (JavaScript) script is signed, and if so, whether the signature is valid. (The current Netscape security model does not allow access to some objects/methods unless the script presents a valid signature. One example is the user preferences object, `navigator.preferences`. We integrate this approach into our access control.)

2. Check for ACL violation.

   - Determine whether subject has permission to access object.
   - Check that the accessed object has not been `setPrivate`

3. Check for property policy violation

   - Determine which security policy applies (in order): site policy, global policy, default policy.
   - Check whether there's a property policy under the applicable security policy.
   - Check whether the property policy has been violated

Note that this hierarchy means that even a signed script is not granted unconditional access to JavaScript objects. A signed script makes some parts of the object model accessible that otherwise would not be, but the signed script's code is still subject to the same set of checks as any other script.

### 7.7 User Interface Changes

We pre-defined three model security policies: `strict`, `moderate`, and `default`, where `default` corresponds to Mozilla's behavior prior to our changes. When a user enables JavaScript in Advanced Preferences, she can now select one of these three security policies as her global security policy. Moreover, there is a table of site security policies, initially empty, where she can add, delete, or edit site security policy selections. This facility allows her to enter a hostname or URL in a text field and to mark a checkbox for the desired policy (`strict`, `moderate`, `default`).

## 8 Summary and Outlook

We have described the benefits of our model and implementation for the end user and the JavaScript programmer. Our model furthermore benefits the Mozilla developers: If new security bugs emerge, browser developers can use specialized fine-grained policies that control access to sensitive objects in order to identify paths that need to be followed to mount a successful security breach. Furthermore, in case of such a security bug, Mozilla can make available specially tailored policies to download, which protect against exploitation of this security bug. This is a more desirable course of action than the current practice of suggesting turning off JavaScript entirely until a bug fix is available. (Our current implementation allows only predefined security policies, but it could be extended to provide for user-defined policies.)

Our positive experience with using site-specific security policies indicates that such policies for the whole browser (on-off for cookies, Java, JavaScript, *etc.*) should be considered.

We hope that the final implementation will be scrutinized early on by the Mozilla open source community, so that remaining weaknesses can be identified before they become actual "bugs".

**Acknowledgments:** We thank Murali Rangarajan, who did the initial work on the implementation of the new security model. We are grateful to Norris Boyd and Tom Pixley at Netscape for their help and encouragement. Finally, Eric Brewer at Inktomi, in his role as USITS shepherd, helped us to make a number of improvements.

## References

[AM98] V. Anupam and A. Mayer, *Security of Web Browser Scripting Languages: Vulnerabilities, Attacks and Remedies*, Proc. 7th USENIX Security Symposium, January 1998.

[AM98b] Vinod Anupam and Alain Mayer, *Secure Web Scripting*, *IEEE Internet Computing*, Nov/Dec 1998.

[CERT97] CERT* Advisory CA-97.20, *JavaScript Vulnerability*, CERT Coordination Center, July 1997, `ftp://info.cert.org/pub/cert_advisories/CA-97.20.javascript`.

[F97] D. Flanagan, JavaScript: The Definitive Guide. *O'Reilly and Associates*, January 1997.

[KK97] P. Kent, J. Kent, Official Netscape JavaScript 1.2 Book *Netscape Press & Ventana.*

[K98] E. Kubaitis, WWW Browser Security and Privacy Flaws, `http://www.cen.uiuc.edu/~ejk/browser-security.html`

[L96] J. R. LoVerso, *JavaScript Security Flaws*, `http://www.schooner.com/~loverso/javascript/`

[L97] P. Lomax, *Learning VBScript*, O'Reilly and Associates, July 1997.

[N98] Netscape Corp., *JavaScript Security in Comminicator 4.x*, `http://developer.netscape.com/docs/manuals/communicator/jssec/contents.htm`

[WT98] A. Whitten and D. Tygar, *Usability of Security; A Case Study*, CMU Tech. Report 98-155.

[ZS96] M. E. Zurko and R. Simon, *User-Centered Security*, *New Security Paradigm Workshop*, 1996.