SECURE WEB SCRIPTING

VINOD ANUPAM AND ALAIN MAYER
Bell Laboratories—Lucent Technologies

Current Web scripting languages lack an explicit security model. The model proposed in this article has been implemented for JavaScript in the Mozilla browser source code.

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 is no longer restricted to 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[1] and Microsoft's VBScript.[2]

At the same time, scripting also adds to the power of an adversarial entity. Users might easily visit a dubious site and unknowingly download scripts to their detriment. To ensure user security in the presence of these scripting capabilities, the interpreter's task is to deny scripts direct access to user resources such as files and network resources. However, the design of current scripting languages does not incorporate an explicit security model. Consequently, it is not surprising that a number of serious security flaws have been uncovered since the inception of JavaScript and VBScript (for example, see LoVerso[3]). Some of these flaws expose even users sitting behind a firewall or using encryption.[4]

Most of the scripting language vulnerabilities uncovered thus far originate within the language itself and execute without penetration of the underlying operating system or concurrent application. Hence, a security model for browser scripting languages must give the following assurances:

- *Data security*. Data provided by the user (possibly encrypted before it is transmitted) into an HTML form can only be accessed by the intended recipients.
- *User privacy*. Information about the user cannot leave the user's machine unless explicitly allowed by the user.

We have developed a security model for scripting languages to meet these requirements. It is realized by a "safe" interpreter and based on three basic building blocks:

 1089-7801/98/$10.00 ©1998 IEEE

- *Access control*, to regulate what data a script can access on a user's machine and in what mode,
- *Independence of contexts*, to ensure that two scripts executing in different contexts (for example, simultaneously in different browser windows or sequentially in the same browser window) cannot access each other's data at will, and
- *Trust management*, to regulate how trust is established and terminated among scripts executing simultaneously in different contexts.

We also advocate a clear separation between a security policy and an implementation. Different users require different degrees of privacy and security, which translate to different degrees of flexibility when interacting with a Web server; these differences can be expressed in different security policies. A sound implementation, however, should be universally applicable. These are principles that first appeared decades ago in work on secure operating systems.

In this article, we present the design of an example security model. Given the free availability of Netscape's Mozilla browser source code, we focus on JavaScript, but our ideas apply equally well to VBScript or similar languages.

THE JAVASCRIPT LANGUAGE

JavaScript is a simple procedural language that is interpreted by Web browsers from Netscape. (JScript is Microsoft's implementation, a clone that is interpreted in Microsoft's Web browsers. In this article, the term JavaScript refers to both strains.) JavaScript is object-based in the sense that it uses built-in and user-defined extensible objects, but it has 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, a *navigator* object provides information about the browser to a script, and a *history* object represents the browsing history in the browser window.

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

EXECUTION ENVIRONMENT

There are six basic entities on a single machine in our execution environment:

- *Windows*. A concrete example is a browser window. A user can have multiple windows open on a machine at any point in time.
- *Contexts*. A window and its content (for example, the current HTML document displayed) define a context. Thus we can describe a context C as $C = (w, URL)$, where w is a reference to a window and URL is the location of the HTML document.
- *Scripts*. These are JavaScript fragments embedded in an HTML document. (For details on different ways of embedding JavaScript code in HTML, see Chapter 10 in Flanagan.[1]) A document may have multiple scripts embedded in it. All of them share the window's context.
- *Interpreters*. An interpreter executes the embedded script within the window's context.
- *Name spaces*. A name space is the hierarchy of objects potentially accessible to a script executing within the window's context.
- *External interfaces*. A script may also be able to invoke actions external to its context via an external interface. For example, a script can cause the browser to open HTTP/FTP connections.

JavaScript provides an object-instance hierarchy that models the browser window and some browser state information.

The execution environment allows a script to establish a trust relationship between its context and another window's context, which enables it to access the name space of the other context via a reference. For example, Figure 1 depicts three open windows (w, w', and w'') on a user machine, where scripts in w can access the name space of w'', and scripts in w' can access some external interfaces, giving those scripts additional capabilities.

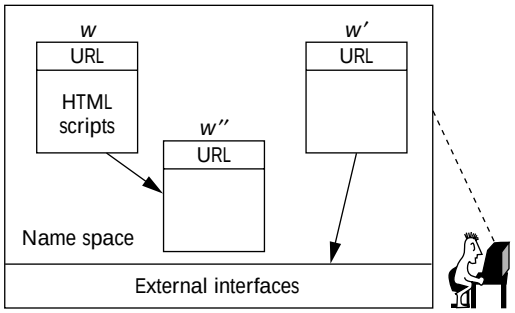

Figure 1. Execution environment for JavaScript security model. The name space defines a hierarchy of objects potentially accessible to a script executing within a window's context.

In cases where a browser window contains multiple frames, each of which is a subwindow containing a separate HTML document, our security model considers each frame to be an independent window.

SECURITY POLICY AND ACCESS CONTROL

JavaScript has a predefined name space, derived from the JavaScript object-instance hierarchy. We can denote this name space by N^i. Scripts can also create language objects such as variables and arrays, which we denote by N^s. We denote the union of N^i and N^s by N, the entire name space of a script. (For an illustration of these terms, see the sidebar below, "Name Spaces: Predefined and Scripted.")

Partitioning the Name Space

In addition to the scripts to be executed, our interpreter accepts a security policy from the user as input. This policy can reflect the different requirements users have with respect to their own privacy or that of the data they submit.

A security policy p defines a partitioning of N into N_w^p, N_r^p, and N_x^p—respectively, the writable, read-only, and totally inaccessible objects of N. Note that N^s is a subset of N_w^p for all policies p: Any object created directly by a script must be writable by the script. Other objects in this partition may also be writable. For example, the value property of most form elements is intended to be modifiable by scripts, and thus belongs to N_w^p.

A user might choose a policy p that places a document.lastModified object from the JavaScript hierarchy in N_r^p, the read-only set of objects in N^i; this semantically nonmodifiable object contains the date and time of the most recent modification of a server document. On the other hand, to protect privacy, a user might choose a policy p that places a document.referrer object in N_x^p, the set of objects in N^i excluded from any kind of access, because it reflects the URL of the documents from which the user reached the current document.

In summary, proper access control implies that the safe interpreter, given a policy p as input, ensures and implements two different kinds of partitionings of N.

- Partitioning by the access a script has to an item in the object hierarchy—that is, N_w^p for items

NAME SPACES: PREDEFINED AND SCRIPTED

The JavaScript environment is defined by its object-instance hierarchy: Objects have properties that may be other objects.

- N^i is roughly equivalent to the entire JavaScript object hierarchy at the time an HTML document is being loaded into a window.
- Every object created directly by a script belongs to N^s.

The *window* object is the root of the JavaScript object-instance hierarchy. It has several properties: *window.name* is a string that contains the browser window's name, *window.document* contains HTML elements reflected from the current document, *window.navigator* encapsulates properties of the browser software, *window.history* represents the browsing history in that window, to name a few. All of these properties are created by the interpreter when a context is loaded and hence belong to N^i.

For more information on JavaScript's predefined name space, see Flanagan.[1]

In contrast, consider the code fragment *var foo; foo = "bar";*. This assignment is equivalent to *window.foo = "bar";* and results in the window object getting a new property *foo*, whose value is the string *"bar"*. This property is created directly by a script. Consequently, *foo* belongs to N^s.

REFERENCE

1. D. Flanagan, *JavaScript: The Definitive Guide*, O'Reilly and Assoc., Sebastopol, Calif., 1997.

that are both readable and writable, N_r^p for items that are read-only, and N_x^p for items that are inaccessible.

- Partitioning by the item's creator—N^s for items created by a script and N^i for items created by the interpreter.

Initial values of some of the items in N_i depend on the window in which the context is loaded and activated—that is, they depend on the execution environment. The role of the safe interpreter is to ensure correct initialization.

A policy p also defines the action taken by the safe interpreter when the current script tries to execute an operation that violates p's access control specification defined via the partitioning of N. For example, the script might attempt to read document.referrer, while this item is in N_x^p. Possible actions are to silently abort the operation and continue executing the script, to abort the execution of the script, or to notify the user via an interactive dialog that presents choices for how to proceed, and continue or abort as specified by the user. The security policy also specifies which external interfaces a script is allowed to access and the action of the safe interpreter when a script invokes a disallowed call.

To hide the complexity of partitioning the whole name space from the average user, we designed a small number of predefined, sensible policies so that the user can choose one policy as part of the browser preferences. Hence, at any given time, one policy p is in effect per browser client and pertains to all of the browser's open windows. A useful generalization allows the user to specify site-specific policies that are used when documents from a specific site are viewed. For example, the user might choose a more liberal policy when viewing pages from within an intranet, but a less permissive policy when viewing pages from the external Internet. Figure 2 depicts our user interface in Mozilla, which allows the user to select appropriate global and site-specific policies.

The security policy approach is similar to the approach taken for Java in JDK 1.2.[5] It allows the user to specify the access modes of potentially sensitive objects in the JavaScript name space and the required action in the event a script attempts a disallowed access. This approach simplifies issues of backwards compatibility as the JavaScript language evolves. For instance, a user can choose a policy that allows scripts using an older version of JavaScript to execute and fail gracefully in the event of a disallowed access. Furthermore, security policies can also

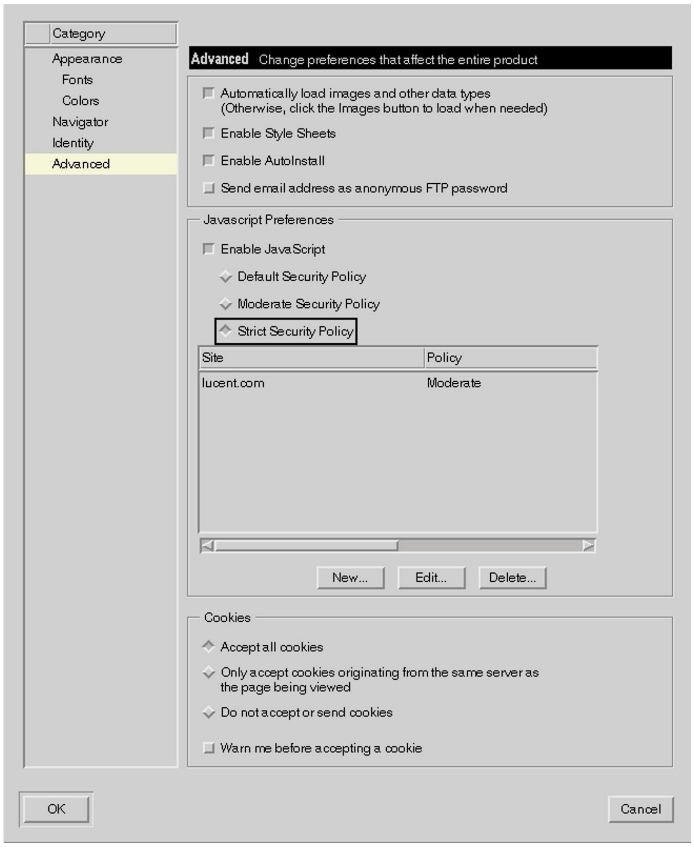

Figure 2. The new user interface in Mozilla to allow a user to select appropriate global and site-specific security policies.

be useful in conjunction with code signing (see the sidebar "Signed Scripts" on page 50).

Access to External Interfaces

The current version of JavaScript (1.2) makes several external interfaces available to scripts. Many of the documented attacks on a user's private data have exploited unregulated access of a script to an external interface. We therefore propose having a security policy control access to each external interface.

A script can send data that a user previously entered into an HTML form, back to an origin server: A script chooses the origin server URL via the form.action command and the submit method via form.method, and submits the user data via form.submit. The script can choose from a variety of protocols by specifying the appropriate prefix in the URL set in form.action. While the most common is HTTP, the script can also choose FTP (file transfer), SMTP (electronic mail), and even Telnet.

SMTP and FTP requests potentially reveal a user's e-mail address. Furthermore, a user might be

browsing via a privacy HTTP proxy that hides the user's IP address and lose this protection unwittingly when being switched to another protocol. A security policy p specifies which protocols the script is allowed to invoke and the appropriate action in the event that a script attempts to invoke a protocol not allowed by p. For example, the submission of data via a particular protocol might be aborted, or the safe interpreter might require interactively getting a user's approval.

A script can load a file from the user's local disk into the browser by using FILE (or ABOUT) in the protocol part of the specified URL. It is the task of trust management (discussed below) to ensure that no non-local script can access this data.

A script can also potentially execute local JavaScript code by specifying a local file as the source of a referenced script. We suggest that the security policy specify which local files can be referenced in this way.

A script can invoke Java (via interfaces like LiveConnect) and thus gain the power of a Java applet. While the Java design includes many features to ensure a safe environment,[5] combining two different security policies is a largely unexplored topic and can lead to unexpected outcomes, as alluded to in Levy et al.[6] We suggest that the security policy include explicit access permissions regulating a script's calls to LiveConnect and hence its Java capabilities.

INDEPENDENCE OF CONTEXTS

To ensure independence of different contexts, our safe interpreter enforces the following restriction:

- For each active context, N_w^p must be disjoint from N of any other context.

If an active context C is terminated and a new context C' is loaded and activated in the same window, our safe interpreter performs the following operations to transform C's name space N into the initial name space N' for C', so that N' does not depend in any way on C:

- Remove and rebuild N^i: Delete the object instance hierarchy and rebuild it according to the newly loaded HTML document. This

SIGNED SCRIPTS

JavaScript 1.2 includes the notion of code signing. The author of a script can add a digital signature to it. A signed script typically requests expanded privileges, gaining access to restricted information. It does so by using LiveConnect and the new Java classes referred to as the Java Capabilities API. The browser can verify a digital signature and, depending on the user's trust toward the author, grant the privileges. A script can make requests from a set of privileges, including *UniversalFileRead*, *UniversalBrowserWrite*, and so on.

We will now show how our framework fits in nicely with signed scripts by raising the level of abstraction offered to the user. We first need the following definition: A *partial order* on security policies, where $p^1 = p^2$ (stricter), holds if and only if

- $N_x^{p_1} \supseteq N_x^{p_2}$ and $N_w^{p_1} \subseteq N_w^{p_2}$ (that is, more objects are inaccessible in p^1 as compared to p^2 and fewer objects are writable.
- The set of allowed external interfaces in p^1's is a subset of the corresponding set in p^2.

We suggest that if a signed script requires a set of privileges (called *targets*), this set is translated into the strictest policy p among the set of predefined policies that allow for these privileges. Then the identity of the author together with the request for p is presented to the user.

In this fashion, a user can base the granting decision on a high-level security policy, rather than on a collection of low-level targets, where the consequences of granting them are difficult to understand. Our approach also requires a script to declare its intended privileges up front, rather than asking the user for each target separately, which puts an unnecessary burden on the user and introduces the problem of having to abort a script when the user does not grant the privilege (target) to the script.

Our approach further enables us to introduce a more fine-grained selection of targets. For example, if a user visits an online broker (say, *fidelity.com*), it might make good sense to enable scripts to read and update files on the user's machine, when they reflect the user's account with *Fidelity*. However, the user would not want to enable these scripts to read files about the user's bank account with say, *Chase Manhattan*.

This approach is not possible in JavaScript 1.2, where the *Fidelity* script would require the *UniversalFileWrite* target, and thus could read/write the *Chase Manhattan* files. We therefore suggest support for finer grained, low-level targets. These targets can be bundled into security policies, some of which are possibly tailored toward the sites that a user chooses to interact with.

process must ensure that while rebuilding, each item in N_w^p is set to a neutral value (null).
- Delete N^s: Remove all its items.
- Invalidate all references to N within other contexts, which have been established by a trust relationship.

Figure 3 illustrates these operations.

We designed a shadow object hierarchy through which a script's access to each object is redirected. If the browser unloads a context (and loads a new one), the safe interpreter destroys the current shadow hierarchy and builds a new shadow hierarchy for the new context. This ensures that objects created by a script in the old context cannot be accessed by a script in the new context.

If a document is unloaded and a new one is loaded within the same window, objects like window.navigator or window.name represent the same browser and window name in both contexts. The safe interpreter ensures that such objects—that is, those that exist in both the old and new context and are writable by a script—have their value set to null. For instance, if a value of any writable item in these objects persisted during the reload, then scripts in different name spaces could communicate. This could be exploited, for example, to implement nonpersistent cookies. Even worse, some past attacks exploited the fact that JavaScript did not enforce strict separation of independent contexts (for example, see LoVerso[3]). In the worst case, small overlaps in contexts open the door for Trojan Horse scripts in one context to obtain user-supplied data in a different context; in a less severe scenario, they can implement covert channels. Proper design of the safe interpreter must therefore ensure complete independence of contexts.

Given this discussion, it follows that garbage collection based on simple reference counting is not sufficient to ensure security since it will not collect all objects that should be collected. For example, if an object in document d is referenced by a script in document d', it would not be collected when d is unloaded.

MANAGEMENT OF TRUST

While most contexts are typically independent, certain applications greatly benefit from establishing trust relationships among contexts in different concurrently active windows. For example, trust relationships allow the coordination of information when presented to a user in multiple windows.

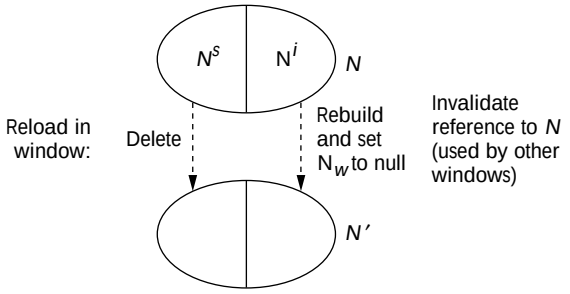

Figure 3. Safe interpreter's tasks when unloading an old context and loading a new one into the same window.

Proper design of the safe interpreter must ensure complete independence of contexts.

In JavaScript 1.2, trust is regulated by the Same Origin Policy, according to which a script can access a different context only if it originates at the same domain. This approach has two weaknesses. First, it is not flexible enough. Consider two users on aol.com or two shops located at e-mall.com. There is no inherent reason why all the users in these two domains should trust each other. Alternatively, lucent.com and bell-labs.com are two domains belonging to the same company and thus might want to establish trust—something that is not possible at this time.

A second weakness in the Same Origin Policy is that it does not apply to all items in N, only to those that are considered "sensitive."

We will now show how to maintain a secure and adequately flexible system. To this end, we need to specify two things:

- when a script in context C is allowed to establish a relationship with context C' and
- the subsequent privileges issued to scripts in C and vice versa.

With regard to the second point, the safe interpreter provides scripts in context C with a reference to context C'. Via this reference, a script in C can read all items in N_r^p, read and write all items in N_w^p, and insert items into N^s. Note that the specification for independence of context implies that if a script in C inserts an object into N^s, this object will only be

accessible to C as long as C' is active. After that, C loses its reference to the window in which C' resides.

However, this all-or-nothing approach is not sufficiently flexible. To achieve a more fine-grained approach, we introduce a new object property, the tag private. If for any object obj in N, a script sets obj.private = true;, then obj (and any of its properties) is subsequently invisible to a script accessing the current name space from any different context C'. Similarly, we introduce the readOnly property for every object. If a script sets obj.readOnly = true;, the property values of this object cannot be changed by a script from another context.

CGI files are a bit tricky in the ACL context, since they may all be located in the same directory for a server.

For example, a script might try to execute

```
victim =  window.open(''spy'',
          ''www.vendor.com'')
snoop =   victim.document.forms[0].
          elements[0].value
```

assuming that in victim's context, the first field in the first form asks the user for a credit-card number. If that field was specified to be private, then the safe interpreter would not execute the statement. Instead, it would execute the action specified in the user's security policy.

Global Access Control List

Assume context C is a document d with URL u in window w. We propose that a script in context C makes a request to establish a trust relationship with another context C' (or more concretely, another window w' containing a document d' with URL u') via either a w.open(u') or window.connect(u') method invocation (the latter method does not currently exist in JavaScript 1.2).

The safe interpreter then must decide if such a relationship is permissible. This decision will depend on the access control policy. We propose the use of access control lists (ACL). We introduce an object document.ACL in the object hierarchy, to be placed in N_r^p (for all possible policies p) and in N_i. For protection, this object is initialized with the tag private set to TRUE. This object represents the document's global ACL. Its value is a set of URL paths, specified as a comma-separated list. Each entry in the list describes either a single document or a directory; in the latter case, all documents in the directory are included.

Regular expressions are used to alter the scope of an entry. Consequently, a script in C only succeeds in establishing a trust relationship with C' if document.domain for d is in document.ACL for d'. This gives Web site developers (and JavaScript programmers) explicit fine-grained control over specifying whom they trust.

CGI files are a bit tricky in the ACL context, since they may all be located in the same directory for a server. However, regular expressions can be used to differentiate between trusted and untrusted CGI scripts within the same directory. The default ACL, if no value is set for document.ACL, is the URL path of the directory where the document is located. This condition is more restrictive than the current default in JavaScript (the domain of the current document URL), and is hence not backwards compatible.

We can now define the window.open() and window.connect() methods in more detail:

- A script in context C = (w, URL) invokes w.open(w', URL'), where w' is a new (that is, not currently open) window: If d.domain is in d'.ACL (where d' is the document referenced by URL'), then a new window w′ is created and d' is loaded into w′; a trust relationship is established.
- A script in C invokes w.open(w', URL'), where w' is an open window with a document having URL'' (possibly URL'' = URL'): If d.domain is in d'.ACL, then d'' is unloaded and d' is loaded into w'; a trust relationship is established.
- A script in C invokes w.connect(w', URL'): If d' is already loaded in window w' and d.domain is in d'.ACL, a trust relationship is established; otherwise it is handled like a w.open(w', URL') call.

If the call returns successfully, a reference to w' is returned in all these cases.

Local Access Control List

The global ACL implements a reasonable trust/security policy (not unlike a Unix file system enhanced with ACLs), but allows that "the right to exercise access carries with it the right to grant access." This right is well known in secure operating systems (see Kain and Landwehr[7] and Gong[8]). For

example, if a script in context C' is allowed to read (and hence copy) an object in context C, then every script in a context C'', such that C'' is in C''s ACL, can read this object. For more sensitive information, such transitivity of trust might not be suitable. For instance, bugs in the design of C' can lead to unintended values of the ACL of C', which then allows scripts of an adversarial context C'' to access data in C, undetectable by C.

We offer a more flexible approach: Each object is associated with its own local ACL in addition to a context-global ACL. The local ACL is the same as the global ACL when the context is loaded. In secure operating systems, the data plus its data security attributes is called a *segment*.[7] When an object gets copied from one context to another, the segment remains intact; in other words, the safe interpreter copies the ACL as well. The open routine works as described for the global ACL; however, if a relationship is established, scripts in C get a reference only to those items in C', for which C is in their local ACL. Hence, a script's context now plays the role of its capability, which is checked against the ACL of the segment of the item made accessible.

A segment's local ACL is not part of the scripting language proper and consequently should be physically stored within the safe interpreter, inaccessible to any script. Segments copied from a different context might have a different ACL.

With this security approach, bugs in context C' no longer allow access of third parties to context C. A somewhat difficult aspect of this scheme is to unequivocally define the semantics of "copying" an object so that the definition guides when the object's ACL must be updated.

Assignments are straightforward. Many scripting language allow control-flow constructs, such as `if ... then ... else` or `for ... do ....` If a conservative policy is assumed, then assignments made in the scope of a control-flow construct should be considered as "copies" of values read in the control part. This is similar to, for example, the tainting facility of PERL (see also the discussion of flow analysis in Volpano, Smith, and Irvine[9]).

If the option of local access control is used, then every object in the JavaScript object hierarchy has a property ACL, its local access control list. Once again, the ACL property belongs in N^p_r. When a context C is loaded and items in N^i are created, their respective ACL is set to the value of `document.ACL`.

Items created by scripts (and hence in N^s) also have the same initial value of their ACL. Now consider the example where a script in context C executed `window.open`, which returned a reference "victim" to another window (context) C'. The script can now execute the following code:

```
creditCardNum = victim.document.forms[0].
                elements[0].value
```

assuming that the first element of the first form of context C' asks the user for a credit-card number.

Independence of contexts ensures that there are no "hidden channels" among scripts in different contexts.

A side effect of this code is

```
creditCardNum.ACL = victim.document.ACL
```

The script in C now executes `window.open` again to open another window with a third context C''. A script in C'' could potentially execute `snoop = window.opener.creditCardNum`. However, before this code is executed, the safe interpreter verifies that C'''s `document.domain` is included in `window.opener.creditCardNum.ACL`.

HOW THE PIECES FIT TOGETHER

The concepts of access control, independence of contexts, and trust management combine to form sound support for data security and user privacy, as defined earlier in the article.

User Privacy

All data on a user's machine that is not directly related to the current HTML document should be considered private to the user. Our notion of access control, inferred by a user-chosen security policy, provides a "padded cell" for scripts, which ensures that they cannot access the file system or other unsafe resources. By maintaining a clear partitioning of the name space into inaccessible, read-only, and writable items, access control ensures that scripts only have access to those parts of browser- and window-related data that do not compromise a user's privacy while browsing.

The security policy also regulates access to external interfaces. Furthermore, independence of contexts ensures that there are no "hidden channels" among scripts in different contexts. For example, if a writable item persisted across changes of context (as is currently the case in JavaScript), it could be used as a user-invisible (albeit nonpersistent) "cookie" accessible to collaborating Web sites.

Accesses to user data by external interfaces are guarded by access control mechanisms.

Data Security

Data provided by a user in a context *C* of a window *w* (for example, by filling out a form in the context's HTML document) is only available to scripts in a different context *C′*, if *C′* is in *C*'s ACL. Scripts in any other context however, are not able to access this data.

Furthermore, setting the private property of an object guarantees that only scripts of the same context can access that object—ensured by trust management.

Independence of contexts ensures that any context *C′* activated after *C* in *w* cannot access any data written in *C*. Furthermore, if *C″* has a reference to a context *C′* that was loaded in *w* before *C*, this reference is set to null at the time *C′* is unloaded—something ensured by independence of contexts. Hence, only scripts in trusted contexts can access the user input data.

Accesses to user data by external interfaces are guarded by access control mechanisms, inferred from a user-chosen security policy.

IMPLEMENTATION

In spring of 1998, Netscape made the source code of the Navigator freely available under the name Mozilla (see www.mozilla.org). We have since implemented most of our security model in the Mozilla source code. We give a brief overview of our efforts here. A more detailed account is forthcoming.[10]

- *Security policy and access control*. We have implemented a security policy facility that imposes proper access control on the JavaScript name space and supports different actions in the event of access violations. This facility also allows users to specify access controls to be used for external interfaces, except for Java, since the initial version of the Mozilla source code did not include a Java virtual machine.

  For demonstration purposes, we have chosen three increasingly strict predefined policies. Users can choose one of them in their browser's preferences window, just as they would any other security or privacy preference, such as enabling Java or JavaScript. The current challenge for a general-use implementation is the design of a user interface that provides succinct but adequate information to assist an average user in choosing a policy.
- *Independence of contexts*. We have implemented the full functionality of this part of the security design. It protects against accidental leakage of information between a JavaScript context and any other context that is subsequently established.
- *Trust management*. We have implemented global ACLs and the private and readOnly tag for restricted trust. Local ACLs require a more complex implementation and are not included in the first version.

In short, the Mozilla implementation of most of these concepts was rather straightforward. The ease of implementation is due in part to the fact the Mozilla code is well organized and structured, and proved to be very amenable to these modifications.

CONCLUSION

Our security design and implementation for scripting languages can help prevent successful Web attacks on a user's security and privacy. The design uses well-known tools in access control, such as ACL and information hiding capability, that also appear in ongoing research to design a more flexible security architecture for Java (for example, see Gong[5] and Wallach et al.[11]). These tools originate in the development of secure operating systems and programming languages. ■

ACKNOWLEDGMENTS

Murali Rangarajan did a great job on an initial implementation of our design. We thank Norris Boyd of Netscape Communications Corporation for helpful discussions and insight as well as assistance during our implementation efforts.

REFERENCES

1. D. Flanagan, *JavaScript: The Definitive Guide*, O'Reilly and Assoc., Sebastopol, Calif., 1997.
2. P. Lomax, *Learning VBScript*, O'Reilly and Assoc., Sebastopol, Calif., 1997.

3. J.R. LoVerso, *JavaScript Security Flaws*, Open Software Foundation; available online at http://www.osf.org/ loverso/javascript/.
4. V. Anupam and A. Mayer, "Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies," *Proc. Seventh Usenix Security Symp.*, Usenix Assn., Berkeley, Calif., Jan. 1998, pp. 187-200.
5. L. Gong et al., "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2," *Proc. Usenix Symp. on Internet Technologies and Systems* (USITS), Usenix Assn., Berkeley, Calif., Dec. 1997, pp. 103-112.
6. J. Levy et al., "The Safe-Tcl Security Model," *Proc. 1998 Usenix Ann. Tech. Conf.*, Usenix Assn., Berkeley, Calif., 1998, pp. 271-282.
7. R.Y. Kain and C.E. Landwehr, "On Access Checking in Capability-Based Systems," *IEEE Trans. Software Engineering*, Vol. SE-13, No. 2, Feb. 1987, pp. 202-207.
8. L. Gong, "On Security in Capability-Based Systems," *ACM Operating Systems Review*, Vol. 23, No. 2, Apr. 1989, pp. 56-60.
9. D. Volpano, G. Smith, and C. Irvine, "A Sound Type System for Secure Flow Analysis," *J. Computer Security*, Vol. 4, No. 3, 1996, pp. 167-187.
10. D. Kristol et al., *Implementing Secure JavaScript in Mozilla*, available online at http://www.bell-labs.com/user/alain.
11. D. Wallach et al., "Extensible Security Architectures for Java," Tech. Report TR-546-97, Princeton Univ., Dept. of Computer Science, New Jersey, Apr. 1997.

Vinod Anupam is a member of the Database Systems Research Department in the Systems and Software Research Center of Bell Labs—Lucent Technologies. His research interests include collaborative computing, Internet and Web security, electronic commerce, graphics and visualization, and mobile computing. He received a PhD in computer science from Purdue University in 1994.

Alain Mayer is a member of the Secure Systems Research Department at Bell Labs-Lucent Technologies. He joined Bell Labs in September 1996 from System Management Arts (Smarts), a network management start-up company. His research interests include electronic commerce, network and Web security, cryptography, privacy, and network management. He received his PhD in computer science in 1995 from Columbia University.

Readers may contact Anupam and Mayer at Bell Laboratories-Lucent Technologies, 600 Mountain Avenue, Murray Hill, NJ 07974 : e-mail {anupam,alain}@research.bell-labs.com.