Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies

Vinod Anupam and Alain Mayer
Bell Laboratories, Lucent Technologies

Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies

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

Abstract

While conducting a security analysis of JavaScript and VBScript, the most popular scripting languages on the Web, we found some serious flaws. Motivated by this outcome, we propose steps towards a sound definition and design of a security framework for scripting languages on the Web. We show that if such a security framework had been integrated into the respective scripting languages from the very beginning, the probability of preventing the multiple security flaws, that we and other research groups identified, would have been greatly increased.

1 Introduction

JavaScript and VBScript are popular scripting languages used for Web-page design. A user accessing a JavaScript/VBScript enhanced Web-page causes scripts to be downloaded onto the user's machine and to be executed by the interpreter of the user's browser. Scripts typically cannot (directly) access the user's file-system or the network. This is probably the reason that, in contrast to the Java programming language, no formal security model and hence no explicit rules were ever documented on what is and what is not allowed by scripts. A string of serious security flaws discovered by several research groups, including successful attacks on patches issued to fix original flaws, shows that this is a dangerous omission. In particular, we found flaws which allow private data supplied by a user (e.g., credit card numbers, passwords, e-mail address, etc) in a Web transaction to be captured by an attacker. Such an intrusion works even when a user employs encryption (e.g., SSL), since the data is captured either before it is encrypted or after it is decrypted. In contrast to some of the security flaws found in Java (see [MF97]), the vulnerability we discovered does not lead to full system penetration where an attacker can access a user's resources (files, processes) at will. It might thus be argued that such flaws are less serious. However, security and privacy concerns (see, e.g., cover story of *Time Magazine*, dated 8/15/97) have been the single most important barrier to electronic commerce achieving its multi-billion dollar potential. In this light, attacks on a user's security and privacy, are a matter of serious concern.

Motivated by the above considerations, we proceed to show necessary steps towards a general security framework for scripting languages on the Web. We put forward the notion of a *safe interpreter*. A safe interpreter must assure:

- *Data Security.* Data provided by the user (possibly encrypted before it is transmitted) can only be accessed by the intended recipient; the possibility that credit-card data, transaction details, etc. may be obtained by someone else is highly damaging.

- *User Privacy.* Information about the user should not be given out unless explicitly allowed by the user; this protects against unwanted tracking, identification dossiers, junk e-mail, surreptitious file uploads, etc.

As a concrete example, we give excerpts from *Secure JS*, our proposal for a more secure version of JavaScript. In this paper, our treatment covers security issues of scripting languages up to Navigator 4.* and IE 4.*. However, we do not address code signing, a topic that transcends scripting languages. Also, today's browser environment allows embedded scripts to communicate with entities like applets and plug-ins, outside the scripting languages proper, adding a lot to the scope of what scripts can do via these entities. This "composition" of different

entities with different security policies and frameworks is not well understood; other safe scripting environments, such as Safe-Tcl (see [B94, OLW96]), do not allow such composition. We can only touch upon this issue; a thorough treatment is beyond the scope of this paper.
The notion of a secure operating system, which provides safe containers for different Web technologies (Java, plug-ins, JavaScript) would help in achieving the above goals. However, we emphasize that most of the uncovered vulnerabilities of scripting languages originate within the language itself. For instance, a rogue site may contain attack scripts which can access (and send back) data obtained from other documents without any penetration of the underlying operating system or concurrent application.
Finally, we show that if our framework had been designed and integrated when either JavaScript or VBScript were conceived, the probability of preventing the string of security flaws, that were identified by us and other research groups, would have been greatly increased. This motivates the need for future designs of scripting languages to explicitly consider security aspects during initial design. However, security is never absolute. Implementation errors, even in a sound security design, are not unlikely and can be exploited by an attacker. Such flaws, however, are harder to identify by an attacker.

Organization of the Paper: In Section 2 we review the basic concepts of browser scripting, and briefly introduce JavaScript and VBScript, the two most popular scripting languages for Web browsers. Section 3 presents the essence of our attacks on JavaScript and VBScript capable browsers. We then propose, in Section 4, a security framework for scripting languages on the Web. We illustrate this framework with concrete notions and examples taken from our proposal of *Secure JS*. In Section 5, we discuss our attack in more detail and point out how our framework would have helped in preventing it. Section 6 concludes. In the Appendix, we discuss attacks designed by other groups and again show how our framework would have helped in preventing them.

2 Browser Scripting Languages: An Overview

JavaScript (see [F97, KK97]) is a simple procedural language that is interpreted by Web browsers from Netscape Corp. (*JScript*, Microsoft Corp.'s implementation, is a clone that is interpreted in Microsoft's Web browsers. In the rest of this paper, we use JavaScript to refer to both strains.) 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. E.g., 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. This defines a unique *name space* for each HTML page and thus for each collection of scripts embedded in that page. Variable data types are not declared, i.e., *loose typing.* Object references are checked at runtime, i.e., *dynamic binding.*

VBScript (see, e.g., [L97]) is an application scripting language that looks a lot like Visual Basic. It is loosely typed and object based. It can be used for scripting Microsoft's browser, Internet Explorer, with which it communicates using ActiveX Scripting. ActiveX Scripting allows host applications, like browsers, to compile and execute scripts, and manage the name-space exposed to the script. The ActiveX Scripting Object Model (SOM) creates an object instance hierarchy containing, among other things, *window*, *navigator* and *history* objects as described above. Also, the *location* object represents the current URL, and the *document* object reflects the current HTML document. VBScript scripts are embedded in HTML documents, and are interpreted automatically when the document is loaded.

3 Our Attack

While conducting a security analysis of JavaScript we discovered a serious vulnerability in JavaScript-capable browsers. We subsequently analyzed VBScript, and discovered that the vulnerability could be exploited equally effectively using VBScript. The vulnerability in Microsoft browsers was thus also in the ActiveX Scripting Object Model - JScript and

VBScript interpreters are front ends that communicate with the underlying ActiveX SOM objects.

3.1 Overview

On the Web, scripts embedded in multiple browser windows containing documents from the same Web site (same domain name) are allowed to access data in each other, in order to support multi-windowed user interfaces. Our analysis revealed, however, that browser windows could be tricked into trusting attack scripts from rogue sites, thus allowing them to access their data. A rogue site could be set up to track all Web-related activity of visitors even after they had left the site, using a Trojan-horse attack. The tracking provided access to all data typed into forms, including password fields, cookies, and visited URLs. The data was extracted right in the browser, so using a secure encrypted connection to retrieve documents didn't accord the user any extra protection. Likewise for users behind firewalls - data was intercepted while in the browser, and transmitted to the outside rogue site via a proxy server.

3.2 Implications

This browser vulnerability has a serious implication for Web users. Once infected by the Trojan horse, the user's Web interaction is fully exposed to the attacker - every URL retrieved, all data typed into forms - including credit card numbers and passwords, all cookies set by servers accessed etc.
The HTTP protocol supports a facility for authenticating Web users. Many Web-based services however use alternate methods of authorization that provide more flexibility. These methods involve the use of dynamically generated, opaque "session keys" embedded in URLs, in hidden fields of forms or in cookies. The ability of the attack to access such information in an HTML document makes all of these authentication mechanisms susceptible to compromise.
This browser vulnerability also has a serious implication for intranets. Most users use the same browser to access information on the intranet as well as the Internet. A user who has been "attacked" using this vulnerability has essentially compromised the firewall for the duration of the browsing session - the Trojan horse is able to extract data from subsequently loaded intranet documents and transmit it to an external entity. Any data that the user enters into forms - ID numbers, vendors and prices, bug reports, passwords and other proprietary information can be relayed to the outside. In addition, the document itself can be subjected to analysis, yielding information about different fields, options and their corresponding values. Particularly serious is the fact that this exploit bypasses security provided by secure sockets or HTTP authentication, since many Web servers and other business platforms are administered over the Web nowadays. Information about document URLs and links on pages can also provide the attacker with an idea of how the domain is organized. This knowledge may be used to help conduct other attacks. One can envisage a scenario where the Trojan horse actively browses the intranet, possibly under remote control from outside the firewall, trolling for any and all information accessible via the browser.

4 The Notion of a Safe Interpreter

4.1 Execution Environment

We consider a networked system. The basic entities on a single machine in our execution environment are *windows*, *contexts*, *scripts*, *interpreters*, *name-spaces*, and *external interfaces*. A concrete example of a *window* is a browser window. A user can have multiple windows open on her machine at any point in time. A window and its content (e.g., the current HTML document displayed) define a *context*. When the content of a window changes (e.g., as the result of loading a new HTML document), the window's context changes as well. A *script* is a code fragment in a scripting language embedded in the content of a window. A document may have multiple code fragments embedded in it. All of them share the window's context. The content of a window (and hence any embedded script) is typically downloaded from a Web server across the network by the browser. An embedded script is executed within the window's context by an *interpreter*. A *name-space* is the hierarchy of objects accessible to a script executing within the window's context. A script might 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, send e-mail etc. Finally, a script can 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 the scripts executing in w's and w''s context can access each other's name-space and scripts in w can access some external interface, giving them additional capabilities.

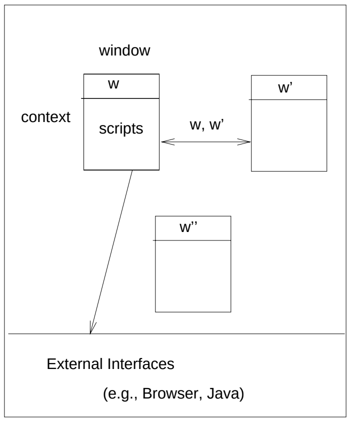

Figure 1: Execution Environment

In some instances, a browser window may contain a framed document with multiple *frames*, which are subwindows containing other documents. From the perspective of a safe interpreter, each frame in a framed document is an independent *window*.

4.2 Safe Interpreter

A script loaded into a window in the above execution environment originates at an arbitrary machine on the network, and is therefore an *untrusted* entity on the user's machine. Hence, a safe interpreter has the task of isolating scripts from executing any unsafe commands (those that could result in security compromises if misused), thus implementing what is called a *padded cell* in [OLW96]. The interpreter has to implement access control with respect to objects within the script's own context. Objects containing browser or window data for example, should only be read-accessible. A safe interpreter has to isolate contexts from each other: a user might decide to provide some information in one context to be sent back to a specific machine - this information should not be accessible to any other machine on the network, and hence must be inaccessible to a script in a different context. On the other hand, under certain circumstances, scripts in different contexts require mutual access, e.g., an application on the user's machine which runs in multiple windows requires the ability to access data in different windows. A safe interpreter must allow this kind of access to scripts in trusted contexts

In summary, a safe interpreter has to implement *access control*, *independence of contexts*, and *management of trust* among different contexts. Provision for these components does not realize a particular security policy. Rather, it gives a *framework* in which a variety of security policies can be easily implemented. We discuss these components in the next few sections.

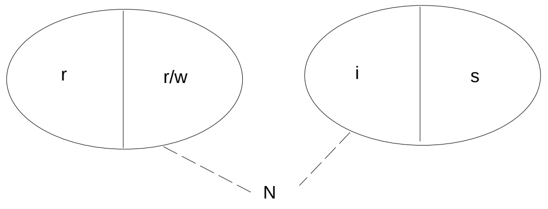

Figure 2: Partitioning of the Name-space

4.3 Access Control

The first task of a safe interpreter is to clearly specify what objects are accessible to an arbitrary context - i.e. what constitutes its name-space. A safe interpreter (1) defines the name-space N_C of a context C, i.e., which objects exist at the time C is activated, (2) the initial values of these objects, and (3) ensures and implements the following partitioning of N_C:

- N_C^r: Items in the object-instance hierarchy readable by a script in C.
- N_C^w: Items in the object-instance hierarchy readable and writable by a script in C.
- N_C^s: Items created by a script in C.
- N_C^i: Items created by the interpreter in C.

We use the term *item* to describe either an object, function, variable, or an object property. Functions do not need to be treated separately in this framework - they are executable if they are readable. Figure 2 shows the following invariants: N_C^r and N_C^w are disjoint, N_C^s and N_C^i are disjoint. Also $(N_C^r \cup N_C^w) = (N_C^s \cup N_C^i) = N_C$. N_C^i is special in the sense that insertion and deletion of elements into N_C^i occurs only when the interpreter creates and deletes a context - scripts in an active context cannot directly add and remove items from this set. Initial values of some of the items in N_C^i depend on the window, in which the context is loaded and activated - the execution environment. It is the role of the safe interpreter to assure correct initialization.

Design Issue 1 for Secure JS: Defining the Name-space The first design issue is to exactly define $N_C = (N_C^s \cup N_C^i)$. The client-side JavaScript environment is defined by its *object instance hierarchy* - objects have *properties* which may be other objects. N_C^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 by a script after the document is already loaded belongs to N_C^s. The `window` object is the root of the JavaScript object instance hierarchy. It has a number of properties. For example, `window.name` is a string that contains the browser-window's name. The `window` object also contains other properties such as the `window.document` object that contains HTML elements reflected from the current document, the `window.navigator` object that encapsulates properties of the browser software, the `window.history` object that represents the browsing history in that window etc. All of these properties are created by the interpreter when a context is loaded and hence belong to N_C^i. 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 `foo` of `window` is created by a script after the context has been loaded. Consequently, `foo` belongs to N_C^s.

Explicit specification of what belongs to the name-space regulates what can ever directly be accessed by a script. A more fine-grained specification of what belongs to N_C^i depends on the policy for *User Privacy*. For example, `document.referrer` contains the URL of the document from which the user reached the current document. Some of the properties of `window.navigator` (e.g. *userAgent*) contain information about the user's operating system. Many privacy-conscious users employ privacy proxies, such as the Anonymizer (see [Anon]) or LPWA (see [LPWA]) to filter this type of information from their HTTP requests. Hence, we recommend against the inclusion of these items in N_C^i (and hence in N_C). The `window.history` object contains an array of strings that specify URLs that have been previously visited by the user in that browser window. In all current versions of JavaScript, this array is **no longer** in N_C^i by default, to protect against unauthorized access to that information by scripts. Note that N_C^s contains objects created directly by scripts. From the standpoint of security and privacy, these objects do not need to be protected from the scripts that created them. This semantically splits the name-space into two disjoint subsets, one that contains objects to which scripts have largely unregulated access (N_C^s), and one that contains objects that can be accessed and manipulated only in specific ways (N_C^i.) □

Design Issue 2 for Secure JS: Read-only vs Writable Objects We now have to specify the partitions N_C^r and N_C^w. Semantically, certain properties of the name-space are unmodifiable, e.g., `document.lastModified` specifies the time at which the currently loaded document was last modified; `document.URL` contains a string that represents the URL from which the document was retrieved. Such objects are trivially in N_C^r, the read-only subset of the name-space. Other properties, such as `document.forms.length` etc., that are reflected from the loaded HTML document should not be directly modifiable by scripts, and hence belong in N_C^r. Object properties, like `document.domain`, are used for trust management (see Section 4.5) and should be in N_C^r. The `value` property of most form elements is intended to be modifiable by scripts, and thus belongs to N_C^w. However, the `value` property of a `FileUpload` element contains a user specified local filename to be transmitted over the network as part of the form submission. To disallow scripts from setting this property and retrieving arbitrary local files, this object rightly belongs in N_C^r. All objects created directly by a script are writable, and hence in N_C^w. One method of unauthorized activity involves a script covertly passing information to another script. This necessarily involves one script writing a value into an object that is subsequently read by another script. By treating writable objects (in N_C^w) more strictly, the interpreter can protect against establishment of such covert channels. □

We have seen how a safe interpreter assures that N_C is always disjoint from any private user resources on the machine, such as the file system, process data, sockets, etc. This is much like the concept of a padded cell in Safe-Tcl ([OLW96]). However, access to such resources is very useful. In Safe-Tcl, such accesses are regulated via a *master interpreter*, whose methods can be invoked by the safe interpreter. Hence, such access is regulated within the language itself. Unfortunately, for scripting languages on the Web, no such master interpreter is available. *External interfaces* give a script capabilities to invoke browser functionality, or methods of Java applets and ActiveX scripts. Each such invocation across interfaces needs to be examined by the interpreter, which has the option of (1) allowing the call to proceed, (2) aborting the call, or (3) asking the user's explicit permission for the call to proceed. However, the resulting overall security policy depends both on the policy of the safe interpreter

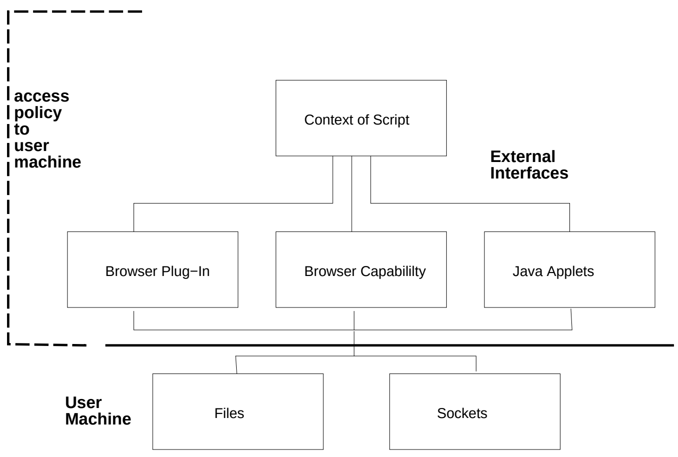

Figure 3: External Interfaces

and the policy of the external interface, the latter being beyond the control of the safe interpreter.

For example, Figure 3 presents a situation, for JavaScript, in which scripts can access the browser, Java applets, and browser plug-ins through external interfaces. As a result, a script can (albeit indirectly) access the file system or the network, making HTPP, FTP, and SMTP requests via the browser. Hence, the overall access security policy with respect to these resources depends, besides the safe interpreter, on the policies for the browser, for applets, and plug-ins. For example, if a new method is introduced for applets to access a file and the safe interpreter remains unchanged, then a script invoking this new method might not be subjected to an appropriate check by the safe interpreter. What is really needed, is sound semantics for *composition* of security policies. This is an area hardly investigated and understood at the time of this writing and is beyond the scope of this paper. A safe interpreter therefore should minimize the number of accepted external interfaces and their methods offered to scripts. Furthermore, it should adopt a conservative policy for every invocation of a call across an external interface, as well as external accesses to the script's name-space. A secure operating system providing safe containers and interfaces for different applications (Java, plug-ins, etc) is an alternative.

Design Issue 3 for Secure JS: Access to External Interfaces In order to be useful, JavaScript needs, at the very least, an interface to browser capabilities to access the network and special files. User supplied data (in forms) need to be transmitted back to an origin server. A script chooses the origin-server (URL) via `form.action` and the SUBMIT method via `form.method`, and submits the user data via `form.submit`. Whenever a protocol specified in the URL of the origin server is **not** HTTP, the safe interpreter should get a user's approval. SMTP and FTP requests potentially reveal a user's e-mail address. Furthermore, a user might be browsing via a privacy HTTP-proxy and thus unwittingly lose this protection when being switched to another protocol. The safe interpreter must prevent a script from directly accessing local files of a user. This includes any browser related files, such as bookmarks, caches, history, etc. A general treatment of external interfaces (with Java, plug-ins etc.) is beyond the scope of this paper - it requires an understanding of the very general issue of composition of security policies. □

4.4 Independence of Contexts

To ensure independence of different contexts, the following restrictions have to be enforced by a safe interpreter:

- For each active context C, N_C^w and N_C^s must be disjoint from $N_{C'}$ of each other context C'.

If an active context C is terminated and a new context C' is loaded and activated in a window w, a safe interpreter has to do the following operation in N_C in order to transform it into the initial name space $N_{C'}$ such that the name space $N_{C'}$ does not depend in any way on C (see also Figure 4):

- *Remove and rebuild* N_C^i*:* The set (instance hierarchy) is deleted and rebuilt according to the newly loaded HTML document. This process must ensure that while rebuilding, each item in N_C^w is set to a neutral value (null).
- *Delete* N_C^s*:* remove all its items.
- *Invalidate all references to* N_C within other contexts, which have been established by a trust relationship (see subsequent section).

Design Issue 4 for Secure JS: Independent Browser Windows A safe interpreter must ensure that the memory of the object hierarchy that corresponds to N_C^w is disjoint for different windows. This includes objects like the `window.navigator` object, which might represent the same browser instance for all windows. Enforcing disjointness protects against the establishment of covert channels between scripts. As described earlier, one method of unauthorized script activity involves a script covertly passing information to another script by

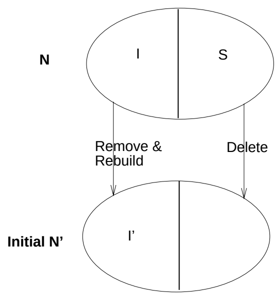

Figure 4: Independence of contexts loaded into same window

writing a value into an object that is subsequently read by the other script. This is particularly relevant for items whose value persists across document retrievals, e.g., some properties of the `window` object. By treating writable objects (in N_C^w) more strictly, the interpreter can protect against such covert channels. Secure JS does not allow values of items in N_C^w to persist when the current document is unloaded. For example, `window.name` is a string that specifies the name of the browser window. If scripts write a value into this property of the window, then any changes will disappear when the current document in the window gets unloaded. □

Design Issue 5 for Secure JS: Garbage Collection From the above 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. Whenever a document is unloaded, the safe interpreter has to traverse the entire object hierarchy: Each item in N_C^s must be deleted. E.g., the code fragment `var i; i = 0` creates an item `window.i`, which must be removed. Appropriate items in N_C^i are deleted. Each remaining item in N_C^w must be set to a neutral value, even if it is above the `document` object in the hierarchy. E.g., `window.status` and `window.name`. All references contained in other windows that point to objects in the current window must be invalidated (see also next section), even if the current window remains open (and thus `window.closed` would be false, if the window reference remained valid). □

4.5 Management of Trust

While most contexts are typically independent, certain applications greatly benefit from establishing mutual trust relationships among contexts in different concurrently active windows. This allows, for example, the coordination of information in multiple windows when presented to a user. In order to maintain a secure system, we need to specify (1) when a script in context C is allowed to establish a relationship with context C' and (2) the subsequent privileges for $N_{C'}$ issued to scripts in C and vice versa.

If context C' (in window w') trusts C (in window w), then access to C''s name-space is allowed. 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 $N_{C'}^r$, read and write all items in $N_{C'}^w$ and insert items into $N_{C'}^s$. Note that the specification in Section 4.4 implies that if a script in C inserts an object into $N_{C'}^s$, this object will only be accessible to C as long as C' is active in w'. After that, C's reference to C' is set to null. We can extend this approach to different trust relationships of various degrees: (1) Read-only access to C''s name-space. (2) Access only to objects in C''s name-space which are not declared *private* by C'. The second approach requires that each object have a property `private` which, when set, makes the object invisible to external scripts.

Global Access Control List. A script in context C requests to establish a trust relationship via either a *open()* or *connect()* function call to its interpreter. The safe interpreter then has to decide if such a relationship is permissible. This decision depends on the *access control policy*. We make use of *access control lists (ACL)*. ACLs are a well-known tools for implementing secure operating and file systems. In our model, an ACL is a set of contexts. For example, an ACL can be a set of URLs, where each URL represents a window's current content. We associate with each context C a single *access control list (ACL)*, which indicates the set of contexts which have the privilege to obtain access to N_C. The ACL is part of N_C and its value is loaded whenever C is loaded into a window. The ACL should only reside in N_C^r; otherwise, a script, possibly from a different context, can change the ACL.
Let us now define the *open()* and *connect()* functions:

- A script in C issues *open(C', w')*, where w' is a new (i.e., not currently open) window: If C is in C's ACL, then a new window w' is created and context C' is activated in w'; a trust relationship is established.
- A script in C issues *open(C', w')*, where w' is an open window with an active context C'' (possibly $C'' = C'$): If C is in C's ACL, then C''

is terminated and C' is activated; a trust relationship is established.

- A script in C issues *connect(C', w')*: If C' is already active in window w', a trust relationship is established; otherwise it is handled like an *open()* call.

Design Issue 6 for Secure JS: Global Access Control We introduce the property `document.ACL`, the document's access control list to be a member of N_C^r and N_C^i. In Secure JS, `window.open` implements the *open()* function described above. The required inputs are a *URL* and a *name*. The safe interpreter checks that `document.domain` in the context of the calling script is included in `document.ACL` of the context (defined via the *URL*) to be loaded. If this check succeeds, then the interpreter proceeds as follows: If *name* matches an open window w', then the current document (context) in w' is unloaded and the new context (*URL*) is loaded. If there is no match with an open window, then the interpreter opens a new window w' with *name* and loads the the new context (*URL*). If the access control check is positive, `window.open` returns a reference to the context in w'. The safe interpreter also checks if `document.domain` of w' is included in the calling script's `document.ACL`. If this check is positive, then a side effect of the this function call is that `w'.opener` provides w' with a reference the calling script's context. We also introduce a new method `window.connect` with the same arguments as `window.open`. This method differs from `window.open` only in the case where a window with name *name* and context *URL* already exists. In that case `connect` simply provides the two contexts with references to each other. Lastly, we introduce the new object property `private`. For Secure JS, each object in the object hierarchy has this additional property. If this property is set to `true`, the corresponding object is invisible to external scripts. After executing `victim = window.open("www.vendor.com", "spy")`, (and passing the ACL check) a script might try `snoop = victim.document.forms[0].elements[0].value`, assuming that in the `victim`-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 above statement. □

Local Access Control List. While the above approach implements a reasonable trust security policy (not unlike a UNIX file system enhanced with ACLs), it allows that "the right to exercise access carries with it the right to grant access" (as noted in the context of "unmodified" capability systems in, e.g., [B84, G89a, KL87]). For example, if a script in context C' is allowed to read (and hence copy) an object in context C, then every script which is 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 to C. Many extensions to capability systems have been proposed to address this concern (see, e.g., [G89b, KH84, KL87]). In our environment, the following simple refinement of the above access list based control is sufficient: Besides a context-global ACL, each object in this context is associated with its own, local ACL (the same as the global ACL when the context is loaded). In the realm of secure operating systems, the data plus its data security attributes is called a *segment* (see, e.g., [KL87]). When an object gets copied (from one context to another), the segment remains intact, i.e., the safe interpreter copies the ACL as well. The *open* and *connect* routines work as described in the first solution; however, if a relationship is established, scripts in N_C get a reference only to those items in $N_{C'}$, for which C is in their local ACL. Hence, a script's context now plays the role of its *capability* (see [KL87]), which is checked against the ACL of the segment of the item made accessible. A local ACL (of a segment) 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 policy, bugs in context C' no longer allow access of third parties to context C.

A somewhat difficult aspect of the scheme described above, is to unequivocally define the semantics of "copying" an object, which guides when the object's ACL has to be updated. Assignments are straightforward. Many scripting language allow control flow constructs, such as *if ... then ... else* or *while ... do*. If a conservative policy is assumed, then assignments made in the scope of a control flow construct should be considered to be "copies" of values read in the control part. This is similar to, for example, the *tainting facility* of PERL (see [WCS96]).

Design Issue 7 for Secure JS: Local Access Control If Secure JS is used with the option of local access control, then every object in the JavaScript object hierarchy has a property `ACL`, its local access control list. Once again, `ACL` property belongs in N^r_C. When a context is loaded and items in N^i_C are created, the value of their respective `ACL` property is set to the value of `window.document.ACL`. Items created by scripts (and hence in N^s_C) 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. A side effect of this code is the assignment `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`, permitting the operation only if the check succeeds. □

4.6 How the pieces fit together

In this section, we briefly show how the concepts of access control, independence of contexts, and trust management support data security and user privacy.
User Privacy: Almost all data on a user's machine which is not directly related to the current HTML document should be considered private to a user. *Access control* provides a padded cell for scripts, which assures that scripts cannot access the file system, process data and other unsafe resources. By maintaining a clear separation of data outside a name-space, read-only items, and writable items within the name-space, access control assures that scripts only have access to those parts of browser and window related data, which do not compromise a user's privacy while browsing. We have also described the *external interface* to a browser. Furthermore, *independence of contexts* assures that there are no "hidden channels" among different contexts. For example, if a writable item persisted across changes of context (as it is currently the case in JavaScript), it could be used as a user-invisible (albeit non-persistent) 'cookie' accessible to collaborating Web-sites.
Data Security: Data provided by a user in a context C of a window w (e.g., 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. This is assured by *trust management. Independence of contexts* assures that any context C' activated after C in w cannot access any data written in C. Furthermore, if C'' had a reference to a context C' that was loaded in w before C, this reference was set to null at the time C' was unloaded; assured 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 the safe interpreter.

5 Security Analysis

In this section, we discuss scripting languages vulnerabilities exposed by different attacks, highlighting how they could have been prevented using a safe interpreter. We discuss other variants of this attack, as well as other problems with JavaScript in the Appendix.

5.1 Bell Labs Attack

Our security analysis of scripting languages on browsers from Netscape Corp. and Microsoft Corp. that are currently in use revealed that browser windows could be tricked into trusting attack scripts from rogue sites, thus allowing them to access their data (and hence the name-space of embedded scripts). See [CERT97, NN-Bug97, MSIE-Bug97]. Using this vulnerability, a rogue site could be set up to track all Web-related activity of visitors even after they had left the site, using a Trojan-horse attack. The tracking provided access to all data typed into forms (e.g., password fields), to cookies, and to visited URLs.
There were three requirements for the Trojan horse to successfully snoop on user activity. It needed to persist across multiple document fetches (script context changes), it needed to be able to access data from other loaded documents (across windows), and it needed to be able to transmit captured data to a desired location. A 'safe interpreter', properly implemented, would have safeguarded against all those attack components.

5.2 Initialization

Initially, the user loads an HTML document (context C) into a browser window w, which (unknown to the user) contains some adversarial script. This script first sets itself up to exist independently of the original window w, so it can persist across multiple document loads in w. To do this, it creates a new window (subsequently called the snooping window) w' using a *window.open()* call, and loads a different HTML document (context C') that contains the attack script into w'. Though a vigilant user could notice the presence of the new window, the creation of a new window is not a suspicious act in itself- many sites use multiple windows to display documents on the Web. In addition, control over size, placement and stacking of windows can be used effectively to reduce the chances of detection.

5.3 Data Extraction

The attack script in w' next gets a handle (object reference) to window w. In Netscape Navigator 2.*, this is done by setting a property of the snooping window w' when it was created by the adversarial script in window w. In all other browser versions, the *window.opener* property of the JavaScript object *window* gives the attack script in window w' the desired reference to w. We discuss more specifics of the vulnerability and the attack in the context of different browsers, highlighting the deficiencies that were exploited.

Netscape Navigator

Navigator 2.*, 3.*, and 4.* all use various means of trust management: (1) document address based access control - only a script from the HTTP server (domain name) that sent the HTML document is able to access data in a document, (2) a Perl-like 'tainting' model ([WCS96, F97, KK97]), and (3) a digital signature based scheme for access control [KK97], in which only signed scripts with the appropriate rights are able to access data in documents from other domains.

The attack script (in window w') bypassed security in all cases by using the window reference to insert arbitrary scripts into the observed window w's context in the form of JavaScript URLs (These are URLs of the form *javascript:<JavaScript code>*.) We discovered that code in these JavaScript URLs was not subject to access control and persisted across document reloads (context changes) in w. The inserted scripts periodically gathered all information available in the current context of the observed window w, and made it available to the snooping window w'. Data was stolen by directly writing into properties of the snooping window document from the observed window, in the case of Navigator 2.* and 3.*. In the case of Navigator 4.*, data was extracted by writing to and reading from a property of a mutually accessible object. Hence, the contexts of these two windows (w and w') were not disjoint!

Microsoft Internet Explorer

Microsoft Internet Explorer 3.* did not impose any security restrictions that needed to be bypassed. Starting with the window reference of the observed window w, attack scripts in the snooping window w' were free to walk the instance object hierarchy of the observed window w, and had direct access to all the information in any documents subsequently loaded into the observed window. The attack scripts periodically polled the current context C of the observed window w for its contents, copied the data, and then transmitted the captured data to a remote server.

We subsequently discovered that the same effect could be achieved using VBScript. This variant of the attack scripts used VBScript to set up a snooping window w', which was subsequently able to walk the instance hierarchy of the observed window w, and capture and transmit all the information.

The beta version of Microsoft IE 4.0 also has a security model centered around script address based access control. Once again, the exploit bypassed security by using JavaScript URLs to inject and execute arbitrary JavaScript code in the observed window, and to move captured data from the observed window to the snooping window.

5.4 Data Transmission

Once data was obtained, scripts in the snooping window w' were free to transmit it to a location of their choosing. They could transmit directly via HTTP by sending the data to a dynamically constructed URL. This could not be detected by the user. The snooping window could put the data into fields of a form and automatically submit the form. This method could be detected and circumvented by a vigilant user who had set up the browser to require interactive confirmation of all form submissions. Finally, code in the snooping window could communicate with an installed Java applet and/or ActiveX script to transmit the information to a desired location. This also could not be detected by the user.

5.5 Role of a Safe Interpreter in the Bell Labs Attack

The safe interpreter described in Section 4 would have protected against this style of attacks. When the attack was set up, context C in w, the observed window, and context C' in w', the snooping window, trusted each other, since they both originated on the attacker's Web-site. Proper trust management would have terminated the trust relationship when C was unloaded from w - trust doesn't persist across unloading of HTML documents (context changes.) The contexts would have been prevented from writing into each other. Enforcement of independence of contexts by the safe interpreter would subsequently have disallowed any accesses from C' to C (and vice versa) via the reference to C, rendering it useless for tracking. Finally, a regulated external interface could have been used to implement policies that alerted the user to potentially suspicious activity when the safe interpreter detected that an attempt was made to send data from a different site to the attacker's machine.

6 Conclusions

We have shown how taking steps towards a careful security design for scripting languages can help in preventing successful attacks on a user's security and privacy on the Web. It is not surprising that well known notions in access control, such as "ACL" and "capability" (see, e.g., [B84, G89a, G89b, KL87]) reappear, not only in our context, but also in ongoing research on how to design a more flexible Java (downloaded executable content) security architecture (see, e.g., [JRP96, G97, WBDF97]). It demonstrates that for a sound security model for downloadable, executable content on the Web, there are no shortcuts to a careful design based on notions and algorithms developed for secure operating systems.

References

[Anon] Anonymizer, http://www.anonymizer.com.

[B84] W. E. Boebert, *On the Inability of an Unmodified Capability Machine to Enforce the *-Property,* 7th DoD/NBS Computer Security Conference, 1984.

[B94] N. Borenstein, *Email with a mind of its own: The Safe-Tcl language for enabled mail*, IFIP Conference on Upper Layer Protocols, Architectures, and Applications, 1994.

[B97] D. Brumleve, *Tracker Privacy Bug*, Aleph2 Software, July 1997, http://www.aleph2.com/tracker/tracker.cgi.

[C97] K. Chiang, *Singapore Privacy Bug*, ITI, Singapore, July 1997, http://www.iti.gov.sg/iti_people/iti_staff/kcchiang/bug/.

[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.

[G89a] L. Gong, *On Security in Capability-Based Systems*, ACM Operating Systems Review, 1989.

[G89b] L. Gong, *A Secure Identity-Based Capability System*, IEEE Symposium on Security and Privacy, 1989.

[G97] L. Gong, *New Security Architectural Directions for Java*, IEEE COMPCON, 1997.

[JRP96] T. Jaeger, A. Rubin, A. Prakash, *Building systems that flexibly control downloaded executable content*, 6th USENIX Security Symposium, 1996.

[KH84] P. A. Karger, A. J. Herbert, *An augmented capability architecture to support lattice security and traceability of access*, Proc. 1984 IEEE Symp. on Security and Privacy.

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

[KL87] R. Y. Kain and C. E. Landwehr, *On Access Checking in Capability-Based Systems.* IEEE Transactions on Software Engineering, Vol **SE-13**, No **2**, February 1987.

[L96] J. R. LoVerso, *JavaScript Security Flaws*, OSF, http://www.osf.org/ loverso/javascript/.

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

[LPWA] Lucent Personalized Web Assistant, http://lpwa.com:8000.

[MF97] G. McGraw and E. Felten, *Java Security: Hostile Applets, Holes, and Antidotes.* Wiley Computer Publishing, 1997.

[MSIE-Bug97] *Bell Labs Privacy Problem*, Microsoft Corp., July 1997, http://www.microsoft.com/ie/security/bell.htm.

[NN-Bug97] *Bell Labs Privacy Bug*, Netscape Corp., July 1997, http://www.netscape.com/assist/security/resources/bell_labs_privacy.html.

[OLW96] J. Ousterhout, J. Levy, B. Welch, *The Safe-Tcl Security Model*, Manuscript.

[S97] A. dos Santos, Santa Barbara Privacy Bug, *UCSB Secure Internet Programming Team*, August 1997, http://www.cs.ucsb.edu/ andre/Attack.html.

[WBDF97] D. Wallach, D. Balfanz, D. Dean, E. Felten, *Extensible Security Architectures for Java.* Princeton CS, TR-546-97, April 1997

[WCS96] L. Wall, T. Christiansen, R. Schwartz, *Programming Perl.* O'Reilly & Associates, Inc., September 1996.

Appendix: Other Attacks

In this section we describe some of the other browser scripting language based attacks that were reported.

The Tracker attack

An attack that behaved similar to the Bell Labs attack, and worked specifically in Netscape Navigator 3.0 was devised independently, see [B97], and was equally effective at monitoring user activity on the Web. It worked even for Navigator 3.02, which incorporated a patch for the Bell Labs attack! The *onUnload()* event handler for document objects is a piece of JavaScript code that is executed when the browser unloads the current HTML document before replacing it with a new document. This attack worked by using attack scripts in an attack window w' (context) to dynamically insert a new *onUnload()* handler in the observed browser window w. This newly inserted code would be activated when the current document C was unloaded, and would walk through the object instance hierarchy and surreptitiously transmit all accessible information to a remote server.

A safe interpreter would not allow scripts of an untrusted context C' to insert code into context C, thus disallowing subversion of the *onUnload()* event handler.

Singapore Attack

Another similar attack was devised independently (see [C97]) and worked specifically in Netscape Navigator 4.*. It was even more insidious. This Trojan horse did not use a separate window to persist across multiple document fetches, and was thus harder to detect. The Java VM in browsers allows applet threads to continue to run even after the document that started the applet is unloaded. (A discussion of Java and its security is outside the scope of this document, see, e.g., [MF97]). LiveConnect [F97] is a 'glue' technology from Netscape Corp. that is designed to allow Java applets, browser plugins and JavaScript scripts to communicate in Netscape's browsers. It is manifested as an API that programmers can use to implement desired interaction between these different entities. An applet can communicate with scripts by using LiveConnect Java classes to get a window handle (reference to a JavaScript context). The attack applet obtained a window handle of its own window w, which remained valid even after the HTML document C that initially loaded the applet was unloaded, and another HTML document C' was loaded into the window. The applet could then, via the window handle, walk through the JavaScript object instance hierarchy of the new document, access any contained information, and transmit it to a remote server.
A safe interpreter would invalidate the reference to a context C (including references given to external interfaces) as soon as C was unloaded and thus would prevent the Singapore attack. This attack however, is a reminder that interaction between different entities with different security policies is not well understood and always a potential threat.
This point was further substantiated by the Santa Barbara attack (see [S97]), which produced similar results in Netscape Communicator 4.02. It used a Java applet to access and steal data in HTML documents in an observed window W by tricking it into executing attack JavaScript code.

Other Scripting Attacks

From the start, JavaScript implementations inadvertently allowed malicious scripts to attack a user in different ways [L96].
Early implementations of JavaScript stored the user's browsing history into the *history* object, and

allowed scripts read-access to the whole object. Malicious scripts were free to transmit this information to a server, allowing operators to trivially build dossiers on Web users. Another problem, discovered early on, was the complete absence of access control! This allowed malicious scripts to actively (and easily) track a user's browsing history by monitoring another browser window. A variant of this method allowed scripts to browse directories in a user's machine by starting with a document obtained via a URL using the *file:* protocol - the script could walk through the instance hierarchy of a dynamically generated document that represented the directory and its contents from the local file system, and transmit this information to a remote server. In another instance, the interpreter let scripts from one document (context) stay resident even after the document was unloaded, opening the door to variants of the above attacks.

Browsers began to support the *mailto:* protocol for submitting forms on the Web, which used the E-mail infrastructure for transport, instead of HTTP. Automatic submission of forms by scripts, using the *mailto:* protocol (and hence, the browser as an external interface) allowed Web-site operators to surreptitiously capture the email address of users. File Upload elements in HTML-forms provided a mechanism for Web users to transmit (or upload) files from the local file system to a remote Web server. Malicious scripts, however, were able to specify the absolute path of an arbitrary file, and cause the browser (again as an external interface) to automatically submit the form, transmitting the contents of that file to the remote server.

The fix for these problems typically involved hobbling the responsible feature - scripts were no longer allowed to access the list of URLs in the *history* object; a script-address based access control policy was developed - JavaScript scripts could read properties of a document only if the scripts were loaded from the same server (domain name) as the document; automatic submission of forms using the *mailto:* protocol was disallowed; and File Upload elements in forms could have their value set only interactively, not by a Web-site operator.

An insidious variant of the File Upload attack was later discovered in Netscape browsers. It exploited an obscure bug in the implementation of JavaScript that allowed an attack script to assign an arbitrary file name to a File Upload element of a form. (An implementation of this attack violated a name-space rule of the safe interpreter by adding elements to N^i_C, the interpreter-created object set, after the context C was already activated!) The specified file would be transmitted to the Web server when the form was submitted, either automatically by JavaScript or as a result of a user-initiated submission.

The recently reported French Privacy Bug in Netscape Communicator 4.02 allowed a malicious script to read and transmit a user's local browser preferences file if the script could successfully guess it's location on the local file-system. The Freiburg Issue affected Microsoft Internet Explorer 4.0, and allowed a script to read and transmit local text, HTML or image files if the JScript or VBScript script could successfully guess their location on the file-system.

Though most of these flaws were fixed as scripting implementations evolved, a formal specification of a safe interpreter or safe interaction with a browser vis-a-vis scripting has not been developed and publicly released.

Access control in the safe interpreter formally specifies what a script is allowed to access in the user's environment - hence policies on access to *history* information, user information, files etc. can easily be implemented. Regulated external interfaces can be used to implement policies that safeguard against surreptitious capture of information via HTTP, FTP and SMTP, or from the file-system. Finally, the formalism of a safe interpreter enables one to reason about browser scripting at the level of semantics, rather than mechanics of different operations.