JavaScript Security in Communicator 4.*x*

This document describes the security model used by JavaScript in Communicator 4.*x* and provides information on how you can use the new security features to create signed JavaScript scripts.

There are two security policies in JavaScript:

- The *same origin* policy is the default policy. It dates from Navigator 2.0, with necessary coverage fixes in Navigator 2.01 and Navigator 2.02.
- The *signed script* policy is new to Communicator 4.0. This new policy for JavaScript is based upon the new Java security model, called *object signing*. To make use of the new policy in JavaScript, you must use the new Java security classes and then sign your JavaScript scripts.

This document is intended for JavaScript programmers. Its contains the following sections:

Same Origin Policy

The same origin policy is quite simple: When loading a document from one origin, a script loaded from a different origin cannot get or set certain *predefined* properties of certain browser and HTML objects in a window or frame. (Those properties are listed in Table 2.)

Here, Communicator defines the origin as the substring of a URL that includes `protocol://host` where `host` includes the optional `:port` part. To illustrate, Table 1 gives examples of origin comparisons to the URL `http://company.com/dir/page.html`.

Table 1 Same origin comparisons

URL	Outcome	Reason
`http://company.com/dir2/other.html`	Success	
`http://company.com/dir/inner/another.html`	Success	
`http://www.company.com/dir/other.html`	Failure	Different domains
`file://D|/myPage.htm`	Failure	Different protocols
`http://company.com:80/dir/etc.html`	Failure	Different port

There is one exception to the same origin rule. A script can set the value of `document.domain` to a suffix of the current domain. If it does so, the shorter domain is used for subsequent origin checks. For example, assume a script in the document at `http://www.company.com/dir/other.html` executes this statement:

```
document.domain = "company.com";
```

After execution of that statement, the page would pass the origin check with `http://company.com/dir/page.html`.

Table 2 lists the properties that can be accessed only by scripts that pass the same origin check.

Table 2 Properties subject to origin check

Object	Properties
`image`	`lowsrc`, `src`
`layer`	`src`
`location`	All except `x` and `y`

Table 2 Properties subject to origin check

Object	Properties
`window`	`find`
`document`	For both read and write: `anchors`, `applets`, `cookie`, `domain`, `elements`, `embeds`, `forms`, `lastModified`, `length`, `links`, `referrer`, `title`, `URL`, *`formName`* (for each named form), *`reflectedJavaClass`* (for each Java class reflected into JavaScript using LiveConnect) For write only: all other properties

New Access Errors

To tighten security, some changes have been made to when the origin checks apply.

Named forms

In Navigator 3.0, named forms were not subject to an origin check even though the `document.forms` array was. In Communicator 4.0, named forms are also subject to an origin check. This can cause existing code to break.

You can easily work around the resulting security errors. To do so, create a new variable as a property of the `window` object, setting the named form as the value of the variable. You can then access that variable (and hence the form) through the `window` object.

File: URLs

In Navigator 3.0, when you use `<SCRIPT SRC="...">` to load a JavaScript file, the URL specified in the `SRC` attribute could be any URL type (`file:`, `http:`, and so on), regardless of the URL type of the file that contained the `SCRIPT` tag.

In Communicator 4.0, if you load a document with any URL other than a `file:` URL, and that document itself contains a `<SCRIPT SRC="...">` tag, the internal `SRC` attribute can't refer to another `file:` URL.

To get the 3.0 behavior in 4.0, users can add the following line to their preferences file:

```
user_pref("javascript.allow.file_src_from_non_file", true);
```

However, be cautious with this preference. It opens a security hole. Users shouldn't set this preference to true unless they have some overriding reason for accepting that risk.

Origin Checks and Layers

A layer can have a different origin than the surrounding document. Origin checks are made between documents and scripts in layers from different origins. That is, if a document has one or more layers, JavaScript checks the origins of those layers before they can interact with each other or with the parent document.

For information on layers, see *Dynamic HTML in Netscape Communicator*.[1]

Origin Checks and Java Applets

Your HTML page can contain `APPLET` tags to use Java applets. If an `APPLET` tag has the `MAYSCRIPT` attribute, that applet can use JavaScript. In this situation, the applet is subject to origin checks when calling JavaScript. For this purpose, the origin of the applet is the URL of the document that contains the `APPLET` tag.

Signed Script Policy

The JavaScript security model for signed scripts is based upon the Java security model for signed objects. The scripts you can sign are inline scripts (those that occur within the `SCRIPT` tag), event handlers, JavaScript entities, and separate JavaScript files.

A signed script requests expanded privileges, gaining access to restricted information. It requests these privileges by using LiveConnect and the new Java classes referred to as the Java Capabilities API. These classes add facilities to and refine the control provided by the standard Java `SecurityManager` class. You can use these classes to exercise fine-grained control over activities beyond the “sandbox”—the Java term for the carefully defined limits within which Java code must otherwise operate.

All access-control decisions boil down to who is allowed to do what. In this model, a **principal** represents the “who,” a **target** represents the “what,” and the **privileges** associated with a principal represent the authorization (or denial of authorization) for a principal to access a specific target.

Once you have written the script, you sign it using Netscape’s Page Signer tool. Page Signer associates a digital signature with the scripts on an HTML page. That digital signature is owned by a particular principal (a real-world entity such as Netscape or John

1. http://developer.netscape.com/library/documentation/communicator/dynhtml/index.htm

Smith). A single HTML page can have scripts signed by different principals. The digital signature is placed in a Java Archive (JAR) file. If you sign an inline script, event handler, or JavaScript entity, Page Signer stores only the signature and the identifier for the script in the JAR file. If you sign a JavaScript file with Page Signer, it stores the source in the JAR file as well.

The associated principal allows the user to confirm the validity of the certificate used to sign the script. It also allows the user to ensure that the script hasn't been tampered with since it was signed. The user then can decide whether to grant privileges based on the validated identity of the certificate owner and validated integrity of the script.

You should always keep in mind that a user may deny the privileges requested by your script. You should write your scripts to react gracefully to such decisions.

This document assumes that you are already familiar with the basic principles of object signing, using the Java Capabilities API, and creating digital signatures. The following documents provide information on these subjects:

- *Netscape Object Signing: Establishing Trust for Downloaded Software*[1] provides an overview of object signing. Be sure you understand this material before using signed scripts.
- *Introduction to the Capabilities Classes*[2] gives more details on how to use the Java Capabilities API. Because signed scripts use this API to request privileges, you need to understand this information as well.
- *Java Capabilities API*[3] introduces the Java API used for object signing and provides details on where to find more information about this API.
- *Using Page Signer*[4] describes the signing tool for creating signed JavaScript scripts.
- *Overview of Object-Signing Resources*[5] contains a list of documents and resources that provide information on object signing, from creating the Java applet to getting a certificate to packaging and signing it.
- *JavaScript Guide*[6] contains information about using LiveConnect to access Java classes from a JavaScript script.

1. http://developer.netscape.com/library/documentation/signedobj/trust/index.htm
2. http://developer.netscape.com/library/documentation/signedobj/capabilities/index.html
3. http://developer.netscape.com/library/documentation/signedobj/capsapi.html
4. http://developer.netscape.com/library/documentation/signedobj/pagesign/index.htm
5. http://developer.netscape.com/library/documentation/signedobj/overview.html
6. http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html

Note Navigator 3.0 provided data tainting to provide a means of secure access to specific components on a page. Because signed scripts provide greater security than tainting, tainting has been disabled in Communicator 4.*x*.

SSL Servers and Unsigned Scripts

An alternative to using the Page Signer tool to sign your scripts is to serve them from a secure server. Communicator treats all pages served from an SSL server as if they were signed with the public key of that server. You do not have to sign the individual scripts for this to happen.

If you have an SSL server, this is a much simpler way to get your scripts to act as though they were signed. This is particularly helpful if you dynamically generate scripts on your server and want them to behave as if signed.

For information on setting up a Netscape server as an SSL server, see *Managing Netscape Servers*.[1]

Codebase Principals

As does Java, JavaScript supports codebase principals. A **codebase principal** is a principal derived from the origin of the script rather than from verifying a digital signature of a certificate. Since codebase principals offer weaker security, they are disabled by default in Communicator.

For deployment, your scripts should not rely on codebase principals being enabled. You might want to enable codebase principals when developing your scripts, but you should sign them before delivery.

To enable codebase principals, end users must add the appropriate preference to their Communicator preference file. To do so, add this line to the file:

```
user_pref("signed.applets.codebase_principal_support", true);
```

Even when codebase principals are disabled, Communicator keeps track of codebase principals to use in enforcement of the same origin security policy, described in "Same Origin Policy" on page 1. Unsigned scripts have an associated set of

1. http://developer.netscape.com/library/documentation/enterprise/mngserv/index.htm

principals that contains a single element, the codebase principal for the page containing the script. Signed scripts also have codebase principals in addition to the stronger certificate principals.

With codebase principals enabled, when the user accesses the script, a dialog displays similar to the one displayed with signed scripts. The difference is that this dialog asks the user to grant privileges based on the URL and doesn't provide author verification. It advises the user that the script has not been digitally signed and may have been tampered with.

Note If a page includes signed scripts and codebase scripts, and `signed.applets.codebase_principal_support` is enabled, all of the scripts on that page are treated as though they are unsigned and codebase principals apply.

For more information on codebase principals, see *Introduction to the Capabilities Classes*.

Scripts Signed by Different Principals

JavaScript differs from Java in several important ways that relate to security. Java signs classes and is able to protect internal methods of those classes through the public/private/protected mechanism. Marking a method as protected or private immediately protects it from an attacker. In addition, any class or method marked `final` in Java cannot be extended and so is protected from an attacker.

On the other hand, because JavaScript has no concept of public and private methods, there are no internal methods that could be protected by simply signing a class. In addition, all methods can be changed at runtime, so must be protected at runtime.

In JavaScript you can add new properties to existing objects, or replace existing properties (including methods) at runtime. You cannot do this in Java. So, once again, protection that is automatic in Java must be handled separately in JavaScript.

While the signed script security model for JavaScript is based on the object signing model for Java, these differences in the languages mean that when JavaScript scripts produced by different principals interact, it is much harder to protect the scripts. Because all of the JavaScript code on a single HTML page runs in the same process, different scripts on the same page can change each other's behavior. For example, a script might redefine a function defined by an earlier script on the same page.

To ensure security, the basic assumption of the JavaScript signed script security model is that *mixed scripts on an HTML page operate as if they were all signed by the intersection of the principals that signed each script.*

For example, assume principals A and B have signed one script, but only principal A signed another script. In this case, a page with both scripts acts as if it were signed by only A.

This assumption also means that if a signed script is on the same page as an unsigned script, both scripts act as if they were unsigned. This occurs because the signed script has a codebase principal and a certificate principal, whereas the unsigned script has only a codebase principal. (See “Codebase Principals” on page 6.) The two codebase principals are always the same for scripts from the same page; therefore, the intersection of the principals of the two scripts yields only the codebase principal. This is also what happens if both scripts are unsigned.

You can use the `import` and `export` functions to allow scripts signed by different principals to interact in a secure fashion. For information on how to do so, see “Importing and Exporting Functions” on page 18.

Checking Principals for Windows and Layers

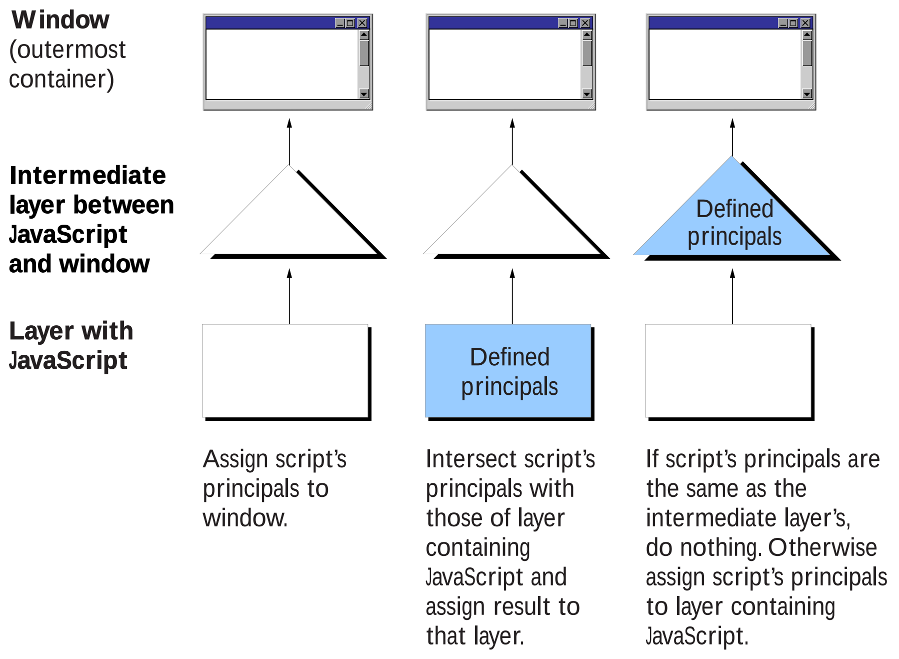
In order to protect signed scripts from tampering, Communicator 4.0 adds a new set of checks at the container level, where a container is either a window or a layer. To access the properties of a signed container, the script seeking access must be signed by a superset of the principals that signed the container.

These cross-container checks apply to most properties, whether predefined (by Communicator) or user-defined (whether by HTML content, or by script functions and variables). The cross-container checks do not apply to the following properties of `window`:

- `closed`
- `height`
- `outerHeight`
- `outerWidth`
- `pageXOffset`
- `pageYOffset`
- `screenX`
- `screenY`
- `secure`
- `width`

If all scripts on a page are signed by the same principals, container checks are applied to the window. If some scripts in a layer are signed by different principals, the special container checks apply to the layer. Figure 1 illustrates the method Communicator uses to determine which containers are associated with which sets of principals.

Figure 1 Assigning principals to layers.


Window (outermost container)
Intermediate layer between JavaScript and window
Layer with JavaScript
Defined principals
Defined principals
Assign script's principals to window.
Intersect script's principals with those of layer containing JavaScript and assign result to that layer.
If script's principals are the same as the intermediate layer's, do nothing. Otherwise assign script's principals to layer containing JavaScript.

This method works as follows: Consider each script on the page in order of declaration, treating `javascript:` URLs as new unsigned scripts.

1. If this is the first script that has been seen on the page, assign this script's principals to be the principals for the window. (If the current script is unsigned, this makes the window's principal a codebase principal.) Done.

2. If the innermost container (the container directly including the script) has defined principals, intersect the current script's principals with the container's principals and assign the result to be the principals for the container. If the two sets of principals are not equal, intersecting the sets reduces the number of principals associated with the container. Done.

3. Otherwise, find the innermost container that has defined principals. (This may be the window itself, if there are no intermediate layers.) If the principals of the current script are the same as the principals of that container, leave the principals as is. Done.

4. Otherwise, assign the current script's principals to be the principals of the container. Done.

illustrates this process.

For example, assume a page has two scripts (and no layers), with one script signed and the other unsigned. Communicator first sees the signed script, which causes the `window` object to be associated with two principals—the certificate principal from the signer of the script and the codebase principal derived from the location of the page containing the script.

When Communicator sees the second (unsigned) script, it compares the principals of that script with the principals of the current container. The unsigned script has only one principal, the codebase principal. Without layers the innermost container is the window itself, which already has principals.

Because the sets of principals differ, they are intersected, yielding a set with one member, the codebase principal. Communicator stores the result on the `window` object, narrowing its set of principals. Note that all functions that were defined in the signed script are now considered unsigned. Consequently, mixing signed and unsigned scripts on a page without layers results in all scripts being treated as if they were unsigned.

Now assume the unsigned script is in a layer on the page. This results in different behavior. In this case, when Communicator sees the unsigned script, its principals are again compared to those of the signed script in the window and the principals are found to be different. However, now that the innermost container (the layer) has no associated principals, the unsigned principals are associated with the innermost container; the outer container (the window) is untouched. In this case, signed scripts continue to operate as signed. However, accesses by the unsigned script in the layer to objects outside the layer are rejected because the layer has insufficient principals. See "Isolating an Unsigned Layer within a Signed Container" on page 17 for more information on this case.

Identifying Signed Scripts

You can sign inline scripts, event handler scripts, JavaScript files, and JavaScript entities. You cannot sign `javascript:` URLs. You must identify the thing you're signing within the HTML file:

- To sign an inline script, you add both an `ARCHIVE` attribute and an `ID` attribute to the `SCRIPT` tag for the script you want to sign. If you do not include an `ARCHIVE` attribute, Communicator uses the `ARCHIVE` attribute from an earlier script on the same page.
- To sign an event handler, you add an `ID` attribute for the event handler to the tag containing the event handler. In addition, the HTML page must also contain a signed inline script preceding the event handler. That `SCRIPT` tag must supply the `ARCHIVE` attribute.
- To sign a JavaScript entity, you do not do anything special to the entity. Instead, the HTML page must also contain a signed inline script preceding the JavaScript entity. That `SCRIPT` tag must supply the `ARCHIVE` and `ID` attributes.
- To sign an entire JavaScript file, you don't add anything special to the file. Instead, the `SCRIPT` tag for the script that uses that file must contain the `ARCHIVE` attribute.

Once you've written the HTML file, see "Signing Scripts" on page 23 for information on how to sign it.

ARCHIVE attribute

All signed scripts (inline script, event handler, JavaScript file, or JavaScript entity) require a `SCRIPT` tag's `ARCHIVE` attribute whose value is the name of the JAR file containing the digital signature. For example, to sign a JavaScript file, you could use this tag:

```
<SCRIPT ARCHIVE="myArchive.jar" SRC="myJavaScript.js"> </SCRIPT>
```

Event handler scripts do not directly specify the `ARCHIVE`. Instead, the handler must be preceded by a script containing `ARCHIVE`. For example:

```
<SCRIPT ARCHIVE="myArchive.jar" ID="a">
...
</SCRIPT>
```

```
<FORM>
<INPUT TYPE="button" VALUE="OK"
   onClick="alert('A signed script')" ID="b">
</FORM>
```

Unless you use more than one JAR file, you need only specify the file once. Include the `ARCHIVE` tag in the first script on the HTML page and the remaining scripts on the page use the same file. For example:

```
<SCRIPT ARCHIVE="myArchive.jar" ID="a">
document.write("This script is signed.");
</SCRIPT>

<SCRIPT ID="b">
document.write("This script is signed too.");
</SCRIPT>
```

ID Attribute

Signed inline and event handler scripts require the `ID` attribute. The value of this attribute is a string that relates the script to its signature in the JAR file. The `ID` must be unique within a JAR file.

When a tag contains more than one event handler script, you only need one `ID`. The entire tag is signed as one piece.

In the following example, the first three scripts use the same JAR file. The third script accesses a JavaScript file so it doesn't use the `ID` tag. The fourth script uses a different JAR file, and its `ID` of `"a"` is unique to that file.

```
<HTML>

<SCRIPT ARCHIVE="firstArchive.jar" ID="a">
document.write("This is a signed script.");
</SCRIPT>

<BODY
   onLoad="alert('A signed script using firstArchive.jar')"
   onLoad="alert('One ID needed for these event handler scripts')"
   ID="b">

<SCRIPT SRC="myJavaScript.js">
</SCRIPT>

<LAYER>
<SCRIPT ARCHIVE="secondArchive.jar" ID="a">
document.write("This script uses the secondArchive.jar file.");
```

```
</SCRIPT>
</LAYER>

</BODY>
</HTML>
```

Using Expanded Privileges

As with Java signed objects, signed scripts use calls to Netscape's Java security classes to request expanded privileges. The Java classes are explained in *Java Capabilities API*.

In the simplest case, you add one line of code asking permission to access a particular target representing the resource you want to access. (See "Targets" on page 14 for more information.) For example:

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalSendMail")
```

When the script calls this function, the signature is verified, and if the signature is valid, expanded privileges can be granted. If necessary, a dialog displays information about the application's author, and gives the user the option to grant or deny expanded privileges.

Privileges are granted only in the scope of the requesting function and only after the request has been granted in that function. This scope includes any functions called by the requesting function. When the script leaves the requesting function, privileges no longer apply.

The following example demonstrates this by printing:

```
7: disabled
5: disabled
2: disabled
3: enabled
1: enabled
4: enabled
6: disabled
8: disabled
```

Function `g` requests expanded privileges, and only the commands and functions called after the request and within function `g` are granted privileges.

```
<SCRIPT ARCHIVE="ckHistory.jar" ID="a">

function printEnabled(i) {
   if (history[0] == "") {
      document.write(i + ": disabled<BR>");
   } else {
      document.write(i + ": enabled<BR>");
   }
}

function f() {
   printEnabled(1);
}

function g() {
   printEnabled(2);
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserRead");
   printEnabled(3);
   f();
   printEnabled(4);
}

function h() {
   printEnabled(5);
   g();
   printEnabled(6);
}

printEnabled(7);
h();
printEnabled(8);

</SCRIPT>
```

Targets

The types of information you can access are called targets. These are listed below.

Target	Description
`UniversalBrowserRead`	Allows reading of privileged data from the browser. This allows the script to pass the same origin check for any document.
`UniversalBrowserWrite`	Allows modification of privileged data in a browser. This allows the script to pass the same origin check for any document.
`UniversalBrowserAccess`	Allows both reading and modification of privileged data from the browser. This allows the script to pass the same origin check for any document.

Target	Description
`UniversalFileRead`	Allows a script to read any files stored on hard disks or other storage media connected to your computer.
`UniversalPreferencesRead`	Allows the script to read preferences using the `navigator.preference` method.
`UniversalPreferencesWrite`	Allows the script to set preferences using the `navigator.preference` method.
`UniversalSendMail`	Allows the program to send mail in the user's name.

For a complete list of targets, see *Netscape System Targets*.

JavaScript Features Requiring Privileges

This section lists the JavaScript features that require expanded privileges and the target used to access each feature. Unsigned scripts cannot use any of these features, unless the end user has enabled codebase principals.

- Setting a file upload widget requires `UniversalFileRead`.
- Submitting a form to a `mailto:` or `news:` URL requires `UniversalSendMail`.
- Using an `about:` URL other than `about:blank` requires `UniversalBrowserRead`.
- `event` object: Setting any property requires `UniversalBrowserWrite`.
- `DragDrop` event: Getting the value of the `data` property requires `UniversalBrowserRead`.
- `history` object: Getting the value of any property requires `UniversalBrowserRead`.
- `navigator` object:
  - — Getting the value of a preference using the `preference` method requires `UniversalPreferencesRead`.
  - — Setting the value of a preference using the `preference` method requires `UniversalPreferencesWrite`.

- `window` object: Allow of the following operations require `UniversalBrowserWrite`.
  - — Adding or removing the directory bar, location bar, menu bar, personal bar, scroll bar, status bar, or toolbar.
  - — Using the methods in the following table under the indicated circumstances

`enableExternalCapture`	To capture events in pages loaded from different servers. Follow this method with `captureEvents`.
`close`	To unconditionally close a browser window.
`moveBy`	To move a window offscreen.
`moveTo`	To move a window offscreen.
`open`	• To create a window smaller than 100 x 100 pixels or larger than the screen can accommodate by using `innerWidth`, `innerHeight`, `outerWidth`, and `outerHeight`. • To place a window off screen by using `screenX` and `screenY`. • To create a window without a titlebar by using `titlebar`. • To use `alwaysRaised`, `alwaysLowered`, or `z-lock` for any setting.
`resizeTo`	To resize a window smaller than 100 x 100 pixels or larger than the screen can accommodate.
`resizeBy`	To resize a window smaller than 100 x 100 pixels or larger than the screen can accommodate.

  - — Setting the properties in the following table under the indicated circumstances:

`innerWidth`	To set the inner width of a window to a size smaller than 100 x 100 or larger than the screen can accommodate.
`innerHeight`	To set the inner height of a window to a size smaller than 100 x 100 or larger than the screen can accommodate.

Example

The following script includes a button, that, when clicked, displays an alert dialog containing part of the URL history of the browser. To work properly, the script must be signed.

```
<SCRIPT ARCHIVE="myArchive.jar" ID="a">

function getHistory(i) {
   //Attempt to access privileged information
   return history[i];
}

function getImmediateHistory() {
   //Request privilege
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserRead");
   return getHistory(1);
}

</SCRIPT>
...
<INPUT TYPE="button" onClick="alert(getImmediateHistory());" ID="b">
```

Writing the Script

This section describes special considerations for writing signed scripts. For more tips on writing your scripts, see Danny Goodman's *View Source*[1] article, *Applying Signed Scripts*.[2]

Capturing Events from Other Locations

If a window with frames needs to capture events in pages loaded from different locations (servers), use the `enableExternalCapture` method in a signed script requesting `UniversalBrowserWrite` privileges. Use this method before calling the `captureEvents` method. For example, with the following code the window can capture all Click events that occur across its frames.

```
<SCRIPT ARCHIVE="myArchive.jar" ID="archive">
...
function captureClicks() {
   netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
```

1. http://developer.netscape.com/news/viewsource/index.html
2. http://developer.netscape.com/news/viewsource/goodman_sscripts.html

```
   enableExternalCapture();
   captureEvents(Event.CLICK);
   ...
}
...
</SCRIPT>
```

Isolating an Unsigned Layer within a Signed Container

To create an unsigned layer within a signed container, you need to perform some additional steps to make scripts in the unsigned layer work properly.

- You must set the `__parent__` property of the layer object to `null` so that variable lookups performed by the script in the unsigned layer do not follow the parent chain up to the `window` object and attempt to access the `window` object's properties, which are protected by the container check.
- Because the standard objects (`String`, `Array`, `Date`, and so on) are defined in the `window` object and not normally in the layer, you must call the `initStandardObjects` method of the `layer` object. This creates copies of the standard objects in the layer's scope.

International Characters in Signed Scripts

When used in scripts, international characters can appear in string constants and in comments. JavaScript keywords and variables cannot include special international characters.

Scripts that include international characters cannot be signed because the process of transforming the characters to the local character set invalidates the signature. To work around this limitation:

- Escape the international characters (`'0x\ea'`, and so on).
- Put the data containing the international characters in a hidden form element, and access the form element through the signed script.
- Separate signed and unsigned scripts into different layers, and use the international characters in the unsigned scripts.

- Remove comments that include international characters.

There is no restriction on international characters the HTML surrounding the signed scripts.

Importing and Exporting Functions

You might want to provide interfaces to call into secure containers (windows and layers). To do so, you use the `import` and `export` statements. Exporting a function name makes it available to be imported by scripts outside the container without being subject to a container test.

You can only import and export functions, either top-level functions (associated with a `window` object) or methods of some other object. You cannot import or export entire objects or properties that aren't functions.

Importing a function into your scope creates a new function of the same name as the imported function. Calling that function calls the corresponding function from the secure container.

To use `import` and `export`, you must explicitly set the `LANGUAGE` attribute of the `SCRIPT` tag to `"JavaScript1.2"`.

In the signed script that defines a function you want to let other scripts access, use the `export` statement. The syntax of this statement is:

```
exportStmt ::= export exprList
exprList ::= expr | expr, exprList
```

where each *expr* must resolve to the name of a function. The `export` statement marks each function as importable.

In the script in which you want to import that function, use the `import` statement. The syntax of this statement is:

```
importStmt ::= import importList
importList ::= importElem | importElem, importList
importElem ::= expr.funName | expr.*
```

Executing `import` *expr.funName* evaluates *expr* and then imports the *funName* function of that object into the current scope. It is an error if *expr* does not evaluate to an object, if there is no function named *funName*, or if the function exists but has not been marked as importable. Executing `import` *expr*`.*` imports all importable functions of *expr*.

Example

The following example has three pages in a frameset. The file `containerAccess.html` defines the frameset and calls a user function when the frameset is loaded. One page, `secureContainer.html`, has signed scripts and exports a function. The other page, `access.html`, imports the exported function and calls it.

While this example exports a function that does not enable or require expanded privileges, you can export functions that do enable privileges. If you do so, you should be very careful to not inadvertently allow access to an attacker. For more information, see "Be Careful What You Export" on page 21.

File containerAccess.html

```
<HTML>
<FRAMESET NAME=myframes ROWS="50%,*" onLoad="inner.myOnLoad()">
<FRAME NAME=inner SRC="access.html">
<FRAME NAME=secureContainer SRC="secureContainer.html">
</FRAMESET>
</HTML>
```

File secureContainer.html

```
<HTML>
This page defines a variable and two functions.
Only one function, publicFunction, is exported.
<BR>

<SCRIPT ARCHIVE="secureContainer.jar" LANGUAGE="JavaScript1.2" ID="a">

function privateFunction() {
   return 7;
}

var privateVariable = 23;

function publicFunction() {
   return 34;
}
export publicFunction;

netscape.security.PrivilegeManager.enablePrivilege(
   "UniversalBrowserRead");
document.write("This page is at " + history[0]);

// Privileges revert automatically when the script terminates.
</SCRIPT>
</HTML>
```

File access.html

```
<HTML>
This page attempts to access an exported function from a signed
container. The access should succeed.

<SCRIPT LANGUAGE="JavaScript1.2">

function myOnLoad() {
   var ctnr = top.frames.secureContainer;
   import ctnr.publicFunction;
   alert("value is " + publicFunction());
}

</SCRIPT>
</HTML>
```

Hints for Writing Secure JavaScript

Check the Location of the Script

If you have signed scripts in pages you have posted to your site, it is possible to copy the JAR file from your site and post it on another site. As long as the signed scripts themselves are not altered, the scripts will continue to operate under your signature. (See “Debugging Invalid Hash Errors” on page 25 for one exception to this rule.)

If you wish to prevent this, you can force your scripts to work only from your site.

```
<SCRIPT ARCHIVE="siteSpecific.jar" ID="a" LANGUAGE="JavaScript1.2">
if (document.URL.match(/^http:\/\/www.company.com\//)) {
   netscape.security.PrivilegeManager.enablePrivilege(...);
   // Do your stuff
}
</SCRIPT>
```

Then if the JAR file and script are copied to another site, they no longer work. If the person who copies the script alters it to bypass the check on the source of the script, the signature is invalidated.

Be Careful What You Export

When you export functions from your signed script, you are in effect transferring any trust the user has placed in you to any script that calls your functions. This means you have a responsibility to ensure that you are not exporting interfaces that can be used in ways you do not want. For example, the following program exports a call to `eval` that can operate under expanded privileges.

```
<SCRIPT ARCHIVE="duh.jar" ID="a">
function myEval(s) {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalFileAccess");
   return eval(s);
}
export myEval; // Don't do this!!!!
</SCRIPT>
```

Now any other script can import `myEval` and read and write any file on the user's hard disk using trust the user has granted to you.

Minimize the Trusted Code Base

In security parlance, the **trusted code base** (TCB) is the set of code that has privileges to perform restricted actions. One way to improve security is reduce the size of the TCB, which then gives fewer points for attack or opportunities for mistakes.

For example, the following code, if executed in a signed script with the user's approval, opens a new window containing the history of the browser:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
netscape.security.PrivilegeManager.enablePrivilege(
   "UniversalBrowserAccess");
var win = window.open();
for (var i=0; i < history.length; i++) {
   win.document.writeln(history[i] + "<BR>");
}
win.close();
</SCRIPT>
```

The TCB in this instance is the entire script because privileges are acquired at the beginning and never reverted. You could reduce the TCB by rewriting the program as follows:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
var win = window.open();
netscape.security.PrivilegeManager.enablePrivilege(
   "UniversalBrowserAccess");
```

```
for (var i=0; i < history.length; i++) {
   win.document.writeln(history[i] + "<BR>");
}
netscape.security.PrivilegeManager.revertPrivilege(
   "UniversalBrowserAccess");
win.close();
</SCRIPT>
```

With this change, the TCB becomes only the loop containing the accesses to the `history` property. You could avoid the extra call into Java to revert the privilege by introducing a function:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
function writeArray() {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserAccess");
   for (var i=0; i < history.length; i++) {
      win.document.writeln(history[i] + "<BR>");
   }
}
var win = window.open();
writeArray();
win.close();
</SCRIPT>
```

The privileges are automatically reverted when `writeArray` returns, so you don't have to do so explicitly.

Use the Minimal Capability Required for the Task

Another way of reducing your exposure to exploits or mistakes is by only using the minimal capability required to perform the given access. For example, the previous code requested `UniversalBrowserAccess`, which is a macro target containing both `UniversalBrowserRead` and `UniversalBrowserWrite`. Only `UniversalBrowserRead` is required to read the elements of the `history` array, so you could rewrite the above code more securely:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
function writeArray() {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserRead");
   for (var i=0; i < history.length; i++) {
      win.document.writeln(history[i] + "<BR>");
   }
}
var win = window.open();
writeArray();
```

```
win.close();
</SCRIPT>
```

Signing Scripts

During development of a script you'll eventually sign, you can use codebase principals for testing, as described in "Codebase Principals" on page 6. Once you've finished modifying the script, you need to sign it.

For any script to be granted expanded privileges, all scripts on the same HTML page or layer must be signed. If you use layers, you can have both signed and unsigned scripts as long as you keep them in separate layers. For more information, see "Signed Script Policy" on page 4.

You can sign JavaScript files (accessed with the `SRC` attribute of the `SCRIPT` tag), inline scripts, event handler scripts, and JavaScript entities. You cannot sign `javascript:` URLs. Before you sign the script, be sure you've properly identified it, as described in "Identifying Signed Scripts" on page 10.

Using Page Signer

Use Page Signer to sign scripts. Page Signer is a Perl script (`signPages`) that uses JAR Packager Command Line to sign your scripts and package the digital signature and related information in a JAR file. For information on JAR Packager Command Line, see *Using JAR Packager Command Line*.[1]

The `signPages` script extracts scripts from HTML files, signs them, and places their digital signatures in the archive specified by the `ARCHIVE` attribute in the `SCRIPT` tag from the HTML files. It also takes care of copying external JavaScript files loaded by the `SRC` attribute of the `SCRIPT` tag. The `SCRIPT` tags in the HTML pages can specify more than one JAR file; if so, `signPages` creates as many JAR files as it needs.

For information on using this tool, see *Using Page Signer*.

1. http://developer.netscape.com/library/documentation/signedobj/command/index.htm

After Signing

Once you've signed a script, any time you change it you must resign it. For JavaScript files, this means you cannot change anything in the file. For inline scripts, you cannot change anything between the initial `<SCRIPT ...>` and the closing `</SCRIPT>`. For event handlers and JavaScript entities, you cannot change anything at all in the tag that includes the handler or entity.

A change can be as simple as adding or removing whitespace in the script.

Changes to a signed script's byte stream invalidate the script's signature. This includes moving the HTML page between platforms that have different representations of text. For example, moving an HTML page from a Windows server to a UNIX server changes the byte stream and invalidates the signature. (This doesn't affect viewing pages from multiple platforms.) To avoid this, you can move the page in binary mode. Note that doing so changes the appearance of the page in your text editor but not in the browser.

Although you cannot make changes to the script, you can make changes to the surrounding information in the HTML file. You can even copy a signed script from one file to another, as long as you make sure you change nothing within the script.

Troubleshooting Signed Scripts

Errors on the Java Console

Be sure to check the Java console for errors if your signed scripts do not function as expected. You may see errors such as the following:

```
# Error: Invalid Hash of this JAR entry (-7882)
# jar file: C:\Program Files\Netscape\Users\norris\cache\MVI9CF1F.JAR
# path: 1
```

The path value printed for signed JavaScript is either the value of the `ID` attribute or the `SRC` attribute of the tag that supplied the script.

Debugging Invalid Hash Errors

Invalid hash errors occur if the script has changed from when it was signed. The most common cause of this problem is that the scripts have been moved from one platform to another with a text transfer rather than a binary transfer. Because line separator characters can differ from platform to platform, the hash could change from when the script was originally signed.

One good way to debug this sort of problem is to use the `-s` option to `signPages`, which will save the inline scripts in the JAR file. You can then unpack the jar file when you get the hash errors and compare it to the HTML file to track down the source of the problems. For information on `signPages`, see *Using Page Signer*.

"User did not grant privilege" Exception or Unsigned Script Dialog

Depending on whether or not you have enabled codebase principals, you see different behavior if a script attempts to enable privileges when it isn't signed or when its principals have been downgraded due to mixing.

If you have not enabled codebase principals and a script attempts to enable privileges for an unsigned script, it gets an exception from Java that the "user did not grant privilege". If you did enable codebase principals, you will see a Java security dialog that asking for permissions for the unsigned code.

This behavior is caused by either an error in verifying the certificate principals (which will cause an error to be printed to the Java console; see "Errors on the Java Console" on page 25), or by mixing signed and unsigned scripts. There are many possible sources of unsigned scripts. In particular, because there is no way to sign Javascript: URLs or dynamically generated scripts, using them causes the downgrading of principals.