Chapter 46: Security and Netscape Signed Scripts

In This Chapter

Exploring browser security policies

Applying JavaScript to Navigator security mechanisms

Using Netscape signed scripts

The paranoia levels about potential threats to security and privacy on the Internet are at an all-time high. As more people rely on e-mail and Web site content for their daily lives and transactions, the fears will only increase for the foreseeable future (an indeterminate number of Web Weeks). As a jokester might say, though, “I may be paranoid, but how do I know someone really isn’t out to get me?” The answer to that question is that you don’t know, and such a person may be out there.

But Web software developers are doing their darnedest to put up roadblocks to those persons out to get you — hence, the many levels of security that pervade browsers. Unfortunately, these roadblocks also get in the way of scripters who have completely honest intentions. Designing a Web site around these barriers is one of the greatest challenges that many scripters face.

Battening Down the Hatches

When Navigator 2 first shipped to the world (way back in February 1996), it was the first browser released to include support for Java applets and scripting — two entirely different but often confused technologies. It didn’t take long for clever programmers in the Internet community to find the ways in which one or the other technology provided inadvertent access to client computer information (such as reading file directories) and Web surfer activities (such as histories of where you’ve been on the Net and even the passwords you may have entered to access secure sites).

JavaScript, in particular, was the avenue that many of these programmers used to steal such information from Web site visitors’ browsers. The sad part is that the same features that provide the access to the information were intentionally made a part of the initial language to aid scripters who would put those features to beneficial use in controlled environments, such as intranets. But out in the Wild Wide Web, a scripter could capture a visitor’s e-mail address by having the site’s home page surreptitiously send a message to the site’s author without the visitor even knowing it.

Word of security breaches of this magnitude not only circulated throughout the Internet, but also reached both the trade and mainstream press. As if the security issues weren’t bad enough on their own, the public relations nightmare compounded the sense of urgency to fix the problem. To that end, Netscape released two revised editions of Navigator 2. The final release of that generation of browser, Navigator 2.02, took care of the scriptable security issues by turning off some of the scripted capabilities that had been put into the original 2.0 version. No more capturing visitors' browser histories; no more local file directory listings; no more silent e-mail. Users could even turn off JavaScript support entirely if they so desired.

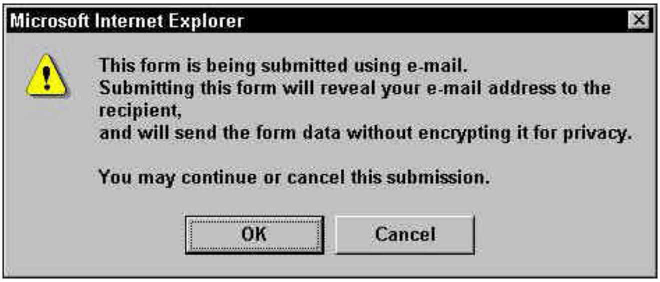
The bottom line on security is that scripts are prevented from performing automated processes that invade the private property of a Web author's page or a client's browser. Thus, any action that may be suspect, such as sending an e-mail message, requires an explicit action on the part of the user — clicking a Submit button, in this case — to carry it out. Security restrictions must also prevent a Web site from tracking your activity beyond the boundaries of that Web site.

When Worlds Collide

If a script tries to do something that is not allowed or is a potential personal security breach, the browser reports the situation to the user. Figure 46-1, for instance, shows an IE/Windows warning a user gets from clicking a Submit button located in a form whose `ACTION` is set to a `mailto:` URL.

Another security error message often confuses scripters who don't understand the possible privacy invasions that can accrue from one window or frame having access to the URL information in another window or frame. In IE5/Windows, for example, an ominous error message — "Permission denied" — warns users of an attempt to access URL information from another frame if that URL is from a different Web site.

Despite the fact that a scripted Web site may have even loaded the foreign URL into the other frame, the security restrictions guard against unscrupulous usage of the ability to snoop in other windows and frames.


Microsoft Internet Explorer

This form is being submitted using e-mail.
Submitting this form will reveal your e-mail address to the recipient,
and will send the form data without encrypting it for privacy.

You may continue or cancel this submission.

Figure 46-1
IE/Windows e-mail warning

The Java Sandbox

Much of the security model for JavaScript is similar to that originally defined for Java applets. Applets had a potentially dangerous facility of executing Java code on the client machine. That is a far cry from the original deployment of the World Wide Web as a read-only publishing medium on the Internet. Here were mini-programs downloaded into a client computer that, if unchecked, could have the same access to the system as a local software program.

Access of this type would clearly be unacceptable. Imagine the dismay caused by someone clicking a link that said “Free Money,” only to have the linked page download an applet that read or damaged local disk files unbeknownst to the user. In anticipation of pranksters, the designers of Java and the Java virtual machine built in a number of safeguards to prevent applets from gaining access to local machines. This mechanism is collectively referred to as the sandbox, a restricted area in which applets can operate. Applets cannot extend their reach outside of the sandbox to access local file systems and many sensitive system preferences. Any applet runs only while its containing page is still loaded in the browser. When the page goes away, so does the applet, without being saved to the local disk cache.

JavaScript adopted similar restrictions. The language provided no read or write access to local files beyond the highly regulated cookie file. Moreover, because JavaScript works more closely with the browser and its documents than applets typically do, the language had to build in extra restrictions to prevent browser-specific privacy invasions. For example, it was not possible for a script in one window to monitor the user’s activity in another window, including the URL of the other window, if the page didn’t come from the same server as the first window. Sometimes the restrictions on the JavaScript side are even more severe than in Java. For example, while a Java applet is permitted to access the network anytime after the applet is loaded, an applet is prevented from reaching out to the Net if the trigger for that transaction comes from JavaScript via LiveConnect (see Chapter 44). Only partial workarounds are available.

Neither the Java nor JavaScript security blankets were fully bug-free at the outset. Some holes were uncovered by the languages’ creators and others in the community. To their credit, Sun and Netscape (and Microsoft for that matter) are quick to plug any holes that are discovered. While the plugs don’t necessarily fix existing copies of insecure browsers out there, it means that a Bad Guy can’t count on every browser to offer the same security hole for exploitation. That generally makes the effort not worth the bother.

Security Policies

Netscape has defined *security mechanism* under the term policies. This usage of the word mirrors that of institutions and governments: A policy defines the way potentially insecure or invasive requests are handled by the browser or scripting language. NN4+ includes two different security policies: *same origin* and *signed* script policies. The same origin policy dates back to Navigator 2, although some additional rules have been added to that policy as Navigator has matured. The signed script policy started with NN4 and utilizes the state of the art in cryptographic signatures of executable code inside a browser, whether that code is a plug-in, a Java applet, or a JavaScript script. Because of the signed script facilities, NN4+ was designed to allow scripts to have wider range of control over the browser's interior working parts, provided the user granted permission for such activity (more about this later in the chapter). NN3 included a partially implemented prototype of another policy known as data tainting. Signed scripts supersede data tainting, so if you encounter any writings about data tainting, you can ignore them because the technology is not being further developed.

By and large, the same origin policy is in force inside IE3 and after. Precise details may not match up with NN one-for-one, but the most common features are identical. The signed script policy is implemented only in NN4+. While Microsoft offers digital signatures for some items that may be embedded within an HTML page (such as ActiveX controls and other components), scripts that are in an HTML page's source code or linked in as a `.js` library cannot be signed for IE. While everything you read in this chapter about signed scripts applies only to NN4+, you should find the next couple of sections informative even if you develop solely for IE.

The Same Origin Policy

The "origin" of the same origin policy means the protocol and domain of a source document. If all of the source files currently loaded in the browser come from the same server and domain, scripts in any one part of the environment can poke around the other documents. Restrictions come into play when the script doing the poking and the document being poked come from different origins. The potential security and privacy breaches this kind of access can cause put this access out of bounds within the same origin policy.

An origin is not the complete URL of a document. Consider the two popular URLs for Netscape's Web sites:

```
http://home.netscape.com
http://developer.netscape.com
```

The protocol for both sites is `http:`. Both sites also share the same domain name: `netscape.com`. But the sites run on two different servers: `home` and `developer` (at least this is how the sites appear to browsers accessing them; the physical server arrangement may be quite different).

If a frameset contains documents from the same server at `netscape.com`, and all frames are using the same protocol, then they have the same origin. Completely open and free access to information, such as `location` object properties, is available to scripts in any frame's document. But if one of those frames contains a document from the other server, their origins don't match. A script in a document from one server would display an “access disallowed” or “permission denied” error message if it tried to get the `location` property of that other document.

A similar problem occurs if you were creating a Web-based shopping service that displays the product catalog in one window and displays the order form from a secure server in another window. The order form, whose protocol might be `https:`, would not be granted access to the `location` object properties in a catalog page whose protocol is `http:`, even though both share the same server and domain name.

Setting the document.domain

When both pages in an origin-protected transaction are from the same domain (but different servers or protocols), you can instruct JavaScript to set the `document.domain` properties of both pages to the domain that they share. When this property is set to that domain, the pages are treated as if from the same origin. Making this adjustment is safe, because JavaScript doesn’t allow setting the `document.domain` property to any domain other than the origin of the document making the setting. See the `document.domain` property entry in Chapter 18 for further details.

Origin checks

Scattered throughout the language reference chapters are notes about items that undergo what you now know to be origin checks. For the sake of convenience, I list them all here to help you get a better feeling for the kind of information that is protected. The general rule is that any object property or method that exposes a local file in a user’s system or can trace Web surfing activity in another window or frame undergoes an origin check. Failure to satisfy the same origin rule yields an “access disallowed” or “permission denied” error message on the client’s machine.

Window object checks

The document object models of windows and frames that don’t share the same origin are not available to each other. Each separate origin window or frame is its own little world that has very little ability to communicate with another window or frame. IE sometimes takes this to the extreme, causing problems between a main window and a subwindow whose content is entirely dynamically generated from the main window’s scripts.

Location object checks

All `location` object properties are restricted to same origin access. Of all same origin policy restrictions, this one seems to interfere with well-meaning page authors’ plans when they want to provide a frame for users to navigate around the Web. Such access, however, would allow spying on your users.

Document object checks

A `document` object's properties are by necessity loaded with information about the content of that document. Just about every property other than the ones that specify color properties are off-limits if the origin of the target document is different from the one making the request:

```
anchors[]
applets[]
cookie
domain
embeds
forms[]
lastModified
length
links[]
referrer
title
URL
```

In addition, no normally modifiable `document` property can be modified if the origin check fails. This, of course, does not prevent you from using `document.write()` to write an entirely new page of content to the frame to replace a document from a different origin. But in IE4+ and W3C DOM browsers, scripts from one origin won't be able to modify (or even copy) partial content from a frame whose content comes from another origin.

NN4 layer object checks

While most of a NN4 layer's content is protected by the restrictions that apply to the `document` object inside, a layer object also has a potentially revealing `src` property. This is essentially similar to the `location.href` property of a frame. Thus the `src` property requires an origin check before yielding its information.

Form object checks

Form data is generally protected by the restriction to a document's `forms[]` array. But should a script in another window or frame also know the name of the form, that, too, won't enable access unless both documents come from the same origin.

Applet object checks

The same goes for named Java applets. A script cannot retrieve information about the class file name unless both documents are from the same origin (although the applet can be from anywhere).

LiveConnect access from a Java applet to JavaScript is not an avenue to other windows and frames from other origins. Any calls from the applet to the objects and protected properties described here undergo origin checks when those objects are in other frames and windows. The applet assumes the origin of the document that contains the applet, not the applet codebase.

Image object checks

While image objects are accessible from other origins, their `src` and `lowsrc` properties are not. These URLs could reveal some or all the URL info about the document containing them.

Linked script library checks

To prevent a network-based script from hijacking a local script library file, NN4+ prevents a page from loading a `file:` protocol library in the `SRC` attribute of a `<SCRIPT>` tag unless the main document also comes from a `file:` protocol source. If you are beginning to think that security engineers are a suspicious and paranoid lot, you are starting to get the idea. It's not easy to curb potential abuses of Bad Guys in a networked environment initially established for openness and free exchange of information among trusted individuals.

The Netscape Signed Script Policy

Just as there are excellent reasons to keep Web pages from poking around your computer and browser, there are equally good reasons to allow such access to a Web site you trust not to be a Bad Guy. To permit trusted access to the client machine and browser, Sun Microsystems and Netscape (in cooperation with other sources) have developed a way for Web application authors to identify themselves officially as authors of the pages and to request permission of the user to access well-defined parts of the computer system and browser.

The technology is called object *signing*. In broad terms, object signing means that an author can electronically lock down a chunk of computer code (whether it be a Java applet, a plug-in, or a script) with the electronic equivalent of a wax seal stamped by the author's signet ring. At the receiving end, a user is informed that a sealed chunk of code is requesting some normally protected access to the computer or browser. The user can examine the "seal" to see who authored the code and the nature of access being requested. If the user trusts the author not to be a Bad Guy, the user grants permission for that code to execute; otherwise the code does not run at all. Additional checks take place before the code actually runs. That electronic "seal" contains an encrypted, reduced representation of the code as it was locked by the author. If the encrypted representation cannot be re-created at the client end (it takes only a fraction of a second to check), it means the code has been modified in transit and will not run.

In truth, nothing prevents an author from being a Bad Guy, including someone you may normally trust. The point of the object signing system, however, is that a trail leads back to the Bad Guy. An author cannot use this technology to sneak into your computer or browser without your explicit knowledge and permission.

Signed objects and scripts

A special version of the signed object technology is the one that lets scripts be locked down by their author and electronically signed. Virtually any kind of script in a document can be signed: a linked `.js` library, scripts in the document, event handlers, and JavaScript entities. As described later in this chapter, you must prepare your scripts for being signed, and then run the entire page through a special tool that attaches your electronic signature to the scripts within that page.

What you get with signed scripts

If you sign your scripts and the user grants your page permission to do its job, signed scripts open up your application to a long list of capabilities, some of which border on acting like genuine local applications. Because the designers of NN4+ know that signed scripts are available to authors, a huge number of properties and actions are exposed to authors.

The most obvious power you get with signed scripts is freedom from the restrictions of the same origin policy. All object properties and methods that perform origin checks for access in other frames and windows become available to your scripts without any special interaction with the user beyond the dialog box that requests the one-time permission for the page.

Some operations that normally display warnings about impending actions — sending a form to a `mailto:` URL or closing the main browser window under script control — lose those warning dialog boxes if the user grants the appropriate permission to a signed script. Object properties considered private information, such as individual URLs stored in the history object and browser preferences, are opened up, including the possibility of altering browser preferences. Existing windows can have their chrome elements hidden. New windows can be set to be always raised or lowered, sized to very small sizes, or positioned offscreen. The `dragDrop` event of a window reveals its URL. All of these are powerful points of access, provided the user grants permission.

Again, however, I emphasize that these capabilities are accessible via Netscape's signed script policy only. Internet Explorer, at least through Version 5.5, does not support Netscape's signed script policy.

The Digital Certificate

Before you can sign a script or other object, you must apply for a digital certificate. A digital certificate (also called a digital ID) is a small piece of software that gets downloaded and bound to the developer's Navigator browser on a particular computer. Each downloaded digital certificate appears in the list of certificates under the “Mine” category in Navigator 6's Security Manager window (accessible through the Tasks menu). If you have not yet applied for a certificate, the list is empty. When you sign a page with the certificate, information about the certificate is included in the file generated by the signing tool.

Possession of a certificate makes you what is known as a principal. If a user loads a page that has signed “stuff” in it, a security alert advises the user that a Web site is requesting enhanced privileges.

Certificates are issued by organizations established as certificate authorities. A certificate authority (known as a CA for short), or a certificate server authorized by a CA, registers applicants and issues certificates to individuals and software developers. When you register for a certificate, the CA queries you for identification information, which it verifies as best it can. The certificate that is issued to you identifies you as the holder of the certificate. Under the “Authorities” category of the Security Manager window are the certificate authorities loaded into the browser when you installed the browser. These are organizations that issue certificates. The CA of the organization that issues your certificate must be listed for you to sign scripts.

How to get a certificate

You must visit a certificate vendor to obtain your certificate. The cost ranges from about $20 to many hundreds of dollars depending on the vendor and the type of certificate you want to obtain. One vendor that is aware of the needs of Netscape object signing is Thawte Digital Certificate Services (`www.thawte.com`). This CA offers a certificate expressly for developers performing Netscape object signing. Verisign (`www.verisign.com`) is also recommended.

Because one of the foundations of a certificate is the identity of the certificate owner, registration requires submitting documentation that proves the identity of your organization. Each CA has different requirements, so check the latest information at the CA’s enrollment Web site. After the CA processes your application, the company sends you an e-mail message with a code number to pick up your certificate at the CA’s Web site. The act of picking up the certificate is actually downloading the certificate into your browser. Therefore, be sure you are using the Navigator browser on the computer with which you will use to sign your pages.

Activating the codebase principal

If you want to try out the capabilities available to signed scripts from a server without purchasing a certificate (or without going through the signing process described later in this chapter during script development and debugging), you can set up your copy of Navigator to accept what is called a *codebase* principal in place of a genuine certificate. A codebase principal means that the browser accepts the source file as a legitimate principal, although it contains no identification as to the owner or certificate.

Depending on which version of Navigator you are running, if you set up your browser for codebase principals, you may not be able to verify a certificate that is presented to you when accessing someone else's Web site — even if it is a valid cryptographic certificate. Therefore, even though secure requests won't slip past you silently, your Navigator won't necessarily have the protective shield it normally does to identify certificate holders beyond the URL of the code. Enable codebase principals only on a copy of Navigator that doesn't venture beyond your development environment. To activate codebase principals for your copy of Navigator, follow these steps:

1. Quit Navigator.

2. Search your hard disk for a Navigator 4 support file named `prefs.js` or Navigator 6 support file named `all.js`.

3. Edit the version-specific file in a text editor as follows:

   a. For NN4, add the following line to the end of the `prefs.js` file:

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

   b. For NN6, change the codebase_principal preference in `all.js` from `false` to `true`:

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

4. Save the file.

To deactivate codebase principals, quit Navigator and then change the `true` setting of the affected line to `false`. Because Navigator 4 rebuilds the preference file upon quitting, the entry will be in alphabetical order rather than at the end of the file where you first entered it. This preferences setting does not affect your ability to sign scripts with your certificate as described in the rest of this article.

Signing Scripts

The process of signing scripts entails some new concepts for even experienced JavaScript authors. You must use a separate signing tool program. You must also prepare the page that bears scripts so that the tool and the object signing facilities of the browser can do their jobs.

Signing tool

Download the latest version of Netscape's SignTool from links you find at `http://developer.netscape.com:80/software/signedobj/jarpack.html` (you find different versions for a variety of Windows and Unix versions). This tool includes a utility program called a JAR Packager. A JAR file is a special kind of zipped file collection that has been designed to work with the Navigator security infrastructure. The letters of the name stand for Java ARchive, which is a file format standard developed primarily by Sun Microsystems in cooperation with Netscape and others.

A JAR file’s extension is `.jar`, and when it contains signed script information, it holds at least one file, known as the *manifest*, or list of items zipped together in the file. Among the items in the manifest is certificate information and data (a hash value code) about the content of the signed items at the instant they were signed. In the case of a single page containing signed scripts, the JAR file contains only the certificate and hash values of the signed scripts within the document. If the document links in an external `.js` script library file, that library file is also packaged in the JAR file. Thus, a page with signed scripts occupies space on the server for the HTML file and its companion JAR file.

The SignTool program combines the JAR Packager with the script signing functions (originally a separate program called `zigbert.exe`). Follow links on the SignTool download page to the latest instructions on packaging and signing your finished files from the command line (there is no GUI for this tool). But before you reach that point, you need to compose your pages in a way that the security mechanism can protect your scripts.

Preparing scripts for signing

Signifying which items in a page are script items that require signing is up to the page author. It is important to remember that if you want to sign even one script element in a document, *every* script in the document must be signed. By “document,” I mean an object model document. Because the content of an NN4-only `<LAYER>` tag exists in its own document, you don’t have to sign its scripts if they don’t require it, nor communicate with the signed scripts in the main document.

The first concept you have to master is recognizing what a script is. For signing purposes, a script is more than just the set of statements between a `<SCRIPT>` and `</SCRIPT>` tag boundary. An event handler — even one that calls a function living in a `<SCRIPT>` tag — is also a script that needs signing. So, too, is a JavaScript entity used to supply a value to an HTML tag attribute. Each one of these items is a script as far as script signing is concerned.

Your job is to mark up the file with special tag attributes that do two things: 1) help SignTool know what items to sign in a file; and 2) help the browser loading the signed document know what items to run through the hash routine again to compare against the values stored in the JAR file.

The ARCHIVE attribute

The first attribute goes in the first `<SCRIPT>` tag of the file, preferably near the very top of the document in the `<HEAD>` portion. This attribute is the `ARCHIVE` attribute, and its value is the name of the JAR file to be associated with the HTML file. For example

```
<SCRIPT LANGUAGE=”JavaScript” ARCHIVE=”myArchive.jar” ID=”1”>
```

You can add script statements to this tag or immediately end it with a `</SCRIPT>` tag.

SignTool utility uses the ~~ARCHIVE~~ attribute to assign a name to its archive output file. After the signed page loads into the visitor’s browser, the attribute points to the file containing signed script information. Having more than one JAR archive file associated with a signed page is possible. Typically, such a situation calls for a second JAR archive overseeing a confined portion of the page. That second archive file may even be embedded in the primary archive file, allowing a script segment signed by one principal to be combined with scripts signed by a different principal.

The ID attribute

More perplexing to scripters coming to script signing for the first time is the ID attribute. The ID attribute is merely a label for each script. Each script must have a label that is unique among all labels specified for a JAR archive file.

As with the `ARCHIVE` attribute, the `ID` plays a dual role. When you run your page through SignTool, the utility scans the page for these `ID` attributes. When SignTool encounters one, it calculates a hash value (something like a checksum) on the content of the script. For a `<SCRIPT>` tag set, it is for the entire content of the tag set; for an event handler, it is for the event handler text. The hash value is associated with the ID attribute label and stored inside the JAR file. After the document loads into the client’s browser, the browser also scans for the ID attributes and performs the same hash calculations on the script items. Then the browser can compare the ID/hash value pairs against the list in the JAR file. If they match, then the file has arrived without being modified by a Bad Guy (or a dropped bit in the network).

Most examples show `ID` attribute values to be numbers, but the attributes are actually strings. No sequence or relationship exists among `ID` attribute values: you can use the names of your favorite cartoon show characters, as long as no two `ID` attributes are given the same name. The only time the same `ID` attribute value may appear in a document is if another JAR file is embedded within the main JAR file. Even so, I recommend avoiding reusing names inside the same HTML file, no matter how many JAR files are embedded.

With one exception, each script item in a document must have its own ~~ID~~ attribute. The exception is a `<SCRIPT>` tag that specifies a `SRC` attribute for an external `.js` file. That file is part of the JAR file, so the browser knows it’s a signed script.

For other `<SCRIPT>` tags, include the ~~ID~~ attribute anywhere within the opening tag, as follows:

```
<SCRIPT LANGUAGE=”JavaScript” ID=”3”>
    statements
</SCRIPT>
```

For a function handler, the ~~ID~~ attribute comes after the event handler inside the object tag, as follows:

```
<INPUT TYPE=”button” VALUE=”Calculate” onClick=”doCalc(this.form)” ID=”bart”>
```

And for a JavaScript entity, the ~~ID~~ attribute must be specified in an empty `<SCRIPT>` tag immediately before the tag that includes the entity for an attribute value, as follows:

```
<SCRIPT ID=”20”>
<INPUT TYPE=”text” NAME=”date” VALUE=&{getToday()};>
```

Listing 46-1 shows a skeletal structure of a document that references a single JAR file and includes five signed scripts: One external `.js` file and four script items in the document itself. The `fetchFile()` function invokes a function imported from `access.js`. Notice that the `ARCHIVE` attribute appears in the very first `<SCRIPT>` tag in the document. This also happens to be a tag that imports an external `.js` file, so that no `ID` attribute is required. If there were no external library file for this page, the `ARCHIVE` attribute would be located in the main `<SCRIPT>` tag, which also has the ID attribute. I arbitrarily assigned increasing numbers as the `ID` attribute values, but I could have used any identifiers. Notice, too, that each script has its own ID value. Just because an event handler invokes a function in a `<SCRIPT>` tag that has an ID value doesn’t mean a relationship exists between the ID attribute values in the `<SCRIPT>` tag and in the event handler that invokes a function there.

Listing 46-1
Basic Signed Script Structure

```
<HTML>
<HEAD>
<TITLE>Signed Scripts Testing</TITLE>
<SCRIPT LANGUAGE="JavaScript" ARCHIVE="myArchive.jar" SRC="access.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript" ID="1">
function fetchFile(form) {
   form.output.value = getFile()
}
function newRaisedWindow() {
   // statements for this function
}
</SCRIPT>
</HEAD>
<BODY>
A Source Code Example Only
<FORM>
<TEXTAREA NAME="output" COLS=60 ROWS=10 WRAP="virtual"></TEXTAREA><BR>
<INPUT TYPE="button" VALUE="Read File" onClick="this.form.output.value = '';
fetchFile(this.form);" ID="2"><BR>

<TEXTAREA NAME="input" COLS=60 ROWS=10 WRAP="virtual"> </TEXTAREA><BR>
<INPUT TYPE="button" VALUE="Save File"
onClick="setFile(this.form.input.value);" ID="3"><P>
<INPUT TYPE="button" VALUE="New Window…" onClick="newRaisedWindow();" ID="4">
</FORM>
</BODY>
</HTML>
```

Editing and moving signed scripts

The nature of the script signing process requires that even the slightest modification you make to a signed script source code requires re-signing the page. For this reason, enabling codebase principals while you create and debug your early code is a convenient alternative.

The rigid link between the hash value of a script element at both the signing and visitor loading times means that you must exercise care when shifting an HTML file that contains signed script elements between servers of differing operating systems. Windows, UNIX, and Macintosh have different ways of treating carriage returns. If you change the representation of an HTML source file when you move the source from, say, a Windows machine to a UNIX server, then the signature may no longer work. However, if you perform a purely binary transfer of the HTML files, every byte is the same, and the signature should work. This operating system-specific text representation affects only how files are stored on servers, not how various client platforms interpret the source code.

Accessing Protected Properties and Methods

For the browser to allow access to protected properties or methods, it must have its privileges enabled. Only the user can grant permission to enable privileges, but it is up to your code to request those privileges of the user.

Gaining privileges

NN4+ comes with some Java classes that allow signed scripts and other signed objects to display the privilege request alert windows, and then turn on the privileges if the user clicks the “OK” or “Grant” button. A lot of these classes show up in the `netscape.security` package, but scripters only work directly with one class and three of its methods:

```
netscape.security.PrivilegeManager.enablePrivilege([“targetName”])
netscape.security.PrivilegeManager.revertPrivilege([“targetName”])
netscape.security.PrivilegeManager.disablePrivilege([“targetName”])
```

The `enablePrivilege()` method is the one that displays the security alert for the user. In NN4, the specific target named as a parameter influenced the details of the security alert message; for NN6, the security alert is generic (and far less intimidating).

If the user grants the privilege, script processing continues with the next statement. But if the user denies access, then processing stops, and the PrivilegeManager class throws a Java exception that gets displayed as a JavaScript error message. Later in this chapter I show you how to gracefully handle the user’s denial of access.

Enabling a privilege in JavaScript is generally not as risky as enabling a Java applet. The latter can be more easily hijacked by an alien class to piggyback on the trusted applet’s privileges. Even though the likelihood of such activity taking place in JavaScript is very low, turning privileges off after the statement that requires privileges is always a good idea. Use the `revertPrivilege()` method to temporarily turn off the privilege; another statement that enables the same privilege target will go right ahead without asking the user again. Disable privileges only when the script requiring privileged access won’t be run again until the page reloads.

Specifying a target

Rather than opening blanket access to all protected capabilities in one blow, the Netscape security model defines narrow capabilities that are opened up when privileges are granted. Each set of capabilities is called a target. Netscape defines dozens of different targets, but not all of them are needed to access the kinds of methods and properties available to JavaScript. You will likely confine your targets to the nine discussed here.

Because NN4's security alerts provided (at times excruciating) detail about the nature of the privilege being requested by the Web site, targets had various risk levels and categories. These concerns are less of an issue in NN6, but they are provided here for your more complete understanding of the mechanisms beneath the Privilege Manager.

Each target has associated with it a risk level (low, medium, or high) and two plain-language descriptions about the kinds of actions the target exposes to code. This information appears in the NN4 security privilege dialog box that faces a user the first time a particular signature requests privileges. All of the targets related to scripted access are medium or high risk, because they tend to open up local hard disk files and browser settings.

Netscape has produced two categories of targets: primitive and macro. A *primitive target* is the most limited target type. It usually confines itself to either reading or writing of a particular kind of data, such as a local file or browser preference. A *macro target* usually combines two or more primitive targets into a single target to simplify the user experience when your scripts require multiple kinds of access. For example, if your script must both read and write a local file, it could request privileges for each direction, but the user would be presented with a quick succession of two similar-looking security dialog boxes. Instead, you can use a macro target that combines both reading and writing into the privilege. The user sees one security dialog box, which, in NN4, explains that the request is for both read and write access to the local hard disk.

Likely targets for scripted access include a combination of primitive and macro targets. Table 46-1 shows the most common script-related targets and the information that appears in the security dialog box.

For each call to `netscape.security.PrivilegeManager.enablePrivilege()`, you specify a single target name as a string, as in

```
netscape.security.PrivilegeManager.enablePrivilege(“UniversalBrowserRead”)
```

This specification allows you to enable, revert, and disable individual privileges as required in your script.

Table 46-1
Scripting-related Privilege Targets

Target Name		
Risk	**Short Description**	**Long Description**
`Universal BrowserAccess`		
High	Reading or modifying browser data	Reading or modifying browser data that may be considered private, such as a list of Web sites visited or the contents of Web forms you may have filled in. Modifications may also include creating windows that look like they belong to another program or positioning windows anywhere on the screen.
`Universal BrowserRead`		
Medium	Reading browser data	Access to browser data that may be considered private, such as a list of Web sites visited or the contents of Web page forms you may have filled in.
`Universal BrowserWrite`		
High	Modifying the browser	Modifying the browser in a potentially dangerous way, such as creating windows that may look like they belong to another program or positioning windows anywhere on the screen.
`Universal FileAccess`		
High	Reading, modifying, or deleting any of your files	This form of access is typically required by a program such as a word processor or a debugger that needs to create, read, modify, or delete files on hard disks or other storage media connected to your

		computer.
`Universal FileRead`		
High	Reading files stored in your computer	Reading any files stored on hard disks or other storage media connected to your computer.
`Universal FileWrite`		
High	Modifying files stored in your computer	Modifying any files stored on hard disks or other storage media connected to your computer.
`Universal PreferencesRead`		
Medium	Reading preferences settings	Access to read the current settings of your preferences.
`Universal PreferencesWrite`		
High	Modifying preferences settings	Modifying the current settings of your preferences.
`Universal SendMail`		
Medium	Sending e-mail messages on your behalf	

Blending Privileges into Scripts

The implementation of signed scripts in Navigator protects scripters from many of the potential hazards that Java applet and plug-in developers must watch for. The chance that a privilege enabled in a script can be hijacked by code from a Bad Guy is very small. Still, exercising safe practices in case you someday work with other kinds of signed objects is good practice.

Keep the window small

Privilege safety is predicated on limiting exposure according to two techniques. The first technique is to enable only the level of privilege required for the protected access your scripts need. For example, if your script only needs to read a normally protected `document` object property, then enable the `UniversalBrowserRead` target rather than the wider `UniversalBrowserAccess` macro target.

The second technique is to keep privileges enabled only as long as the scripts need them enabled. If a statement calls a function that invokes a protected property, enable the privilege for that property in the called function, not at the level of the calling statement. If a privilege is enabled inside a function, the browser automatically reverts the privilege at the end of the function. Even so, if the privilege isn't needed all the way to the end of the function, you should explicitly revert it after you are through with the privilege.

Think of the users

One other deployment concern focuses more on the user's experience with your signed page. You should recognize that the call to the Java `PrivilegeManager` class is a LiveConnect call from JavaScript in NN4. Because the Java virtual machine does not start up automatically when Navigator 4 does, a brief delay occurs the first time a LiveConnect call is made in a session (the statusbar displays "Starting Java..."). Such a delay may interrupt the user flow through your page if, for example, a click of a button needs access to a privileged property. Therefore, consider gaining permission for protected access as the page loads. Execute an `enablePrivilege()` and `revertPrivilege()` method in the very beginning. If Java isn't yet loaded into the browser, the delay is added to the other loading delays for images and the rest of the page. Thereafter, when privileges are enabled again for a specific action, neither the security dialog box nor the startup delay get in the way for the user.

Also remember that users don't care for security dialog boxes to interrupt their navigation. If your page utilizes a couple of related primitive targets, at the outset enable the macro target that encompasses those primitive targets. The user gets one security dialog box covering all potential actions in the page. Then let your script enable and revert each primitive target as needed.

Example

To demonstrate signed scripts in action, I show a page that accesses a typical target that allows the script to open an always-raised new window. No error checking occurs for the user's denial of privilege in this example. Therefore, if you experiment with this page (either with codebase principals turned on or signing them yourself), you will see the JavaScript error that displays the Java exception. Error detection is covered later in the chapter.

Accessing a protected window property

Listing 46-2 is a small document that contains one button. The button calls a function that opens a new window with the NN-proprietary `alwaysRaised` parameter turned on. Setting protected `window.open()` parameters in NN4+ requires the `UniversalBrowserWrite` privilege target. Inside the function, the privilege is enabled only for the creation of the new window. For this simple example, I do not enable the privilege when the document loads.

Listing 46-2
Creating an AlwaysRaised Window

```
<HTML>
<HEAD>
<TITLE>Simple Signed Script</TITLE>
<SCRIPT LANGUAGE="JavaScript" ARCHIVE="myJar.jar" ID="1">
function newRaisedWindow() {
    netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite")
    var newWindow = window.open("","","HEIGHT=100,WIDTH=300,alwaysRaised")

netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite")
    var newContent = "<HTML><BODY><B>It\'s good to be the King!</B>"
    newContent += "<FORM><CENTER><INPUT TYPE='button' VALUE='OK'"
    newContent += "onClick='self.close()'></CENTER></FORM></BODY></HTML>"
    newWindow.document.write(newContent)
    newWindow.document.close()
}
</SCRIPT>
</HEAD>
<BODY>
<B>This button generates an always-raised new window.</B>
<FORM>
<INPUT TYPE="button" VALUE="New 'Always Raised' Window"
onClick="newRaisedWindow()" ID="2">
</BODY>
</HTML>
```

Listing 46-2 has two script items that need signing: the `<SCRIPT>` tag and the event handler for the button. Also, the `ARCHIVE` attribute points to the JAR file that contains the script signature. Note that this example file is not signed, and therefore does not include a companion JAR archive on the companion CD-ROM.

Handling Privilege Manager Errors

The change between the ways NN4 and NN6 allows scripts to intercept errors causes no small problem if you need to serve both browser versions. The primary reason you want to handle errors is that when a user denies access to advanced privileges, the `PrivilegeManager` generates an error. While the error is not destructive in any way, and it appears only in the JavaScript Console window (NN4.5+), accounting for such factors is good coding practice. Unfortunately, the mechanism that works for NN4 doesn’t work in NN6; the mechanism that works in NN6 cannot even be placed in a page that loads into NN4 without generating syntax errors. The bottom line is that you need to serve up different pages for NN4 and NN6 until such time as the NN4 installed base drops away.

For NN4, you can define an `onerror()` function that looks for the specific error message thrown by the `PrivilegeManager` class through LiveConnect. That function looks as the following:

```
function onerror(msg, URL, lineNum) {
    var errorMsg = msg
    if (msg.indexOf("ForbiddenTargetException") != -1) {
        errorMsg = "You have elected not to grant privileges to this script."
    }
    alert(errorMsg)
    return true
}
```

Of course, you don't have to display any message, but it may be a good place to advise users about what they're missing by not granting privilege.

For NN6, you can use the native `try...catch` exception handling, which means that the calls to the `enablePrivilege()` method of the `PrivilegeManager` class must be wrapped inside a `try` block. The function from Listing 46-2 is modified as follows:

```
function newRaisedWindow() {
    try {

netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite")
    }
    catch(err) {
        alert("You have elected not to grant privileges to this script.")
        return
    }
    var newWindow = window.open("","","HEIGHT=100,WIDTH=300,alwaysRaised")

netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserWrite")
    var newContent = "<HTML><BODY><B>It\'s good to be the King!</B>"
    newContent += "<FORM><CENTER><INPUT TYPE='button' VALUE='OK'"
    newContent += "onClick='self.close()'></CENTER></FORM></BODY></HTML>"
    newWindow.document.write(newContent)
    newWindow.document.close()
    return
}
```

Signed Script Miscellany

In this last section of the chapter, I list some of the more esoteric issues surrounding signed scripts. Three in particular are: 1) how to allow unsigned scripts in other frames, windows, or layers to access signed scripts; 2) how to make sure your signed scripts are not stolen and reused; and 3) special notes about international text characters.

Exporting and importing signed scripts

JavaScript provides an escape route that lets you intentionally expose functions from signed scripts for access by unsigned pages. If such a function contains a trusted privilege without careful controls on how that privilege is used, a page that is not as well intentioned as yours could hijack the trust.

The command for exposing this function is export. The following example exports a function named `fileAccess()`:

```
export fileAccess
```

A script in another window, frame, or layer can use the import command to bring that function into its own set of scripts:

```
import fileAccess
```

Even though the function is now also a part of the second document, it executes within the context of the original document, whose signed script governs the privilege. For example, if you exported a function that did nothing but enable a file access privilege, a Bad Guy who studies your source code could write a page that imports that function into a page that now has unbridled file access.

If you wish to share functions from signed scripts in unsigned pages loaded into your own frames or layers, avoid exporting functions that enable privileges. Other kinds of functions, if hijacked, can’t do the same kind of damage as a privileged function can.

Locking down your signed pages

Speaking of hijacking scripts, it would normally be possible for someone to download your HTML and JAR archive files and copy them to another site. When a visitor comes to that other site and loads your copied page and JAR file, your signature is still attached to the scripts. While this may sound good from a copyright point of view, you may not want your signature to appear as coming from someone else’s Web server. You can, however, employ a quick trick to ensure that your signed scripts work only on your server. By embedding the domain of the document in the code, you can branch execution so that scripts work only if the file comes from your server.

The following script segment demonstrates one way to employ this technique:

```
<SCRIPT LANGUAGE=”JavaScript1.2” ARCHIVE=”myPage.jar” ID=”1”>
if (document.URL.match(/^http:\/\/www.myDomain.com\//)) {
    privileges statements execute only from my server
}
</SCRIPT>
```

This technique works only if you specify JavaScript 1.2 as the script language. Even though this branching code is visible in the HTML file, the hash value of your code is saved and signed in the archive. If someone modifies the HTML, the hash value that is recalculated when a visitor loads the page won’t match the JAR file manifest and the script signature fails.

International characters

While international characters are fine for HTML content, they should not be used in signed scripts. The problem is that international characters are often converted to other character sets for display. This conversion invalidates the signature, because the signed and recalculated hash values don’t match. Therefore, do not put international characters in any signable script item. If you must include such a character, you can escape it, or, in NN4, put such scripts in unsigned layers.