CHAPTER 5

Browser Security Principles: The Same-Origin Policy

We'll Cover

- Defining the same-origin policy
- Exceptions to the same-origin policy

Many of the security principles we've talked about and will talk about in this book deal with protecting your server resources. You want to be sure that unauthorized users can't access the system; you want to be sure that attackers can't break into your databases and steal your data; you want to be sure that no one can slow your system down so that it's unavailable for everyone else. And it's vital that you do defend against these kinds of attacks, but this chapter will focus on a completely different yet equally important area; specifically, we'll be focusing on the principles of browser security.

Web browsers have controls built into them in order to prevent malicious web sites from stealing users' personal data. Browsers restrict the ways that web pages can communicate with web servers and with other windows in the browser. However, they only slightly restrict these communications—they couldn't block them completely, or the web would be a pretty boring place. Usually, these subtle limitations are enough to thwart would-be attackers. But if a web application has certain flaws in its code—sometimes very small flaws that are very easy to overlook—then all bets are off, and attackers can completely bypass the inherent protections that browsers offer their users. The next two chapters provide a detailed look at both the browser protections that we're supposed to have, and the vulnerabilities and exploits that attackers use to negate them.

It makes sense to start our examination of browser security with a discussion of the same-origin policy, since the same-origin policy is essentially the foundation of most browser security principles. Without it, any site on the Internet could access the confidential user information of any other site. All of the attack techniques we'll be talking about in the next chapter, like cross-site scripting and cross-site request forgery, are essentially ways for attackers to bypass the inherent defense of the same-origin policy. Unfortunately, the attackers are sometimes unwittingly aided in their attempts by the application developers themselves. But before we get into that, we need to define exactly what the same-origin policy is and how it works.

Defining the Same-Origin Policy

The same-origin policy is essentially an agreement among browser manufacturers—mainly Microsoft, Apple, Google, Mozilla and Opera—on a standard way to limit the functionality of scripting code running in users' web browsers. You might wonder why this is a good thing and why we would want any limits on scripting functionality. If so, don't worry; we'll go into this in detail in the next section. Until then, please trust us that without the same-origin policy, the World Wide Web would be more like a Wild West Web where anything would go, no data would be safe, and you'd never even think about using a credit card to buy something there.

In short, the same-origin policy states that when a user is viewing a web page in his browser, script running on that web page should only be able to read from or write to the content of another web page if both pages have the same "origin." It goes on to define "origin" as a combination of the page's application layer protocol (HTTP or HTTPS), its TCP port (usually 80 for HTTP or 443 for HTTPS), and its domain name (that is, "www .amazon.com" or "www.mhprofessional.com").

This concept is easier to explain with an example. Let's say you're getting bored with your desktop background photo and you want to look for a new one, so you open your browser to the page http://www.flicker.cxx/galleries/. If there were any scripting code running on this page, which other web pages could that script read from?

- **https://www.flicker.cxx/galleries/**
  No, the script could not read from this page: the protocols between the two pages are different (HTTP vs. HTTPS).
- **http://www.photos.cxx/galleries**
  www.photos.cxx and www.flicker.cxx are completely different domains. The script could not read from this page.
- **http://my.flicker.cxx/galleries/**
  This one is a little trickier, but it still won't work: my.flicker.cxx is not the same domain as www.flicker.cxx.
- **http://flicker.cxx/galleries/**
  This is another tricky one, but removing part of the domain name does not change the fact that the domains are different. This still won't work.
- **http://mirror1.www.flicker.cxx/galleries/**
  Adding to the domain name doesn't change the fact either. The script could not read from this page.
- **http://www.flicker.cxx:8080/galleries/**
  The answer to this one is actually a little complicated. If you're using any browser besides Microsoft's Internet Explorer (IE), then the script could not read from that page, because the TCP ports between the two pages are different. (Remember that for

HTTP pages, if you don't explicitly specify a port, the default value is 80. So http://www.flicker.cxx/ is actually the same page as http://www.flicker.cxx:80/.) However, IE acts a little differently from the other major browsers: IE does not use a page's port as part of its origin; instead, it uses the page's defined security zone, which is either "Internet," "Local intranet," "Trusted site," or "Restricted site." The user can configure which sites fall into which zones, so if you're using IE, then the script could potentially read from this page, depending on the user's setup.

- **http://www.flicker.cxx/favorites**
  In this case, all three of the important URL attributes are the same: the protocol (HTTP), the port (80), and the domain (www.flicker.cxx). So the answer is "yes," the script could read from this page. The fact that it's a different directory doesn't make any difference to the same-origin policy.

Table 5-1 shows the results of attempting a scripting request from http://www.flicker.cxx/galleries/ to specific URLs.

An Important Distinction: Client-Side vs. Server-Side

It's important to note that the same-origin policy has absolutely no effect on what pages or sites any server-side code can access. The server at www.flicker.cxx is free to make requests to my.flicker.cxx, mirror1.www.flicker.cxx, google.com, or even any intranet sites that the company may have that aren't directly accessible via the Internet. The same-origin policy only applies to browsers running client-side scripting code. You might be wondering what the difference is, and why browsers would go to the trouble of implementing a restriction like this when there are no restrictions at all for the server code. The answer is simple: cookies.

URL	Scripting Requests Allowed?
https://www.flicker.cxx/galleries/	No, different protocol (HTTP vs. HTTPS)
http://www.photos.cxx/galleries/	No, different domain
http://my.flicker.cxx/galleries/	No, different domain
http://flicker.cxx/galleries/	No, different domain
http://mirror1.www.flicker.cxx/galleries/	No, different domain
http://www.flicker.cxx:8080/galleries/	No, different port (except on IE browsers)
http://www.flicker.cxx/favorites/	Yes

Table 5-1 The Same-Origin Policy as Applied to a Hypothetical Page http://www.flicker.cxx/galleries/

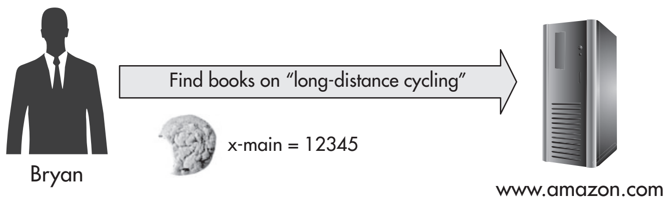

Figure 5-1 Every request I make to www.amazon.com automatically includes the cookie "x-main."

As we saw in the last chapter, any time you visit a page in a web browser, the browser automatically sends all the cookies it's saved for that site along with your request for the page. So, for example, when I visit www.amazon.com, my browser sends a cookie back with my request (in this particular case, the cookie's name is "x-main") with a value that uniquely identifies me to Amazon (let's say the value is "12345"). You can see a diagram of this in Figure 5-1.

Since I'm the only person in the world with a www.amazon.com x-main cookie value of "12345," the Amazon server knows that I'm the person visiting the site, and it personalizes the content accordingly. This is why when I visit Amazon, I get a completely different page than when you visit Amazon. It shows a banner with my name ("Hello Bryan. We have recommendations for you."), it shows that I'm a member of their Amazon Prime free shipping club, and since I've recently ordered books on Objective-C programming, it shows me pictures and information for other programming books I might want to buy.

However, if I set up a web application (for example, "www.bryanssite.cxx") and program its server-side code to call out to www.amazon.com, the response it gets back from Amazon won't have any of my personal information, because the request it sends out won't have my cookie value. Figure 5-2 shows this in action. Even if I'm the one making the request to www.bryanssite.cxx, my browser is only going to send the cookies for

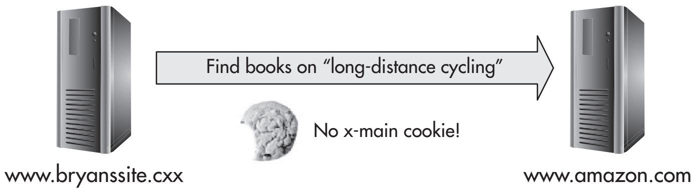

Figure 5-2 Requests made from server to server don't include the user's cookies for the target web site.

www.bryanssite.cxx. It won't send the cookies for www.amazon.com or www.google.com or for any other site.

So the main point of the same-origin policy is not to prevent web applications from reading resources from other sites, but rather to prevent web applications from reading personalized (or more specifically, *credentialed*), potentially sensitive and private resources from other sites. Now that we have a clearer understanding of what the same-origin policy is, let's take a closer look at why we need it.

A World Without the Same-Origin Policy

The one thing that application developers and attackers may have in common is a shared loathing of the same-origin policy, and for exactly the same reason: it keeps them from getting to other sites' data. Their reasons for wanting this data may vary: the attacker wants data so he can sell it, and the developer may just have an idea for creating a new *mashup* (a web site combining the functionality of two or more other sites; for example, TwitterVision is a mashup of Twitter and Google Maps that shows you where tweets are coming from in real time). But for better *and* worse, the same-origin policy gets in the way in both cases.

To demonstrate what we mean about both developers and attackers hating the same-origin policy, let's imagine a world without it. In this world, an online florist—Amy's Flowers—is building their new web site. Amy's Flowers knows that a lot of people buy flowers for their mother for her birthday. So whenever you visit www.amysflowers.cxx, the client-side page script makes a request to get your personalized page at http://calendar.google.com. The script looks through the Google Calendar page contents to see if you have an upcoming event titled something like "Mom's Birthday."

If the script finds your mom's birthday, its next step is to check your e-mail to see if maybe you forgot about it last year. It makes requests to gmail.com, mail.yahoo.com, and hotmail.com to see if it gets back personalized pages from any of those services. If so, the script looks through the mailbox data in those pages for messages from last year around your mom's birthday containing words like "disappointed" or "hurt" or "neglected."

The script's final step is to check to see how much money you have in your bank account, so that Amy's Flowers can offer you an appropriately priced bouquet for your budget. Again, the script makes requests to Bank of America, Chase, and Wells Fargo to look for personalized page responses. It sees that you've just paid your car payment, so you don't have much money in your account right now, but it also sees that you're due to get paid in the next week and you'll have more money then. Now Amy's Flowers has everything it needs to offer you a completely personalized, full-service shopping experience:

"Welcome to Amy's Flowers, Bryan! Did you remember that your mother's birthday is coming up on the 23rd? Since you forgot last year, you might want to splurge a little on the extra-large Bouquet of Magnificence. Don't worry; we'll wait to bill you until Friday when your paycheck cashes. Surely a mother's love is worth at least as much as that new Porsche Boxster?"

By this point, I'm sure you can see both the good side and the bad of the same-origin policy. A world without it would undoubtedly have some amazing web applications, but at an enormous cost to both privacy and security.

Exceptions to the Same-Origin Policy

However, even here in the real world, there are ways for applications to make exceptions to the same-origin policy. The demand from web application developers for the ability to create full-featured mashups like Amy's Flowers—but significantly more secure—is just too great to ignore, so browser manufacturers and browser plugin manufacturers have given them the ability to bypass the same-origin policy in certain controlled ways. But if you accidentally misuse these bypasses, you can leave your application open to attack and leave your users at risk of losing their private data. Let's look at a few ways to get around the same-origin policy, and a few ways to ensure this is done as securely as possible.

HTML ‹script› Element

The HTML <script> element is undoubtedly the most widely used method of cross-origin communication. If you open just about any page on the Internet, chances are good that you'll find a <script> tag in its page code somewhere. What <script> does is to define a block of client-side script code, usually written in JavaScript. For example, the following <script> block will pop up an alert box in the user's browser window with the message "Hello JavaScript World!":

```
<script>
  alert('Hello JavaScript World!');
</script>
```

Instead of defining a block of script code directly within the <script> tags, you can also use the script element's "src" attribute to load script code from some other location on the Internet:

```
<script src="http://www.site.cxx/some_script.js"/>
```

This is where the cross-origin communication comes in, because the same-origin policy is not enforced for <script src> tags. Any page is free to load and run script code from anywhere on the Internet.

You should be extremely cautious when using <script src> in your applications unless you have complete control over the script that the tag is loading. If you're the owner of www.siteA.cxx, and you're using <script src> to load from some location on siteA.cxx, you're probably okay:

```
<script src="http://siteA.cxx/script.js" />
```

But be careful when pointing <script src> at other domains you don't own:

```
<script src="http://siteB.cxx/script.js" />
```

Whenever a user visits your page that contains this tag, their browser will automatically download the script code from siteB.cxx and run it. If the script code turns out to be malware, it will look to the user as if it was your page that was responsible. And even if siteB.cxx is a completely honest site that would never knowingly host malware, if they get hacked, then so will you and so will your users.

JSON and JSONP

There is one big catch to using <script src>: While you can point a <script> tag at any site on the Internet without regard to the same-origin policy, it only works if the resource you specify in the "src" attribute is well-formed, valid script. If you try to point it at any arbitrary HTML page, your page code will throw an error and you won't be able to view the cross-domain data you tried to retrieve. It must be script, valid script, and only valid script.

In the early 2000s, Douglas Crockford, currently a software architect at Yahoo!, realized that it would be possible to create a data format that would also be valid JavaScript code. He called this format *JSON*, short for JavaScript Object Notation. If you wanted to create a JSON object to represent a music album, it might look something like this:

```
{
  "artist" : "The Black Keys",
  "album" : "Brothers",
  "year" : 2010,
  "tracks" : [ "Everlasting Light", "Next Girl", "Tighten Up"]
}
```

Many web services now use JSON as their data format instead of XML because JSON is generally more compact, more human-readable, and because JSON objects are also valid JavaScript objects and are easy to work with from JavaScript code. However, web services send data as strings, not native JavaScript objects. If you get a string of JSON from a web service, maybe as the result of an Ajax call (if you're unfamiliar with Ajax, we'll cover it later in this section), you'll need to convert it into an object before you can work with it. There are several ways to do this, but only one of them is really safe.

One popular but insecure way is to use the JavaScript function "eval" to evaluate the JSON string (that is, execute it as if it were code) and create an object from it:

```
jsonString = '{"artist":"The Black Keys","album":"Brothers"}';
var album = eval('(' + jsonString + ')');
```

This is highly dangerous unless you have complete control over the JSON string being eval'd. If an attacker has any way to add or change items in the JSON, he could potentially add malware to it, which the user's browser would then execute when it evals the JSON.

Tip

"eval" is not the only JavaScript function that can execute arbitrary, potentially malicious string values. Other less well-known, but equally dangerous, equivalents include "setTimeout" and "setInterval." You should avoid these just as you should avoid "eval."

A better alternative to "eval" is to use the native JavaScript function JSON.parse, which is much more secure and won't execute malicious script. JSON.parse is available in IE as of IE8, Firefox as of version 3.5, Safari as of 4.0, and all versions of Chrome. If you need to support browsers older than this, it's still better to avoid "eval" and instead use one of the free JavaScript libraries like jQuery or Prototype that can safely parse JSON.

At this point, you might be wondering why you have to go to the trouble of parsing JSON, and why you can't just use <script src> to fetch JSON data directly. After all, JSON is valid script code, so <script src> should be able to work with it just fine. Actually, you can do this, but there are some reasons you might not want to. In the first place, using <script src> is essentially the same thing as calling "eval," so unless you can completely trust the source you're pulling data from, you could be putting yourself at risk. Second, unless the site serving the JSON supports a callback mechanism called JSONP, you won't be able to do anything with the returned data. And third, even if they do, it's still somewhat dangerous to use.

The reason you wouldn't be able to do anything with straight JSON if you tried to get it with <script src> is that while it's a valid JavaScript object, it's just that: an object. It doesn't have a name, or any other way to access it in the script. In fact, since the script has no other references to the new JSON object, it's likely to just be immediately deleted (or "garbage collected") by the script engine.

Application developers sometimes work around this problem by using a variation of JSON called *JSONP*, short for JSON with Padding. JSONP is similar to JSON, except that the data is wrapped in (or "padded with") a JavaScript function call that's usually specified by the client requesting the data. For example, if you make a request to a music catalog web service and specify that you want the returned data wrapped in the function "displayAlbumInfo," the JSON you get back might look something like this:

```
displayAlbumInfo({"artist":"The Black Keys","album":"Brothers"});
```

All you have to do now is to implement a displayAlbumInfo function in your script code. Now when you point a <script src> tag at the JSONP-returning music service, your displayAlbumInfo callback function will automatically execute and you can do whatever you want with the returned JSONP data.

Just as in our previous discussion of JSON and "eval," you have to trust the JSONP source not to return malicious script code to you, but there's also another reason you might want to avoid JSONP. Let's switch places for a minute and say that you're the one who's actually *serving* the JSONP data instead of the one who's requesting it. You already know that the same-origin policy doesn't apply to JSONP; that's why you're using it in the first place. But without the protection of the same-origin policy, there's nothing to keep an attacker from getting the data either. If you're not clear on how an attack like this would work, we'll discuss it in more detail later in this chapter when we talk about cross-site request forgery. For now, just keep in mind that we recommend against serving data in JSONP form, except when that data is completely public and can safely be viewed by anybody.

iframes and JavaScript document.domain

Another way for web pages to communicate across different origins is through the use of frame elements like <frame>, <frameset>, and <iframe> that open views to other web pages. For example, your web page at www.siteA.cxx/welcome.html could open a 300-by-300 pixel frame to the page www.siteB.cxx/home.html like this:

```
<iframe
         src=http://www.siteB.cxx/home.html
         width="300px"
         height="300px">
</iframe>
```

Figure 5-3 shows the web site www.amysflowers.cxx opening an iframe to www.amazon.com. It's important to note that the same-origin policy doesn't prevent you from opening frames to any other sites—you can point your frames anywhere on the Internet, and they'll work just fine. However, it does prevent you from using script code to read the

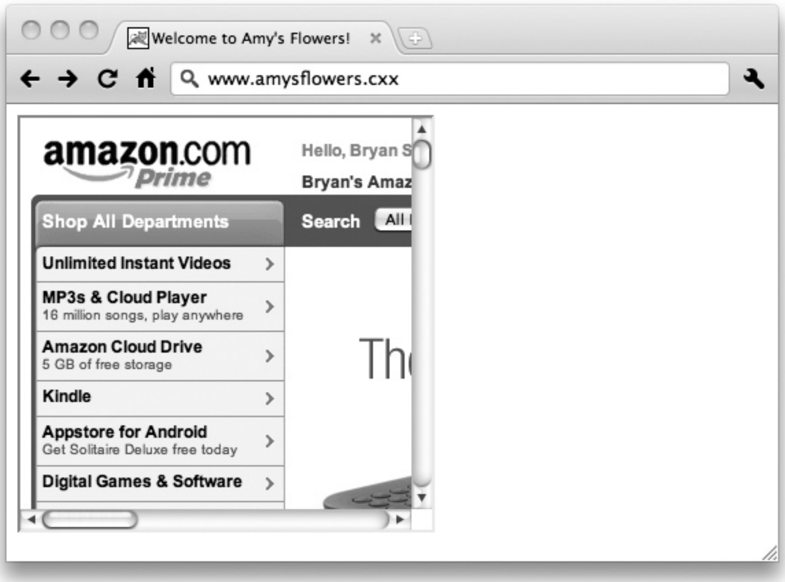

Figure 5-3 The web site www.amysflowers.cxx includes a frame pointing to www.amazon.com.

contents of frames loaded from different origins, or to change their contents. This would be a clear security violation: amysflowers.cxx could open an iframe to yourbank.cxx and read your bank statement.

Under certain circumstances, though, two pages from different origins can cooperate to allow their contents to be read and changed through frames. Every page has a "domain" property that shows the site the page was loaded from. So the domain property value for the page www.siteA.cxx/welcome.html would be www.siteA.cxx. You can read this value with the JavaScript property document.domain. However, you can use document.domain not just to read a page's domain, but also to change it.

Normally, the page foo.siteA.cxx/home.html could not read the contents of the page bar.siteA.cxx/home.html from an iframe; remember that foo.siteA.cxx and bar.siteA.cxx are considered different origins by the same-origin policy. You can see an example of this in Figure 5-4. But if both pages agree to lower their domain value to just "siteA.cxx," then they will be able to read each other's contents in iframes. In Figure 5-5 you can see the difference once both pages have lowered their domain values.

```
<script type="javascript">
  document.domain = 'siteA.cxx';
</script>
```

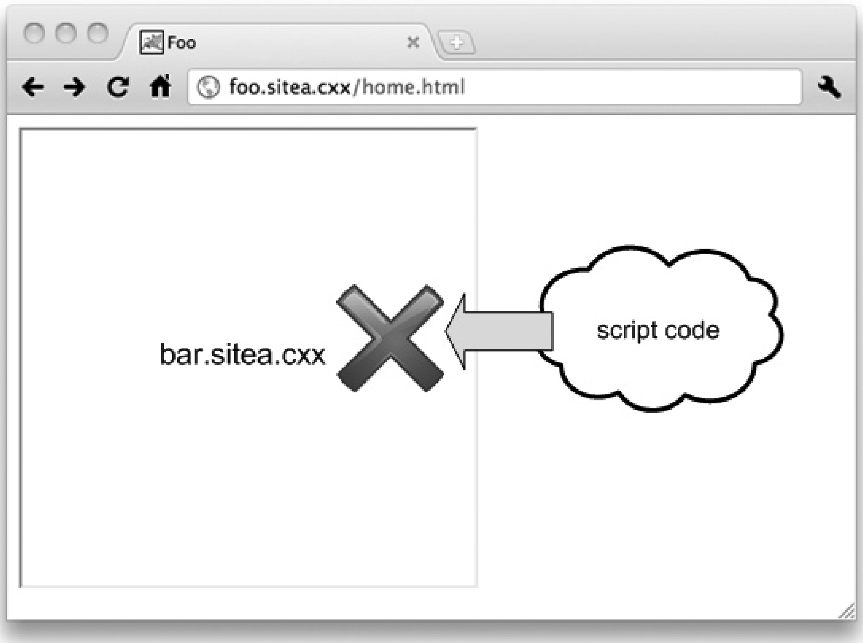

Figure 5-4 Script code running on foo.siteA.cxx is unable to access the contents of an iframe pointing to bar.siteA.cxx due to the same-origin policy.

You can only use this method to change the domain value to a subdomain of the original; for example, you can't go from "www.siteA.cxx" to "www.siteB.cxx." And you can't go lower than the base domain for the site, so setting it to just ".cxx" isn't allowed either.

It's also important that both pages explicitly set their domain property to the new common value. Even if you just want your page at www.siteA.cxx/page.html to read an iframe from siteA.cxx/page.html, it's not enough just to change document.domain in www.siteA.cxx/page.html; you'll also need to explicitly set it in siteA.cxx/page.html too. This can lead to weird-looking code like this:

```
<script type="javascript">
  document.domain = document.domain;
</script>
```

It may look like a waste of processing power, but it's necessary to make the process work.

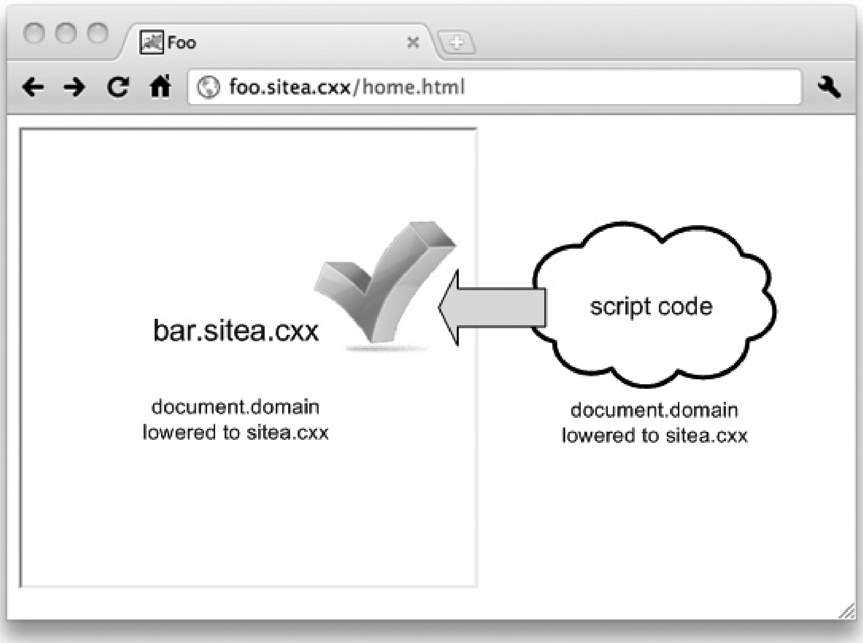
Foo
foo.sitea.cxx/home.html
bar.sitea.cxx
document.domain lowered to sitea.cxx
script code
document.domain lowered to sitea.cxx

Figure 5-5 Script code running on foo.siteA.cxx can now access the contents of an iframe pointing to bar.siteA.cxx once both pages lower their document .domain to siteA.cxx

Note

A little-known fact about document.domain is that, in most browsers, once you lower it, you can't go back up again. If you start off at "www.siteA.cxx" and then change it to just "siteA.cxx," trying to set it back to "www.siteA.cxx" will cause an error. Once you "lower your shields," you can't raise them again. The exceptions to this rule are IE6 and IE7, which do let you reset a page's document.domain value. But given that most other browsers don't allow this, and that IE6 and IE7 are pretty dated at this point (IE9 is the current version of Internet Explorer as of this writing), it's best not to rely on this feature.

Adobe Flash Player Cross-Domain Policy File

Adobe Flash is a programming framework used to create rich Internet applications (RIAs) with more multimedia and user interactivity capabilities than traditional web applications. Flash applications run in the user's browser, just as client-side JavaScript code does. Many popular browser-based games, like Zynga's "Farmville" and Popcap's "Plants vs. Zombies," are written in Flash, and many web ads are also Flash-based.

Flash applications also have the ability to make cross-origin requests, but under much tighter restrictions than web pages using <script src> . In order for a Flash application to

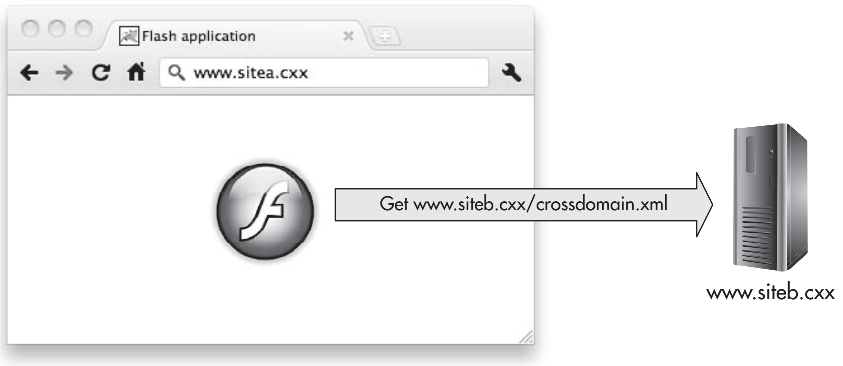

Figure 5-6 The web site www.siteB.cxx uses a Flash cross-domain policy file to allow Flash applications to access it from other origins.

read data from a different origin, the owners of the *target* site—not the Flash application's site—have to put a policy file named crossdomain.xml on their site that details exactly who can make cross-origin requests for their data.

For example, let's say you have a Flash application on your site at http://www.siteA.cxx/flash.swf, and you want this application to be able to load data from the user's personal calendar on the page http://www.siteB.cxx/mycalendar. In order for this to work, the administrator at www.siteB.cxx will have to create a crossdomain.xml file and place it on the root of the web site (that is, http://www.siteB.cxx/crossdomain.xml). Figure 5-6 shows a diagram of this behavior.

However, just because a crossdomain.xml file is present on a site doesn't necessarily mean that the site is completely open to cross-origin requests. Flash cross-domain policies can be configured to only allow cross-origin requests from certain sites. Here's an example of a crossdomain.xml file that will only allow Flash applications loaded from www.siteA.cxx:

```
<cross-domain-policy>
  <allow-access-from domain="www.siteA.cxx" />
</cross-domain-policy>
```

You can list multiple domains and wildcards in a crossdomain.xml policy file too. This example policy allows access from www.siteA.cxx, www.siteB.cxx, and any subdomain of siteC.cxx:

```
<cross-domain-policy>
  <allow-access-from domain="www.siteA.cxx" />
  <allow-access-from domain="www.siteB.cxx" />
  <allow-access-from domain="*.siteC.cxx" />
</cross-domain-policy>
```

It's even possible to allow complete access from the entire Internet:

```
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

Setting up a crossdomain.xml file with widely scoped wildcards like this is usually bad for security. What you're essentially saying when you do this is, "Any site on the Internet can read my users' personal data." The only time this is safe is when your site doesn't actually have users' personal data. If your site is completely open, with no authentication and no personalized information, then it's fine to allow cross-origin access like this. Otherwise, you could be opening your users to the possibility of having their personal data stolen.

Tip

One good strategy for dealing with this is to break your site into separate subdomains, since you can control the cross-origin policy for each subdomain with separate crossdomain.xml files. Let's say you have a photo gallery site, and you want to let users upload both public pictures that the whole world can see, and private ones that shouldn't be shared. To control cross-origin access correctly, put the public picture pages into one subdomain (such as www.photos.cxx) and the private pages into a separate one (such as my.photos.cxx). Now you can add a crossdomain.xml file to the root of www.photos.cxx, and you can make this policy as open as you want; it won't have any effect on the private pages in my.photos.cxx.

One final note about cross-origin communication and Flash is that you can include Flash applications from other domains directly in your web pages; you don't need to copy them to your site. So if a company wants to pay you to host their Flash-based advertisement (say, http://www.adsite.cxx/buynow.swf) on your home page, you can embed a direct reference to that ad's URL in your page. The origin of a Flash application is the origin of the site where it's served from, not the origin of the page that it's embedded in. So the origin for this example Flash ad is http://www.adsite.cxx, not http://www.siteA.cxx.

Microsoft Silverlight

Microsoft also has its own RIA browser plugin, called Silverlight. Cross-origin access in Silverlight works very similarly to cross-origin access in Flash; Silverlight even checks the exact same crossdomain.xml policy files that Flash does. Silverlight will also check for a separate, Silverlight-specific policy file named clientaccesspolicy.xml. Here's an example clientaccesspolicy.xml file that allows access from www.siteA.cxx and www.siteB.cxx:

```
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="www.siteA.cxx" />
        <domain uri="www.siteB.cxx" />
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

Just like crossdomain.xml policy files, clientaccesspolicy.xml files can specify wildcards in their lists of allowed domains:

```
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="*" />
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

But again, be careful when allowing wildcard domain access in either of these policy files: it's probably only safe to do this when your site has no authentication or sensitive personal data.

XMLHttpRequest (Ajax) and Cross-Origin Resource Sharing

Yet another popular way to develop RIA applications is by using the JavaScript XMLHttpRequest object. Just like Flash and Silverlight, XMLHttpRequest allows client-side script code to make requests to other web servers to fetch new data for the page, without having to do a complete page refresh. (You'll often hear this style of programming referred to as *Ajax* programming, which is an acronym for Asynchronous

JavaScript And XML.) One advantage Ajax has over other RIA frameworks is that it's a pure JavaScript feature already built into browsers and doesn't require the user to install a browser plugin. Also, as of this writing, Apple's iOS devices (iPhone, iPad, iPod Touch) don't support Flash or Silverlight, so organizations developing RIAs meant to be used on iOS will need to go with Ajax.

Like Flash and Silverlight, XMLHttpRequest objects are constrained by the same-origin policy. By default, they can only make requests to the same origin of the page they're included on. However, also like Flash and Silverlight, you can selectively disable the same-origin policy for Ajax. While there's no direct Ajax equivalent to a crossdomain.xml policy file, there is an alternative called *cross-origin resource sharing* (CORS).

Instead of policy files, CORS uses HTTP response headers to control cross-origin permissions. If you want your web server to allow Ajax applications from other origins to be able to access it, you can add an "Access-Control-Allow-Origin" HTTP header to your server's responses. Either set the value of this header to "*", which gives access to any origin, or set it to a specific domain like "www.siteA.cxx," which gives access only to that particular site.

Note

As of the current W3C CORS specification, there's no way to set multiple specific domains. You have to either pick just one or pick everything.

Considering all the other warnings and recommendations against using widely accessible cross-domain privileges that we've given you in this chapter, you're probably expecting us to tell you to avoid using the "*" wildcard with CORS. But there's one key difference between CORS and the other cross-origin access methods we've discussed in this chapter that makes CORS much safer than the others. The difference is that by default, browsers will not send users' cookies or other authentication information along with their CORS requests. Without cookies, the server can't identify the user who's making the request, so it can't return sensitive personal data. With no confidentiality threat, there's much less danger.

It is possible to opt into using cookies with CORS, if the client code sets the "withCredentials" property of its XMLHttpRequest object and the server adds an extra "Access-Control-Allow-Credentials : true" header to its response. If you do decide to allow credentialed CORS requests, we would definitely recommend that you not allow "*" wildcard access to your server.

You should also be aware that CORS is not available in every web browser. The browsers that do support it only support it in their recent versions: it's in Safari 4 and later, Firefox 3.5

and later, and Chrome 3 and later. Opera does not support CORS at all, and neither does IE, although IE does have a similar feature called XDomainRequest, which we'll discuss next.

XDomainRequest

In IE, just as in older versions of Safari and Firefox, when you use the XMLHttpRequest JavaScript object to make Ajax calls, you can only call back to the same origin that the page came from. However, IE versions 8 and later support cross-origin calls through a different object named XDomainRequest.

XDomainRequest works very similarly to CORS. Just as in CORS, XDomainRequest cross-origin calls are only allowed if the server responds with an appropriate "Access-Control-Allow-Origin" header. (It's nice that this is the same header used by CORS; otherwise, server administrators and web application developers would have had to manage two sets of cross-origin headers.) Also just like CORS, XDomainRequest will not send the user's cookies or authentication information along with its requests, so XDomainRequest calls are safer from confidentiality threats.

XDomainRequest does differ from CORS in that XDomainRequest is even more restrictive in its use of cookies. In CORS, you can opt to send cookies by setting the withCredentials property of the XMLHttpRequest object. In XDomainRequest, there's no such equivalent. XDomainRequest will never send cookies.

Final Thoughts on the Same-Origin Policy

Before we move on to discussing some browser-side attack techniques, we should probably clarify that there are lots of ways to make cross-origin calls other than the ones we've talked about here. Technically, all script code needs to make a cross-origin request is a way for it to send an HTTP GET message, and there are dozens of ways to do this. The catch is that just being able to send a cross-origin request usually isn't useful unless you're able to read the response. But the operative word here is "usually."

If an attacker is trying to steal your private information like your bank account number, and he can't get the bank to send it to him directly, maybe he can take an alternative tack and trick you into sending it to him. And in this case, you won't even need to wire a money order to someone claiming to be the Prime Minister of Nigeria—all you'll have to do is visit a vulnerable web site, and your browser will silently and automatically send the attacker the information he's looking for. In the next chapter, we'll look at the most popular browser-based attacks designed to bypass the same-origin policy: cross-site scripting and cross-site request forgery.

Your Plan

Sometimes it may seem as if the same-origin policy exists only to make programming more difficult, but the truth is that it's there to protect you and your users. If you have a great idea for a cross-origin application and you want to bend the rules a little, follow these steps to do it in the safest possible way.

- ❑ Use caution when embedding <script> elements in your application that point to third-party sites. If an attacker were able to gain access to any of these scripts, your application would be compromised and your users' personal data could be stolen.
- ❑ Don't use the JavaScript "eval" function to parse JSON strings into script objects. The parseJSON function is a much safer alternative, although it's only available in newer versions of web browsers.
- ❑ If you can't use parseJSON because your users may be on older browsers, consider using an open JavaScript library like jQuery to do your JSON parsing.
- ❑ Using third-party JSONP is essentially the same thing as using third-party <script> elements. If you wouldn't trust the site to give you JavaScript, you shouldn't trust it enough to give you JSONP.
- ❑ Don't use JSONP to send sensitive data. Since JSONP is valid JavaScript, it's not protected by the same-origin policy.
- ❑ Lowering a page's document.domain value increases the number of sites that can read data from that page, but this means that it also increases the number of sites that can steal data from that page. Also be aware that in most browsers, once you lower a page's document.domain value, you can't change it back again.
- ❑ When you're defining cross-origin policy files for Flash and Silverlight, it's safer to list specific sites that you want to grant access to than it is to use wildcards like *.com. It's really only appropriate to allow wildcard access when your site doesn't have any sensitive data.
- ❑ Consider splitting your site into separate subdomains for authenticated and anonymous use. This will let you create an open cross-origin policy file for the anonymous portion, and put tighter controls on the authenticated content.
- ❑ You can use the cross-origin resource sharing (CORS) feature of XMLHttpRequest without worry, as long as you don't allow credentialed requests.
- ❑ XDomainRequest is much safer to use than other alternatives, but remember that its functionality is limited so that it can never send cookies, and that it's only available in Internet Explorer versions 8 and later.

We've Covered

Defining the same-origin policy

- Which request components define an origin
- How different browsers define origin in different ways
- Why we need the same-origin policy: taking a look at a world without it

Exceptions to the same-origin policy

- The HTML <script> element
- JSON and JSONP
- iframes and JavaScript's document.domain property
- Adobe Flash cross-domain policy file
- Microsoft Silverlight client access policy file
- XMLHttpRequest (Ajax) and cross-origin resource sharing (CORS)
- Internet Explorer's XDomainRequest