Chapter 9. Content Isolation Logic

Most of the security assurances provided by web browsers are meant to isolate documents based on their origin. The premise is simple: Two pages from different sources should not be allowed to interfere with each other. Actual practice can be more complicated, however, as no universal agreement exists about where a single document begins and ends or what constitutes a single origin. The result is a sometimes unpredictable patchwork of contradictory policies that don't quite work well together but that can't be tweaked without profoundly affecting all current legitimate uses of the Web.

These problems aside, there is also little clarity about what actions should be subject to security checks in the first place. It seems clear that some interactions, such as following a link, should be permitted without special restrictions as they are essential to the health of the entire ecosystem, and that others, such as modifying the contents of a page loaded in a separate window, should require a security check. But a large gray area exists between these extremes, and that middle ground often feels as if it's governed more by a roll of the dice than by any unified plan. In these murky waters, vulnerabilities such as cross-site request forgery (see Chapter 4) abound.

It's time to start exploring. Let's roll a die of our own and kick off the journey with JavaScript.

Same-Origin Policy for the Document Object Model

The *same-origin policy* (*SOP*) is a concept introduced by Netscape in 1995 alongside JavaScript and the Document Object Model (DOM), just one year after the creation of HTTP cookies. The basic rule behind this policy is straightforward: Given any two separate JavaScript execution contexts, one should be able to access the DOM of the other only if the protocols, DNS names,[43] and port numbers associated with their host documents match exactly. All other cross-document JavaScript DOM access should fail.

The protocol-host-port tuple introduced by this algorithm is commonly referred to as *origin*. As a basis for a security policy, this is pretty robust: SOP is implemented across all modern browsers with a good degree of consistency and with only occasional bugs.[44] In fact, only Internet Explorer stands out, as it ignores the port number for the purpose of origin checks. This practice is somewhat less secure, particularly given the risk of having non-HTTP services running on a remote host for HTTP/0.9 web servers (see Chapter 3). But usually it makes no appreciable difference.

Table 9-1 illustrates the outcome of SOP checks in a variety of situations.

Table 9-1. Outcomes of SOP Checks

Originating document	Accessed document	Non-IE browser	Internet Explorer
http://example.com/a/	http://example.com/b/	Access okay	Access okay
http://example.com/	http://www.example.com/	Host mismatch	Host mismatch
http://example.com/	https://example.com/	Protocol mismatch	Protocol mismatch
http://example.com:81/	http://example.com/	Port mismatch	Access okay

NOTE

This same-origin policy was originally meant to govern access only to the DOM ; that is, the methods and properties related to the contents of the actual displayed document. The policy has been gradually extended to protect other obviously sensitive areas of the root JavaScript object, but it is not all-inclusive. For example, non-same-origin scripts can usually still call *location.assign()* or *location.replace(...)* on an arbitrary window or a frame. The extent and the consequences of these exemptions are the subject of Chapter 11.

The simplicity of SOP is both a blessing and a curse. The mechanism is fairly easy to understand and not too hard to implement correctly, but its inflexibility can be a burden to web developers. In some contexts, the policy is too broad, making it impossible to, say, isolate home pages belonging to separate users (short of giving each a separate domain). In other cases, the opposite is true: The policy makes it difficult for legitimately cooperating sites (say, *login.example.com* and *payments.example.com*) to seamlessly exchange data.

Attempts to fix the first problem—to narrow down the concept of an origin—are usually bound to fail because of interactions with other explicit and hidden security controls in the browser. Attempts to broaden origins or facilitate cross-domain interactions are more common. The two broadly supported ways of achieving these goals are *document.domain* and *postMessage(...)*, as discussed below.

document.domain

This JavaScript property permits any two cooperating websites that share a common top-level domain (such as *example.com*, or even just *.com*) to agree that for the purpose of future same-origin checks, they want to be considered equivalent. For example, both *login.example.com* and *payments.example.com* may perform the following assignment:

```
document.domain = "example.com"
```

Setting this property overrides the usual hostname matching logic during same-origin policy checks. The protocols and port numbers still have to match, though; if they don't, tweaking *document.domain* will not have the desired effect.

Both parties must explicitly opt in for this feature. Simply because *login.example.com* has set its *document.domain* to *example.com* does not mean that it will be allowed to access content originating from the website hosted at http://example.com/. That website needs to perform such an assignment, too, even if common sense would indicate that it is a no-op. This effect is symmetrical. Just as a page that sets *document.domain* will not be able to access pages that did not, the action of setting the property also renders the caller mostly (but not fully!)[45] out of reach of normal documents that previously would have been considered same-origin with it. Table 9-2 shows the effects of various values of *document.domain*.

Despite displaying a degree of complexity that hints at some special sort of cleverness, *document.domain* is not particularly safe. Its most significant weakness is that it invites unwelcome guests. After two parties mutually set this property to *example.com*, it is not simply the case that *login.example.com* and *payments.example.com* will be able to communicate; *funny-cat-videos.example.com* will be able to jump on the bandwagon as well. And because of the degree of access permitted between the pages, the integrity of any of the participating JavaScript contexts simply cannot be guaranteed to any realistic extent. In other words, touching *document.domain* inevitably entails tying the security of your page to the security of the weakest link in the entire domain. An extreme case of setting the value to **.com* is essentially equivalent to assisted suicide.

Table 9-2. Outcomes of document.domain Checks

Originating document		Accessed document		Outcome
URL	document.domain	URL	document.domain	
http://www.example.com/	*example.com*	http://payments.example.com/	*example.com*	Access okay
http://www.example.com/	*example.com*	https://payments.example.com/	*example.com*	Protocol mismatch
http://payments.example.com/	*example.com*	http://example.com/	**(not set)**	Access denied
http://www.example.com/	**(not set)**	http://www.example.com/	*example.com*	Access denied

postMessage(...)

The *postMessage(...)* API is an HTML5 extension that permits slightly less convenient but remarkably more secure communications between non-same-origin sites without automatically giving up the integrity of any of the parties involved. Today it is supported in all up-to-date browsers, although because it is fairly new, it is not found in Internet Explorer 6 or 7.

The mechanism permits a text message of any length to be sent to any window for which the sender holds a valid JavaScript handle (see Chapter 6). Although the same-origin policy has a number of gaps that permit similar functionality to be implemented by other means,[46] this one is actually safe to use. It allows the sender to specify what origins are permitted to receive the message in the first place (in case the URL of the target window has changed), and it provides the recipient with the identity of the sender so that the integrity of the channel can be ascertained easily. In contrast, legacy methods that rely on SOP loopholes usually don't come with such assurances; if a particular action is permitted without robust security checks, it can usually also be triggered by a rogue third party and not just by the intended participants.

To illustrate the proper use of *postMessage(...)*, consider a case in which a top-level document located at *payments.example.com* needs to obtain user login information for display purposes. To accomplish this, it loads a frame pointing to *login.example.com*. This frame can simply issue the following command:

```
parent.postMessage("user=bob", "https://payments.example.com");
```

The browser will deliver the message only if the embedding site indeed matches the specified, trusted origin. In order to securely process this response, the top-level document needs to use the following code:

```
// Register the intent to process incoming messages:
addEventListener("message", user_info, false);


// Handle actual data when it arrives:
function user_info(msg) {
  if (msg.origin == "https://login.example.com") {
    // Use msg.data as planned
  }
}
```

PostMessage(...) is a very robust mechanism that offers significant benefits over *document.domain* and over virtually all other guerrilla approaches that predate it; therefore, it should be used as often as possible. That said, it can still be misused. Consider the following check that looks for a substring in the domain name:

```
if (msg.origin.indexOf(".example.com") != -1) { ... }
```

As should be evident, this comparison will not only match sites within *example.com* but will also happily accept messages from www.example.com.bunnyoutlet.com. In all likelihood, you will stumble upon code like this more than once in your journeys. Such is life!

NOTE

Recent tweaks to HTML5 extended the *postMessage(...)* API to incorporate somewhat overengineered "ports" and "channels," which are meant to facilitate stream-oriented communications between websites. Browser support for these features is currently very limited and their practical utility is unclear, but from the security standpoint, they do not appear to be of any special concern.

Interactions with Browser Credentials

As we are wrapping up the overview of the DOM-based same-origin policy, it is important to note that it is in no way synchronized with ambient credentials, SSL state, network context, or many other potentially security-relevant parameters tracked by the browser. Any two windows or frames opened in a browser will remain same-origin with each other even if the user logs out from one account and logs into another, if the page switches from using a good HTTPS certificate to a bad one, and so on.

This lack of synchronization can contribute to the exploitability of other security bugs. For example, several sites do not protect their login forms against cross-site request forgery, permitting any third-party site to simply submit a username and a password and log the user into an attacker-controlled account. This may seem harmless at first, but when the content loaded in the browser before and after this operation is considered same-origin, the impact of normally ignored "self-inflicted" cross-site scripting vulnerabilities (i.e., ones

where the owner of a particular account can target only himself) is suddenly much greater than it would previously appear. In the most basic scenario, the attacker may first open and keep a frame pointing to a sensitive page on the targeted site (e.g., http://www.fuzzybunnies.com/address_book.php) and then log the victim into the attacker-controlled account to execute self-XSS in an unrelated component of fuzzybunnies.com. Despite the change of HTTP credentials, the code injected in that latter step will have unconstrained access to the previously loaded frame, permitting data theft.

Same-Origin Policy for XMLHttpRequest

The *XMLHttpRequest* API, mentioned in this book on several prior occasions, gives JavaScript programs the ability to issue almost unconstrained HTTP requests to the server from which the host document originated, and read back response headers and the document body. The ability to do so would not be particularly significant were it not for the fact that the mechanism leverages the existing browser HTTP stack and its amenities, including ambient credentials, caching mechanisms, keep-alive sessions, and so on.

A simple and fairly self-explanatory use of a synchronous *XMLHttpRequest* could be as follows:

```
var x = new XMLHttpRequest();

x.open("POST", "/some_script.cgi", false);

x.setRequestHeader("X-Random-Header", "Hi mom!");

x.send("...POST payload here...");

alert(x.responseText);
```

Asynchronous requests are very similar but are executed without blocking the JavaScript engine or the browser. The request is issued in the background, and an event handler is called upon completion instead.

As originally envisioned, the ability to issue HTTP requests via this API and to read back the data is governed by a near-verbatim copy of the same-origin policy with two minor and seemingly random tweaks. First, the *document.domain* setting has no effect on this mechanism, and the destination URL specified for *XMLHttpRequest.open(...)* must always match the true origin of the document. Second, in this context, port number is taken into account in Internet Explorer versions prior to 9, even though this browser ignores it elsewhere.

The fact that *XMLHttpRequest* gives the user an unprecedented level of control over the HTTP headers in a request can actually be advantageous to security. For example, inserting a custom HTTP header, such as *X-Coming-From: same-origin*, is a very simple way to verify that a particular request is not coming from a third-party domain, because no other site should be able to insert a custom header into a browser-issued request. This assurance is not very strong, because no specification says that the implicit restriction on cross-domain headers can't change;[47] nevertheless, when it comes to web security, such assumptions are often just something you have to learn to live with.

Control over the structure of an HTTP request can also be a burden, though, because inserting certain types of headers may change the meaning of a request to the destination server, or to the proxies, without the browser realizing it. For example, specifying an incorrect *Content-Length* value may allow an attacker to smuggle a second request into a keep-alive HTTP session maintained by the browser, as shown here.

```
var x = new XMLHttpRequest();

x.open("POST", "http://www.example.com/", false);


// This overrides the browser-computed Content-Length header:

x.setRequestHeader("Content-Length", "7");


// The server will assume that this payload ends after the first

// seven characters, and that the remaining part is a separate

// HTTP request.

x.send(

  "Gotcha!\n" +

  "GET /evil_response.html HTTP/1.1\n" +

  "Host: www.bunnyoutlet.com\n\n"

);
```

If this happens, the response to that second, injected request may be misinterpreted by the browser later, possibly poisoning the cache or injecting content into another website. This problem is especially pronounced if an HTTP proxy is in use and all HTTP requests are sent through a shared channel.

Because of this risk, and following a lot of trial and error, modern browsers blacklist a selection of HTTP headers and request methods. This is done with relatively little consistency: While *Referer*, *Content-Length*, and *Host* are universally banned, the handling of headers such as *User-Agent*, *Cookie*, *Origin*, or *If-Modified-Since* varies from one browser to another. Similarly, the TRACE method is blocked everywhere, because of the unanticipated risk it posed to *httponly* cookies—but the CONNECT method is permitted in Firefox, despite carrying a vague risk of messing with HTTP proxies.

Naturally, implementing these blacklists has proven to be an entertaining exercise on its own. Strictly for your amusement, consider the following cases that worked in some browsers as little as three years ago:[176]

```
XMLHttpRequest.setRequestHeader("X-Harmless", "1\nOwned: Gotcha");
```

or

```
XMLHttpRequest.setRequestHeader("Content-Length: 123 ", "");
```

or simply

```
XMLHttpRequest.open("GET\thttp://evil.com\tHTTP/1.0\n\n", "/", false);
```

NOTE

Cross-Origin Resource Sharing [177] *(CORS)* is a proposed extension to *XMLHttpRequest* that permits HTTP requests to be issued across domains and then read back if a particular response header appears in the returned data. The mechanism changes the semantics of the API discussed in this session by allowing certain "vanilla" cross-domain requests, meant to be no different from regular navigation, to be issued via *XMLHttpRequest.open(...)* with no additional checks; more elaborate requests require an OPTIONS-based preflight request first. CORS is already available in some browsers, but it is opposed by Microsoft engineers, who pursued a competing *XDomainRequest* approach in Internet Explorer 8 and 9. Because the outcome of this conflict is unclear, a detailed discussion of CORS is reserved for Chapter 16, which provides a more systematic overview of upcoming and experimental mechanisms.

Same-Origin Policy for Web Storage

Web storage is a simple database solution first implemented by Mozilla engineers in Firefox 1.5 and eventually embraced by the HTML5 specification.[178] It is available in all current browsers but not in Internet Explorer 6 or 7.

Following several dubious iterations, the current design relies on two simple JavaScript objects: *localStorage* and *sessionStorage*. Both objects offer an identical, simple API for creating, retrieving, and deleting name-value pairs in a browser-managed database. For example:

```
localStorage.setItem("message", "Hi mom!");

alert(localStorage.getItem("message"));

localstorage.removeItem("message");
```

The *localStorage* object implements a persistent, origin-specific storage that survives browser shutdowns, while *sessionStorage* is expected to be bound to the current browser window and provide a temporary caching mechanism that is destroyed at the end of a browsing session. While the specification says that both *localStorage* and *sessionStorage* should be associated with an SOP-like origin (the protocol-host-port tuple), implementations in some browsers do not follow this advice, introducing potential security bugs. Most notably, in Internet Explorer 8, the protocol is not taken into account when computing the origin, putting HTTP and HTTPS pages within a shared context. This design makes it very unsafe for HTTPS sites to store or read back sensitive data through this API. (This problem is corrected in Internet Explorer 9, but there appears to be no plan to backport the fix.)

In Firefox, on the other hand, the *localStorage* behaves correctly, but the *sessionStorage* interface does not. HTTP and HTTPS use a shared storage context, and although a check is implemented to prevent HTTP content from reading keys created by HTTPS scripts, there is a serious loophole: Any key first created over HTTP, and then updated over HTTPS, will remain visible to nonencrypted pages. This bug, originally reported in 2009,[179] will eventually be resolved, but when is not clear.

Security Policy for Cookies

We discussed the semantics of HTTP cookies in Chapter 3, but that discussion left out one important detail: the security rules that must be implemented to protect cookies belonging to one site from being tampered with by unrelated pages. This topic is particularly interesting because the approach taken here predates the same-origin policy and interacts with it in a number of unexpected ways.

Cookies are meant to be scoped to domains, and they can't be limited easily to just a single hostname value. The *domain* parameter provided with a cookie may simply match the current hostname (such as *foo.example.com*), but this will not prevent the cookie from being sent to any eventual subdomains, such as *bar.foo.example.com*. A qualified right-hand fragment of the hostname, such as *example.com*, can be specified to request a broader scope, however.

Amusingly, the original RFCs imply that Netscape engineers wanted to allow exact host-scoped cookies, but they did not follow their own advice. The syntax devised for this purpose was not recognized by the descendants of Netscape Navigator (or by any other implementation for that matter). To a limited extent, setting host-scoped cookies is possible in some browsers by completely omitting the *domain* parameter, but this method will have no effect in Internet Explorer.

Table 9-3 illustrates cookie-setting behavior in some distinctive cases.

Table 9-3. A Sample of Cookie-Setting Behaviors

Cookie set at *foo.example.com*, *domain* parameter is:	Scope of the resulting cookie	
	Non-IE browsers	Internet Explorer
(value omitted)	*foo.example.com* (exact)	*.foo.example.com
bar.foo.example.com	Cookie not set: domain more specific than origin	
foo.example.com	*.foo.example.com	
baz.example.com	Cookie not set: domain mismatch	
example.com	*.example.com	
ample.com	Cookie not set: domain mismatch	
.com	Cookie not set: domain too broad, security risk	

The only other true cookie-scoping parameter is the path prefix: Any cookie can be set with a specified *path* value. This instructs the browser to send the cookie back only on requests to matching directories; a cookie scoped to *domain* of *example.com* and *path* of */some/path/* will be included on a request to

```
http://foo.example.com/some/path/subdirectory/hello_world.txt
```

This mechanism can be deceptive. URL paths are not taken into account during same-origin policy checks and, therefore, do not form a useful security boundary. Regardless of how cookies work, JavaScript code can simply hop between any URLs on a single host at will and inject malicious payloads into such targets, abusing any functionality protected with path-bound cookies. (Several security books and white papers recommend path scoping as a security measure to this day. In most cases, this advice is dead wrong.)

Other than the true scoping features (which, along with cookie name, constitute a tuple that uniquely identifies every cookie), web servers can also output cookies with two special, independently operated flags: *httponly* and *secure*. The first, *httponly*, prevents access to the cookie via the *document.cookie* API in the hope of making it more difficult to simply copy a user's credentials after successfully injecting a malicious script on a page. The second, *secure*, stops the cookie from being submitted on requests over unencrypted protocols, which makes it possible to build HTTPS services that are resistant to active attacks.[48]

The pitfall of these mechanisms is that they protect data only against reading and not against overwriting. For example, it is still possible for JavaScript code delivered over HTTP to simply overflow the per-domain cookie jar and then set a new cookie without the *secure* flag.[49] Because the *Cookie* header sent by the browser provides no metadata about the origin of a particular cookie or its scope, such a trick is very difficult to detect. A prominent consequence of this behavior is that the common "stateless" way of preventing cross-site request forgery vulnerabilities by simultaneously storing a secret token in a client-side cookie and in a hidden form field, and then comparing the two, is not particularly safe for HTTPS websites. See if you can figure out why!

NOTE

Speaking of destructive interference, until 2010, *httponly* cookies also clashed with *XMLHttpRequest*. The authors of that API simply have not given any special thought to whether the *XMLHttpRequest.getResponseHeader(...)* function should be able to inspect server-supplied *Set-Cookie* values flagged as *httponly*—with predictable results.

Impact of Cookies on the Same-Origin Policy

The same-origin policy has some undesirable impact on the security of cookies (specifically, on the path-scoping mechanism), but the opposite interaction is more common and more problematic. The difficulty is that HTTP cookies often function as credentials, and in such cases, the ability to obtain them is roughly equivalent to finding a way to bypass SOP. Quite simply, with the right set of cookies, an attacker could use her own browser to interact with the target site on behalf of the victim; same-origin policy is taken out of the picture, and all bets are off.

Because of this property, any discrepancies between the two security mechanisms can lead to trouble for the more restrictive one. For example, the relatively promiscuous domain-scoping rules used by HTTP cookies mean that it is not possible to isolate fully the sensitive content hosted on *webmail.example.com* from the less trusted HTML present on *blog.example.com*. Even if the owners of the webmail application scope their cookies tightly (usually at the expense of complicating the sign-on process), any attacker who finds a script injection vulnerability on the blogging site can simply overflow the per-domain cookie jar, drop the current credentials, and set his own **.example.com* cookies. These injected cookies will be sent to *webmail.example.com* on all subsequent requests and will be largely indistinguishable from the real ones.

This trick may seem harmless until you realize that such an action may effectively log the victim into a bogus account and that, as a result, certain actions (such as sending email) may be unintentionally recorded within that account and leaked to the attacker before any foul play is noticed. If webmail sounds too exotic, consider doing the same on Amazon or Netflix: Your casual product searches may be revealed to the attacker before you notice anything unusual about the site. (On top of this, many websites are simply not prepared to handle malicious payloads in injected cookies, and unexpected inputs may lead to XSS or similar bugs.)

The antics of HTTP cookies also make it very difficult to secure encrypted traffic against network-level attackers. A *secure* cookie set by https://webmail.example.com/ can still be clobbered and replaced by a made-up value set by a spoofed page at http://webmail.example.com/, even if there is no actual web service listening on port 80 on the target host.

Problems with Domain Restrictions

The misguided notion of allowing domain-level cookies also poses problems for browser vendors and is a continuing source of misery. The key question is how to reliably prevent *example.com* from setting a cookie for **.com* and avoid having this cookie unexpectedly sent to every other destination on the Internet.

Several simple solutions come to mind, but they fall apart when you have to account for country-level TLDs: *example.com.pl* must be prevented from setting a **.com.pl* cookie, too. Realizing this, the original Netscape cookie specification provided the following advice:

> *Only hosts within the specified domain can set a cookie for a domain and domains must have at least two (2) or three (3) periods in them to prevent domains of the form: ".com", ".edu", and ".va.us".*
>
> *Any domain that fails within one of the seven special top level domains listed below only requires two periods. Any other domain requires at least three. The seven special top level domains are: "COM", "EDU", "NET", "ORG", "GOV", "MIL", and "INT".*

Alas, the three-period rule makes sense only for country-level registrars that mirror the top-level hierarchy (*example.co.uk*) but not for the just as populous group of countries that accept direct registrations (*example.fr*). In fact, there are places where both approaches are allowed; for example, both *example.jp* and *example.co.jp* are perfectly fine.

Because of the out-of-touch nature of this advice, most browsers disregarded it and instead implemented a patchwork of conditional expressions that only led to more trouble. (In one case, for over a decade, you could actually set cookies for **.com.pl*.) Comprehensive fixes to country-code top-level domain handling have shipped in all modern browsers in the past four years, but as of this writing they have not been backported to Internet Explorer 6 and 7, and they probably never will be.

NOTE

To add insult to injury, the Internet Assigned Numbers Authority added a fair number of top-level domains in recent years (for example, *.int* and *.biz*), and it is contemplating a proposal to allow arbitrary generic top-level domain registrations. If it comes to this, cookies will probably have to be redesigned from scratch.

The Unusual Danger of "localhost"

One immediately evident consequence of the existence of domain-level scoping of cookies is that it is fairly unsafe to delegate any hostnames within a sensitive domain to any untrusted (or simply vulnerable) party; doing so may affect the confidentiality, and invariably the integrity, of any cookie-stored credentials—and, consequently, of any other information handled by the targeted application.

So much is obvious, but in 2008, Tavis Ormandy spotted something far less intuitive and far more hilarious:[180] that because of the port-agnostic behavior of HTTP cookies, an additional danger lies in the fairly popular and convenient administrative practice of adding a "localhost" entry to a domain and having it point to 127.0.0.1.[50] When Ormandy first published his advisory, he asserted that this practice is widespread—not a controversial claim to make—and included the following resolver tool output to illustrate his point:

```
localhost.microsoft.com has address 127.0.0.1

localhost.ebay.com has address 127.0.0.1

localhost.yahoo.com has address 127.0.0.1
```

```
localhost.fbi.gov has address 127.0.0.1

localhost.citibank.com has address 127.0.0.1

localhost.cisco.com has address 127.0.0.1
```

Why would this be a security risk? Quite simply, it puts the HTTP services on the user's own machine within the same domain as the remainder of the site, and more importantly, it puts all the services that only *look* like HTTP in the very same bucket. These services are typically not exposed to the Internet, so there is no perceived need to design them carefully or keep them up-to-date. Tavis's case in point is a printer-management service provided by CUPS (Common UNIX Printing System), which would execute attacker-supplied JavaScript in the context of *example.com* if invoked in the following way:

```
http://localhost.example.com:631/jobs/?[...]

  &job_printer_uri=javascript:alert("Hi mom!")
```

The vulnerability in CUPS can be fixed, but there are likely many other dodgy local services on all operating systems—everything from disk management tools to antivirus status dashboards. Introducing entries pointing back to 127.0.0.1, or any other destinations you have no control over, ties the security of cookies within your domain to the security of random third-party software. That is a good thing to avoid.

Cookies and "Legitimate" DNS Hijacking

The perils of the domain-scoping policy for cookies don't end with *localhost*. Another unintended interaction is related to the common, widely criticized practice of some ISPs and other DNS service providers of hijacking domain lookups for nonexistent (typically mistyped) hosts. In this scheme, instead of returning the standard-mandated NXDOMAIN response from an upstream name server (which would subsequently trigger an error message in the browser or other networked application), the provider will falsify a record to imply that this name resolves to its site. Its site, in turn, will examine the *Host* header supplied by the browser and provide the user with unsolicited, paid contextual advertising that appears to be vaguely related to her browsing interests. The usual justification offered for this practice is that of offering a more user-friendly browsing experience; the real incentive, of course, is to make more money.

Internet service providers that have relied on this practice include Cablevision, Charter, Earthlink, Time Warner, Verizon, and many more. Unfortunately, their approach is not only morally questionable, but it also creates a substantial security risk. If the advertising site contains any script-injection vulnerabilities, the attacker can exploit them in the context of any other domain simply by accessing the vulnerable functionality through an address such as *nonexistent.example.com*. When coupled with the design of HTTP cookies, this practice undermines the security of any arbitrarily targeted services on the Internet.

Predictably, script-injection vulnerabilities can be found in such hastily designed advertising traps without much effort. For example, in 2008, Dan Kaminsky spotted and publicized a cross-site scripting vulnerability on the pages operated by Earthlink.[181]

All right, all right: It's time to stop obsessing over cookies and move on.

Plug-in Security Rules

Browsers do not provide plug-in developers with a uniform and extensible API for enforcing security policies; instead, each plug-in decides what rules should be applied to executed content and how to put them into action. Consequently, even though plug-in security models are to some extent inspired by the same-origin policy, they diverge from it in a number of ways.

This disconnect can be dangerous. In Chapter 6, we discussed the tendency for plug-ins to rely on inspecting the JavaScript *location* object to determine the origin of their hosting page. This misguided practice forced browser developers to restrict the ability of JavaScript programs to tamper with some portions of their runtime environment to save the day. Another related, common source of incompatibilities is the interpretation of URLs. For example, in the middle of 2010, one researcher discovered that Adobe Flash had trouble with the following URL:[182]

```
http://example.com:80@bunnyoutlet.com/
```

The plug-in decided that the origin of any code retrieved through this URL should be set to *example.com*, but the browser, when presented with such a URL, would naturally retrieve the data from bunnyoutlet.com instead and then hand it over to the confused plug-in for execution.

While this particular bug is now fixed, other vulnerabilities of this type can probably be expected in the future. Replicating some of the URL-parsing quirks discussed in Chapter 2 and Chapter 3 can be a fool's errand and, ideally, should not be attempted at all.

It would not be polite to end this chapter on such a gloomy note! Systemic problems aside, let's see how some of the most popular plug-ins approach the job of security policy enforcement.

Adobe Flash

The Flash security model underwent a major overhaul in 2008,[183] and since then, it has been reasonably robust. Every loaded Flash applet is now assigned an SOP-like origin derived from its originating URL[51] and is granted nominal origin-related permissions roughly comparable to those of JavaScript. In particular, each applet can load cookie-authenticated content from its originating site, load some constrained datatypes from other origins, and make same-origin *XMLHttpRequest*-like HTTP calls through the *URLRequest* API. The set of permissible methods and request headers for this last API is managed fairly reasonably and, as of this writing, is more restrictive than most of the browser-level blacklists for *XMLHttpRequest* itself.[184]

On top of this sensible baseline, three flexible but easily misused mechanisms permit this behavior to be modified to some extent, as discussed next.

Markup-Level Security Controls

The embedding page can specify three special parameters provided through *<embed>* or *<object>* tags to control how an applet will interact with its host page and the browser itself:

- ***AllowScriptAccess* parameter** This setting controls an applet's ability to use the JavaScript *ExternalInterface.call(...)* bridge (see Chapter 8) to execute JavaScript statements in the context of the embedding site. Possible values are *always*, *never*, and *sameorigin*; the last setting gives access to the page only if the page is same-origin with the applet itself. (Prior to the 2008 security overhaul, the plug-in defaulted to *always*; the current default is the much safer *sameorigin*.)
- ***AllowNetworking* parameter** This poorly named setting restricts an applet's permission to open or navigate browser windows and to make HTTP requests to its originating server. When set to *all* (the default), the applet can interfere with the browser; when set to *internal*, it can perform only nondisruptive, internal communications through the Flash plug-in. Setting this parameter to *none* disables most network-related APIs altogether.[52] (Prior to recent security improvements, *allowNetworking=all* opened up

several ways to bypass *allowScriptAccess=none*, for example, by calling *getURL(...)* on a *javascript:* URL. As of this writing, however, all scripting URLs should be blacklisted in this scenario.)

- ***AllowFullScreen* parameter** This parameter controls whether an applet should be permitted to go into full-screen rendering mode. The possible values are *true* and *false*, with *false* being the default. As noted in Chapter 8, the decision to give this capability to Flash applets is problematic due to UI spoofing risks; it should be not enabled unless genuinely necessary.

Security.allowDomain(...)

The *Security.allowDomain(...)* method[185] allows Flash applets to grant access to their variables and functions to any JavaScript code or to other applets coming from a different origin. Buyer beware: Once such access is granted, there is no reliable way to maintain the integrity of the original Flash execution context. The decision to grant such permissions should not be taken lightly, and the practice of calling *allowDomain("*")* should usually be punished severely.

Note that a weirdly named *allowInsecureDomain(...)* method is also available. The existence of this method does not indicate that *allowDomain(...)* is particularly secure; rather, the "insecure" variant is provided for compatibility with ancient, pre-2003 semantics that completely ignored the HTTP/HTTPS divide.

Cross-Domain Policy Files

Through the use of *loadPolicyFile(...)*, any Flash applet can instruct its runtime environment to retrieve a security policy file from an almost arbitrary URL. This XML-based document, usually named *crossdomain.xml*, will be interpreted as an expression of consent to cross-domain, server-level access to the origin determined by examining the policy URL.[186] The syntax of a policy file is fairly self-explanatory and may look like this:

```
<cross-domain-policy>
  <allow-access-from domain="foo.example.com"/>
  <allow-http-request-headers-from domain="*.example.com"
     headers="X-Some-Header" />
</cross-domain-policy>
```

The policy may permit actions such as loading cross-origin resources or issuing arbitrary *URLRequest* calls with whitelisted headers, through the browser HTTP stack. Flash developers do attempt to enforce a degree of path separation: A policy loaded from a particular subdirectory can in principle permit access only to files within that path. In practice, however, the interactions with SOP and with various path-mapping semantics of modern browsers and web application frameworks make it unwise to depend on this boundary.

> **NOTE**
>
> Making raw TCP connections via *XMLSocket* is also possible and controlled by an XML policy, but following Flash's 2008 overhaul, *XMLSocket* requires that a separate policy file be delivered on TCP port 843 of the destination server. This is fairly safe, because no other common services run on this port and, on many operating systems, only privileged users can launch services on any port below 1024. Because of the interactions with certain firewall-level mechanisms, such as FTP protocol helpers, this design may still cause some network-level interference,[187] but this topic is firmly beyond the scope of this book

As expected, poorly configured *crossdomain.xml* policies are an appreciable security risk. In particular, it is a very bad idea to specify *allow-access-from* rules that point to any domain you do not have full confidence in. Further, specifying "*" as a value for this parameter is roughly equivalent to executing *document.domain = "com"*. That is, it's a death wish.

Policy File Spoofing Risks

Other than the possibility of configuration mistakes, another security risk with Adobe's policy-based security model is that random user-controlled documents may be interpreted as cross-domain policies, contrary to the site owner's intent.

Prior to 2008, Flash used a notoriously lax policy parser, which when processing *loadPolicyFile(...)* files would skip arbitrary leading garbage in search of the opening *<cross-domain-policy>* tag. It would simply ignore the MIME type returned by the server when downloading the resource, too. As a result, merely hosting a valid, user-supplied JPEG image could become a grave security risk. The plug-in also skipped over any HTTP redirects, making it dangerous to do something as simple as issuing an HTTP redirect to a location you did not control (an otherwise harmless act).

Following the much-needed revamp of the *loadPolicyFile* behavior, many of the gross mistakes have been corrected, but the defaults are still not perfect. On the one hand, redirects now work intuitively, and the file must be a well-formed XML document. On the other, permissible MIME types include *text/**, *application/xml*, and *application/xhtml+xml*, which feels a bit too broad. *text/plain* or *text/csv* may be misinterpreted as a policy file, and that should not be the case.

Thankfully, to mitigate the problem, Adobe engineers decided to roll out *meta-policies*, policies that are hosted at a predefined, top-level location (*/crossdomain.xml*) that the attacker can't override. A meta-policy can specify sitewide restrictions for all the remaining policies loaded from attacker-supplied URLs. The most important of these restrictions is *<site-control permitted-cross-domain-policies="...">*. This parameter, when set to *master-only*, simply instructs the plug-in to disregard subpolicies altogether. Another, less radical value, *by-content-type*, permits additional policies to be loaded but requires them to have a nonambiguous *Content-Type* header set to *text/x-cross-domain-policy*.

Needless to say, it's highly advisable to use a meta-policy that specifies one of these two directives.

Microsoft Silverlight

If the transition from Flash to Silverlight seems abrupt, it's because the two are easy to confuse. The Silverlight plug-in borrows from Flash with remarkable zeal; in fact, it is safe to say that most of the differences between their security models are due solely to nomenclature. Microsoft's platform uses the same-origin-determination approach, substitutes *allowScriptAccess* with *enableHtmlAccess*, replaces *crossdomain.xml* with the slightly different *clientaccesspolicy.xml* syntax, provides a *System.Net.Sockets* API instead of *XMLSocket*, uses *HttpWebRequest* in place of *URLRequest*, rearranges the flowers, and changes the curtains in the living room.

The similarities are striking, down to the list of blocked request headers for the *HttpWebRequest* API, which even includes *X-Flash-Version* from the Adobe spec.[188] Such consistency is not a problem, though: In fact, it is preferable to having a brand-new security model to take into account. Plus, to its credit, Microsoft did make a couple of welcome improvements, including ditching the insecure *allowDomain* logic in favor of *RegisterScriptableObject*, an approach that allows only explicitly specified callbacks to be exposed to third-party domains.

Java

Sun's Java (now officially belonging to Oracle) is a very curious case. Java is a plug-in that has fallen into disuse, and its security architecture has not received much scrutiny in the past decade or so. Yet, because of its large installed base, it is difficult to simply ignore it and move on.

Unfortunately, the closer you look, the more evident it is that the ideas embraced by Java tend to be incompatible with the modern Web. For example, a class called *java.net.HttpURLConnection* [189] permits credential-bearing HTTP requests to be made to an applet's originating website, but the "originating website" is understood as *any* website hosted at a particular IP address, as sanctioned by the *java.net.URL.equals(...)* check. This model essentially undoes any isolation between HTTP/1.1 virtual hosts—an isolation strongly enforced by the same-origin policy, HTTP cookies, and virtually all other browser security mechanisms in use today.

Further along these lines, the *java.net.URLConnection* class[190] allows arbitrary request headers, including *Host*, to be set by the applet, and another class, *Socket*,[191] permits unconstrained TCP connections to arbitrary ports on the originating server. All of these behaviors are frowned upon in the browser and in any other contemporary plug-in.

Origin-agnostic access from the applet to the embedding page is provided through the *JSObject* mechanism and is expected to be controlled by the embedding party through the *mayscript* attribute specified in the *<applet>*, *<embed>*, or *<object>* tags.[192] The documentation suggests that this is a security feature:

> *Due to security reasons, JSObject support is not enabled in Java Plug-in by default. To enable JSObject support in Java Plug-in, a new attribute called MAYSCRIPT needs to be present in the EMBED/OBJECT tag.*

Unfortunately, the documentation neglects to mention that another closely related mechanism, *DOMService*,[193] ignores this setting and gives applets largely unconstrained access to the embedding page. While *DOMService* is not supported in Firefox and Opera, it is available in other browsers, which makes any attempt to load third-party Java content equivalent to granting full access to the embedding site.

Whoops.

NOTE

Interesting fact: Recent versions of Java attempt to copy the *crossdomain.xml* support available in Flash.

Coping with Ambiguous or Unexpected Origins

This concludes our overview of the basic security policies and consent isolation mechanisms. If there is one observation to be made, it's that most of these mechanisms depend on the availability of a well-formed, canonical hostname from which to derive the context for all the subsequent operations. But what if this information is not available or is not presented in the expected form?

Well, that's when things get funny. Let's have a look at some of the common corner cases, even if just for fleeting amusement.

IP Addresses

Due to the failure to account for IP addresses when designing HTTP cookies and the same-origin policy, almost all browsers have historically permitted documents loaded from, say, *http://1.2.3.4/* to set cookies for a "domain" named **.3.4*. Adjusting *document.domain* in a similar manner would work as well. In fact, some of these behaviors are still present in older versions of Internet Explorer.

This behavior is unlikely to have an impact on mainstream web applications, because such applications are not meant to be accessed through an IP-based URL and will often simply fail to function properly. But a handful of systems, used primarily by technical staff, are meant to be accessed by their IP addresses; these systems may simply not have DNS records configured at all. In these cases, the ability for *http://1.2.3.4/* to inject cookies for *http://123.234.3.4/* may be an issue. The IP-reachable administrative interfaces of home routers are of some interest, too.

Hostnames with Extra Periods

At their core, cookie-setting algorithms still depend on counting the number of periods in a URL to determine whether a particular *domain* parameter is acceptable. In order to make the call, the count is typically correlated with a list of several hundred entries on the vendor-maintained Public Suffix List (http://publicsuffix.org/).

Unfortunately for this algorithm, it is often possible to put extra periods in a hostname and still have it resolve correctly. Noncanonical hostname representations with excess periods are usually honored by OS-level resolvers and, if honored, will confuse the browser. Although said browser would not automatically consider a domain such as www.example.com.pl. (with an extra trailing period) to be the same as the real www.example.com.pl, the subtle and seemingly harmless difference in the URL could escape even the most attentive users.

In such a case, interacting with the URL with trailing period can be unsafe, as other documents sharing the **.com.pl.* domain may be able to inject cross-domain cookies with relative ease.

This period-counting problem was first noticed around 1998.[194] About a decade later, many browser vendors decided to roll out basic mitigations by adding a yet another special case to the relevant code; as of this writing, Opera is still susceptible to this trick.

Non-Fully Qualified Hostnames

Many users browse the Web with their DNS resolvers configured to append local suffixes to all found hostnames, often without knowing. Such settings are usually sanctioned by ISPs or employers through automatic network configuration data (Dynamic Host Configuration Protocol, DHCP).

For any user browsing with such a setting, the resolution of DNS labels is ambiguous. For example, if the DNS search path includes *coredump.cx*, then www.example.com may resolve to the real www.example.com website or to www.example.com.coredump.cx if such a record exists. The outcomes are partly controlled by configuration settings and, to some extent, can be influenced by an attacker.

To the browser, both locations appear to be the same, which may have some interesting side effects. Consider one particularly perverse case: Should *http://com*, which actually resolves to http://com.coredump.cx/, be able to set **.com* cookies by simply omitting the *domain* parameter?

Local Files

Because local resources loaded through the *file:* protocol do not have an explicit hostname associated with them, it's impossible for the browser to compute a normal origin. For a very long time, the vendors simply decided that the best course of action in such a case would be to simply ditch the same-origin policy. Thus, any HTML document saved to disk would automatically be granted access to any other local files via *XMLHttpRequest* or DOM and, even more inexplicably, would be able to access any Internet-originating content in the same way.

This proved to be a horrible design decision. No one expected that the mere act of downloading an HTML document would put all of the user's local files, and his online credentials, in jeopardy. After all, accessing that same document over the Web would be perfectly safe.

Many browsers have tried to close this loophole in recent years, with varying degrees of success:

Chrome (and, by extension, other WebKit browsers)

The Chrome browser completely disallows any cross-document DOM or *XMLHttpRequest* access from *file:* origins, and it ignores *document.cookie* calls or *<meta http-equiv="Set-Cookie" ...>* directives in this setting. Access to a *localStorage* container shared by all *file:* documents is permitted, but this may change soon.

Firefox

Mozilla's browser permits access only to files within the directory of the original document, as well as nearby subdirectories. This policy is pretty good, but it still poses some risk to documents stored or previously downloaded to that location. Access to cookies via *document.cookie* or *<meta http-equiv="Set-Cookie" ...>* is possible, and all *file:* cookies are visible to any other local JavaScript code.[53] The same holds true for access to storage mechanisms.

Internet Explorer 7 and above

Unconstrained access to local and Internet content from *file:* origins is permitted, but it requires the user to click through a nonspecific warning to execute JavaScript first. The consequences of this action are not explained clearly (the help subsystem cryptically states that *"Internet Explorer restricts this content because occasionally these programs can malfunction or give you content you don't want"*), and many users may well be tricked into clicking through the prompt.

Internet Explorer's cookie semantics are similar to those of Firefox. Web storage is not supported in this origin, however.

Opera and **Internet Explorer 6**

Both of these browsers permit unconstrained DOM or *XMLHttpRequest* access without further checks. Noncompartmentalized *file:* cookies are permitted, too.

> **NOTE**
>
> Plug-ins live by their own rules in *file:* land: Flash uses a *local-with-filesystem* sandbox model,[195] which gives largely unconstrained access to the local filesystem, regardless of the policy enforced by the browser itself, while executing Java or Windows Presentation Framework applets from the local filesystem may in some cases be roughly equivalent to running an untrusted binary.

Pseudo-URLs

The behavior of pseudo-URLs such as *about:*, *data:*, or *javascript:* originally constituted a significant loophole in the implementations of the same-origin policy. All such URLs would be considered same-origin and would permit unconstrained cross-domain access from any other resource loaded over the same scheme. The current behavior, which is very different, will be the topic of the next chapter of this book; in a nutshell, the status quo reflects several rounds of hastily implemented improvements and is a complex mix of browser-specific special cases and origin-inheritance rules.

Browser Extensions and UI

Several browsers permit JavaScript-based UI elements or certain user-installed browser extensions to run with elevated privileges. These privileges may entail circumventing specific SOP checks or calling normally unavailable APIs in order to write files, modify configuration settings, and so on.

Privileged JavaScript is a prominent feature of Firefox, where it is used with XUL to build large portions of the browser user interface. Chrome also relies on privileged JavaScript to a smaller but still notable degree.

The same-origin policy does not support privileged contexts in any specific way. The actual mechanism by which extra privileges are granted may involve loading the document over a special and normally unreachable URL scheme, such as *chrome:* or *res:*, and then adding special cases for that scheme in other portions of the browser code. Another option is simply to toggle a binary flag for a JavaScript context, regardless of its actual origin, and examine that flag later. In all cases, the behavior of standard APIs such as *localStorage*, *document.domain*, or *document.cookie* may be difficult to predict and should not be relied upon: Some browsers attempt to maintain isolation between the contexts belonging to different extensions, but most don't.

> **NOTE**
>
> Whenever writing browser extensions, any interaction with nonprivileged contexts must be performed with extreme caution. Examining untrusted contexts can be difficult, and the use of mechanisms such as *eval(...)* or *innerHMTL* may open up privilege-escalation paths.

Other Uses of Origins

Well, that's all to be said about browser-level content isolation logic for now. It is perhaps worth noting that the concept of origins and host- or domain-based security mechanisms is not limited to that particular task and makes many other appearances in the browser world. Other quasi-origin-based privacy or security features include preferences and cached information related to per-site cookie handling, pop-up blocking, geolocation sharing, password management, camera and microphone access (in Flash), and much, much more. These features tend to interact with the security features described in this chapter at least to some extent; we explore this topic in more detail soon.

SECURITY ENGINEERING CHEAT SHEET

Good Security Policy Hygiene for All Websites

To protect your users, include a top-level *crossdomain.xml* file with the *permitted-cross-domain-policies* parameter set to *master-only* or *by-content-type*, even if you do not use Flash anywhere on your site. Doing so will prevent unrelated attacker-controlled content from being misinterpreted as a secondary *crossdomain.xml* file, effectively undermining the assurances of the same-origin policy in Flash-enabled browsers.

When Relying on HTTP Cookies for Authentication

- Use the *httponly* flag; design the application so that there is no need for JavaScript to access authentication cookies directly. Sensitive cookies should be scoped as tightly as possible, preferably by not specifying *domain* at all.
- If the application is meant to be HTTPS only, cookies must be marked as *secure*, and you must be prepared to handle cookie injection gracefully. (HTTP contexts may overwrite *secure* cookies, even though they can't read them.) Cryptographic cookie signing may help protect against unconstrained modification, but it does not defend against replacing a victim's cookies with another set of legitimately obtained credentials.

When Arranging Cross-Domain Communications in JavaScript

- Do not use *document.domain*. Rely on *postMessage(...)* where possible and be sure to specify the destination origin correctly; then verify the sender's origin when receiving the data on the other end. Beware of naïve substring matches for domain names: *msg.origin.indexOf(".example.com")* is very insecure.
- Note that various pre-*postMessage* SOP bypass tricks, such as relying on *window.name*, are not tamper-proof and should not be used for exchanging sensitive data.

When Embedding Plug-in-Handled Active Content from Third Parties

Consult the cheat sheet in Chapter 8 first for general advice.

- **Flash:** Do not specify *allowScriptAccess=always* unless you fully trust the owner of the originating domain and the security of its site. Do not use this setting when embedding HTTP applets on HTTPS pages. Also, consider restricting *allowFullScreen* and *allowNetworking* as appropriate.
- **Silverlight:** Do not specify *enableHtmlAccess=true* unless you trust the originating domain, as above.
- **Java:** Java applets can't be safely embedded from untrusted sources. Omitting *mayscript* does not fully prevent access to the embedding page, so do not attempt to do so.

When Hosting Your Own Plug-in-Executed Content

- Note that many cross-domain communication mechanisms provided by browser plug-ins may have unintended consequences. In particular, avoid *crossdomain.xml*, *clientaccesspolicy.xml*, or *allowDomain(...)* rules that point to domains you do not fully trust.

When Writing Browser Extensions

- Avoid relying on *innerHTML*, *document.write(...)*, *eval(...)*, and other error-prone coding patterns, which can cause code injection on third-party pages or in a privileged JavaScript context.
- Do not make security-critical decisions by inspecting untrusted JavaScript security contexts, as their behavior can be deceptive.

[43] This and most other browser security mechanisms are based on DNS labels, not on examining the underlying IP addresses. This has a curious consequence: If the IP of a particular host changes, the attacker may be able to talk to the new destination through the user's browser, possibly engaging in abusive behaviors while hiding the true origin of the attack (unfortunate, not very interesting) or interacting with the victim's internal network, which normally would not be accessible due to the presence of a firewall (a much more problematic case). Intentional change of an IP for this purpose is known as *DNS rebinding*. Browsers try to mitigate DNS rebinding to some extent by, for example, caching DNS lookup results for a certain time (*DNS pinning*), but these defenses are imperfect.

[44] One significant source of same-origin policy bugs is having several separate URL-parsing routines in the browser code. If the parsing approach used in the HTTP stack differs from that used for determining JavaScript origins, problems may arise. Safari, in particular, combated a significant number of SOP bypass flaws caused by pathological URLs, including many of the inputs discussed in Chapter 2.

[45] For example, in Internet Explorer, it will still be possible for one page to navigate any other documents that were nominally same-origin but that became "isolated" after setting *document.domain*, to *javascript:* URLs. Doing so permits any JavaScript to execute in the context of such as a pseudo-isolated domain. On top of this, obviously nothing stops the originating page from simply setting its own *document.domain* to a value identical with that of the target in order to eliminate the boundary. In other words, the ability to make a document non-same-origin with other pages through *document.domain* should not be relied upon for anything even remotely serious or security relevant.

[46] More about this in Chapter 11, but the most notable example is that of encoding data in URL fragment identifiers. This is possible because navigating frames to a new URL is not subject to security restrictions in most cases, and navigation to a URL where only the fragment identifier changes does not actually trigger a page reload. Framed JavaScipt can simply poll *location.hash* and detect incoming messages this way.

[47] In fact, many plug-ins had problems in this area in the past. Most notably, Adobe Flash permitted arbitrary cross-domain HTTP headers until 2008, at which point its security model underwent a substantial overhaul. Until 2011, the same plug-in suffered from a long-lived implementation bug that caused it to resend any custom headers to an unrelated server following an attacker-supplied HTTP 307 redirect code. Both of these problems are fixed now, but discovery-to-patch time proved troubling.

[48] It does not matter that *https://webmail.example.com/* is offered only over HTTPS. If it uses a cookie that is not locked to encrypted protocols, the attacker may simply wait until the victim navigates to http://www.fuzzybunnies.com/, silently inject a frame pointing to http://webmail.example.com/ on that page, and then intercept the resulting TCP handshake. The browser will then send all the *webmail.example.com* cookies over an unencrypted channel, and at this point the game is essentially over.

[49] Even if this possibility is prevented by separating the jars for *httponly* and normal cookies, multiple identically named but differently scoped cookies must be allowed to coexist, and they will be sent together on any matching requests. They will be not accompanied by any useful metadata, and their ordering will be undefined and browser specific.

[50] This IP address is reserved for loopback interfaces; any attempt to connect to it will route you back to the services running on your own machine.

[51] In some contexts, Flash may implicitly permit access from HTTPS origins to HTTP ones but not the other way round. This is usually harmless, and as such, it is not given special attention throughout the remainder of this section.

[52] It should not be assumed that this setting prevents any sensitive data available to a rogue applet from being relayed to third parties. There are many side channels that any Flash applet could leverage to leak information to a cooperating party without directly issuing network requests. In the simplest and most universal case, CPU loads can be manipulated to send out individual bits of

information to any simultaneously loaded applet that continuously samples the responsiveness of its runtime environment.

[53] Because there is no compartmentalization between *file:* cookies, it is unsafe to rely on them for legitimate purposes. Some locally installed HTML applications ignore this advice, and consequently, their cookies can be easily tampered with by any downloaded, possibly malicious, HTML document viewed by the user.