Same-origin policy

From Wikipedia, the free encyclopedia

In computing, the **same-origin policy** is an important concept in the web application security model. Under the policy, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same *origin*. An origin is defined as a combination of URI scheme, hostname, and port number.[1][2] This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's Document Object Model.

This mechanism bears a particular significance for modern web applications that extensively depend on HTTP cookies[3] to maintain authenticated user sessions, as servers act based on the HTTP cookie information to reveal sensitive information or take state-changing actions. A strict separation between content provided by unrelated sites must be maintained on the client-side to prevent the loss of data confidentiality or integrity.

Contents

History

The concept of same-origin policy dates back to Netscape Navigator 2 in 1995. All modern browsers implement some form of the Same-Origin Policy as it is an important security cornerstone.[4] The policies are not required to match an exact specification [5] but are often extended to define roughly compatible security boundaries for other web technologies, such as Microsoft Silverlight, Adobe Flash, or Adobe Acrobat, or for mechanisms other than direct DOM manipulation, such as XMLHttpRequest.

Origin determination rules

The algorithm used to calculate the "origin" of a URI is specified in RFC 6454, Section 4. For absolute URIs, the origin is the triple {protocol, host, port}. If the URI does not use a hierarchical element as a naming authority (see RFC 3986, Section 3.2) or if the URI is not an absolute URI, then a globally unique identifier is used. Two resources are considered to be of the same origin if and only if all these values are exactly the same.

To illustrate, the following table gives an overview of typical outcomes for checks against the URL "**http://www.example.com/dir/page.html**".

Compared URL	Outcome	Reason
http://www.example.com/dir/page2.html	Success	Same protocol, host and port
http://www.example.com/dir2/other.html	Success	Same protocol, host and port
http://username:password@**www.example.com**/dir2/other.html	Success	Same protocol, host and port
http://www.example.com:**81**/dir/other.html	Failure	Same protocol and host but different port
https://www.example.com/dir/other.html	Failure	Different protocol
http://**en.example.com**/dir/other.html	Failure	Different host
http://**example.com**/dir/other.html	Failure	Different host (exact match required)
http://**v2.www.example.com**/dir/other.html	Failure	Different host (exact match required)
http://www.example.com:**80**/dir/other.html	Depends	Port explicit. Depends on implementation in browser.

Unlike other browsers, Internet Explorer does not include the port in the calculation of the origin, using the Security Zone in its place.[6]

Security Concerns

The main reason to have this restriction is because without the same-origin policy there would be a security risk. Assume that a user is visiting a banking website and doesn't log out. Then he goes to any random other site and that site has some malicious JavaScript code running in the background that requests data from the banking site. Because the user is still logged in on the banking site, that malicious code could do anything on the banking site. For example, get a list of your last transactions, create a new transaction, etc. This is because the browser can send and receive session cookies to the banking website based on the domain of the banking website.

A user visiting that malicious site would expect that the site he is visiting has no access to the banking session cookie. While this is true, the JavaScript has no direct access to the banking session cookie, but it could still send and receive requests to the banking site with the banking site's session cookie, essentially acting as a normal user of the banking site. Regarding the sending of new transactions, even CSRF protections by the banking site have no effect, because the script can simply do the same as the user would do.

So this is a concern for all sites where you use sessions and/or need to be logged in. If the banking site from the example (or any other site of course) only presents public data and you cannot trigger anything, then there is usually no danger which the same-origin policy protects against. Also, if the two sites are under control of the same party or trust each other completely, then there is probably no danger either.

Relaxing the same-origin policy

In some circumstances the same-origin policy is too restrictive, posing problems for large websites that use multiple subdomains. Here are some techniques for relaxing it:

document.domain property

If two windows (or frames) contain scripts that set domain to the same value, the same-origin policy is relaxed for these two windows, and each window can interact with the other. For example, cooperating scripts in documents loaded from orders.example.com and catalog.example.com might set their document.domain properties to “example.com”, thereby making the documents appear to have the same origin and enabling each document to read properties of the other. This might not always work as the port stored in the internal representation can become marked as null. In other words, example.com port 80 will become example.com port null because we update document.domain. Port null might not be treated as 80 (depending on your browser) and hence might fail or succeed depending on your browser.[7]

Cross-Origin Resource Sharing

The second technique for relaxing the same-origin policy is standardized under the name Cross-Origin Resource Sharing. This standard extends HTTP with a new Origin request header and a new Access-Control-Allow-Origin response header.[8] It allows servers to use a header to explicitly list origins that may request a file or to use a wildcard and allow a file to be requested by any site. Browsers such as Firefox 3.5, Safari 4 and Internet Explorer 10 use this header to allow the cross-origin HTTP requests with XMLHttpRequest that would otherwise have been forbidden by the same-origin policy.[9]

Cross-document messaging

Another new technique, cross-document messaging allows a script from one page to pass textual messages to a script on another page regardless of the script origins. Calling the postMessage() method on a Window object asynchronously fires an "onmessage" event in that window, triggering any user-defined event handlers. A script in one page still cannot directly access methods or variables in the other page, but they can communicate safely through this message-passing technique.

JSONP

JSONP allows a page to receive JSON data from a different domain by adding a `<script>` element to the page which loads a JSON response with a callback from different domain.

WebSockets

Modern browsers will permit a script to connect to a WebSocket address without applying the same-origin policy. However, they recognize when a WebSocket URI is used, and insert an **Origin:** header into the request that indicates the origin of the script requesting the connection. To ensure cross-site security, the WebSocket server must compare the header data against a whitelist of origins permitted to receive a reply.

Corner cases and exceptions

The behavior of same-origin checks and related mechanisms is not well-defined in a number of corner cases such as for pseudo-protocols that do not have a clearly defined host name or port associated with their URLs (file:, data:, etc.). This historically caused a fair number of security problems, such as the generally undesirable ability of any locally stored HTML file to access all other files on the disk, or communicate with any site on the Internet.

In addition, many legacy cross-domain operations predating JavaScript are not subjected to same-origin checks; one such example is the ability to include scripts across domains, or submit POST forms.

Lastly, certain types of attacks, such as DNS rebinding or server-side proxies, permit the host name check to be partly subverted, and make it possible for rogue web pages to directly interact with sites through addresses other than their "true", canonical origin. The impact of such attacks is limited to very specific scenarios, since the browser still believes that it is interacting with the attacker's site, and therefore does not disclose third-party cookies or other sensitive information to the attacker.

Workarounds

To enable developers to, in a controlled manner, circumvent the same-origin policy, a number of "hacks" such as using the fragment identifier or the `window.name` property have been used to pass data between documents residing in different domains. With the HTML5 standard, a method was formalized for this: the `postMessage` interface,[10][11] which is only available on recent browsers.[12] JSONP can also be used to enable Ajax-like calls to other domains.[13]

See also

- Cross-origin resource sharing (CORS)
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)
- Cross-document messaging

References

1. IETF The Web Origin Concept, Dec, 2011 (https://tools.ietf.org/html/rfc6454)
2. Same Origin Policy - Web Security (http://www.w3.org/Security/wiki/Same_Origin_Policy). W3.org. Retrieved on 2013-08-20.
3. IETF HTTP State Management Mechanism, Apr, 2011 (https://tools.ietf.org/html/rfc6265)
4. "Browser Security Handbook, part 2". google.com. Retrieved 31 January 2014.
5. "Same Origin Policy". W3C. Retrieved 31 January 2014.
6. Lawrence, Eric. "IEInternals - Same Origin Policy Part 1". Retrieved 22 October 2013.
7. LePera, Scott. "Cross-domain security woes". *The Strange Zen Of JavaScript*. Retrieved 4 April 2014.
8. Creating WSGI Middleware (https://www.youtube.com/watch?v=afnDANxsaYo)
9. Cross-Origin Resource Sharing (http://www.w3.org/TR/cors/). W3.org. Retrieved on 2014-11-11.
10. HTML Living Standard. Communication: Cross-document messaging: Posting messages (http://www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html#posting-messages) — WHATWG.
11. HTML5. Communication: Cross-document messaging: Posting messages (http://dev.w3.org/html5/postmsg/#posting-messages) — W3C.
12. When can I use: Support for Cross-document messaging (http://caniuse.com/#feat=x-doc-messaging)
13. "Blog Post: Using CORS with all (modern) browsers".

External links

- A detailed comparison of several flavors of same-origin policies (http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy)
- A review of deficiencies in same-origin policies and their implication for web security (https://web.archive.org/web/20070211191158/http://taossa.com/index.php/2007/02/08/same-origin-policy/) at the Wayback Machine (archived February 11, 2007)
- Sample vendor-provided same-origin policy specification (http://www.mozilla.org/projects/security/components/same-origin.html)
- The HTML5 spec's definition of Origin (http://www.w3.org/TR/html5/browsers.html#origin)
- W3C Article on the Same Origin Policy (http://www.w3.org/Security/wiki/Same_Origin_Policy)

- RFC 6454 on The Web Origin Concept (http://tools.ietf.org/html/rfc6454)
- Blog post: The Cookie Same Origin Policy (http://identitymeme.org/http-cookie-processing-algorithm-etlds/)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Same-origin_policy&oldid=752092627"

Categories: Computer network security | Computer security procedures

- This page was last modified on 29 November 2016, at 13:13.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.