WHITEPAPER
APRIL 2018

netsparker

THE DEFINITIVE GUIDE TO SAME-ORIGIN POLICY

Ziyahan Albeniz

TABLE OF CONTENTS

INTRODUCTION

Back in the 1980s, the Internet was far different than it is today. Internet content was available only via email, special message boards like dial-in Bulletin Board Systems, newsgroups, etc. There was no well-defined rich content to the Internet, only plain text and plain files. But then in 1989, Sir Tim Berners-Lee invented the World Wide Web – a name no longer used, simply called "the Internet" today – as a way to enrich the content available online as something more than just text and data, but also content layouts, text decoration, media embedding, and so forth.

Rather quickly, this idea caught on. Software called "Web Browsers" began to explode in popularity, including Cello, Mosaic, and most especially Netscape Navigator, rendering content generated from documents containing Berners-Lee's Hyper-Text Markup Language (HTML). A few years later in 1994, the "magic cookie" feature was defined as part of the Hyper-Text Transfer Protocol (HTTP) as a way to distinguish users from one another, since web pages had started to become more dynamic and user oriented, and was immediately implemented into Netscape Navigator. The following year, Netscape introduced two new incredible and truly world-changing features to their Navigator web browser to add enhancement to the view of the web: JavaScript, and an API to access the HTML elements, known as the Document Object Model (DOM).

Thanks to DOM, by using the JavaScript language it would be possible to reach all properties of an HTML document – from its URL to cookies contained by document, events triggered by document interactions, and much more. The richness of HTML introduced additional resources, such as other documents or media items, and they in turn have their own cookies, DOM, JavaScript namespace, and other rich elements. Within the scope of the browser, there would need to be a way to securely interact with these entities.

As a solution to this, Netscape engineers decided to manage relations between these resources by using a rule they called Same-origin Policy (SOP), which defines borders for each resource loaded by a browser. According to this rule, all resources loaded by a browser will be defined by a string which is known as the origin, consisting of the protocol, URL, and port being used to reach the resource. Only resources that have same origin can reach one another resource's attributes. But why is this important?

A WORLD WITHOUT SAME-ORIGIN POLICY

Let us say you are somehow tricked into visiting *www.your-bank.bad-site.com*. On that bad site, there is an iframe that loads *www.your-bank.com*, where you proceed to login legitimately. After logging in, a simple JavaScript call on the bad site could be used to access the DOM elements of *www.your-bank.com* loaded in the iframe, such as your account balance.

```
frames.bank_frame.document.getElementById("balance").value
```

What this does is accesses the iframe element (named *bank_frame*), then within that iframe's DOM, it accesses the HTML element named balance and gets its value. This could of course even be extended to forging browser calls to send your money elsewhere! Without Same-origin Policy, these kind of cross-site requests could be executed without your consent or knowledge.

SAME-ORIGIN POLICY IN DETAIL

Imagine we have a web page hosted at http://www.example.com/dir/test.html. Within this document is an iframe that loads a different web page. Our source host is defined as *www.example.com*. The following table depicts the full URLs we want to reach, and whether or not they are reachable due to Same-origin Policy:

URL	Result	Reason
http://www.example.com/dir/page.htm	Accessible	Protocol, host and port match
http://www.example.com/dir2/other.htm	Accessible	Protocol, host and port match
http://www.example.com:81/dir/test.htm	Not Accessible	Same protocol and host, but port is different (81)
https://www.example.com/dir/test.htm	Not Accessible	Same host, but schema/protocol (https) different
http://demo.example.com/dir/test.htm	Not Accessible	Schema and port are same, but host is different (demo.example.com)
http://example.com/dir/test.htm	Not Accessible	Host is different (example.com)
http://example.com/dir/test.htm	Not Accessible	Host is different (www2.example.com)

Because Same-origin Policy is supported by effectively all modern browsers, web resources can reach one another's contents, attributes, and so forth if they use same protocol, same domain and same port; therefore they have same origin. If not, reaching and changing document attributes are prevented by browsers.

In today's world, Same-origin Policy usually is thought as if it is only for DOM. However, this is not entirely true, as all resources on the web have a special Same-origin Policy check mechanism. Cookies are just one example of this. This is because a cookie will be sent only in the event where the cookie domain, path, and attributes match with the domain that is requested. If they match and the cookie is not expired, the cookie will be sent. The only major difference from the previously described Same-origin Policy is that port and schema (except in the event of secure-only cookies) are not subject in controls that are checked before sending a cookie.

Although Same-origin Policy is a concept in the center of web browser security, it is often misunderstood and incorrect assumptions are commonly made about it. Even though Same-origin Policy, at its first glance, looks like a basic origin matching, when you dive into the details, there are far more complexities and ambiguities that can go overlooked.

The most prevalent myth about Same-origin Policy is that it plainly forbids a browser to load a resource from a different origin. Though we know that the thing makes today's web technologies so rich and colorful is the content loaded from different origins. The presence of a huge content delivery network (CDN) ecosystem proves this is not true.

Another prevalent myth is that an origin cannot send information to another one. That is also not true. Again we know that an origin can make a request to another one. The information of the forms in one origin can be reached from another origin. If we think of cloud payment systems integrated into a business workflow, these often operate by sending request to another origin. Even one of the most common web vulnerabilities, Cross-Site Request Forgery (CSRF), arises from that point. CSRF is possible because of the ability of sites to make request to each other.

As an example, when evaluate the behavior of an ambiguous image, we must remember the rules that Same-origin Policy defines:

1. Each site has its own resources like cookies, DOM, and Javascript namespace.
2. Each page takes its origin from its URL (normally schema, domain, and port).
3. Scripts run in the context of the origin which they are loaded. It does not matter where you load it from, only where it is finally executed.
4. Many resources, like media and images, are passive resources. They do not have access to objects and resources in the context they were loaded.

Given these rules, we can assume that a site with origin A:

1. Can load a script from origin B, but it works in A's context
2. Cannot reach the raw content and source code of the script
3. Can load CSS from the origin B
4. Cannot reach the raw text of the CSS files in origin B
5. Can load a page from origin B by iframe
6. Cannot reach the DOM of the iframe loaded from origin B
7. Can load an image from origin B
8. Cannot reach the bits of the image loaded from origin B
9. Can play a video from origin B
10. Cannot capture the images of the video loaded from origin B

Thanks to rules above, rich web contents are well-contained and reasonably secured.

SAME-ORIGIN POLICY IMPLEMENTATIONS

As we have established, Same-origin Policy is a concept at center of the security process with many things of the web: DOM access, Javascript, cookie, etc. But even Rich Internet Applications (RIA) like Silverlight and Java are subject to Same-origin Policy. However, this is where implementation differences of Same-origin Policy come into play, and where developers should be cautious.

Indeed, not only are there sometimes different implementations of Same-origin Policy for various types of web content, but there are also differences defined for how Same-origin Policy applies to cookies, Javascript, and DOM access between different browsers. At times, it can resemble freedom in a minefield.

DOM ACCESS AND SAME-ORIGIN POLICY

Internet Explorer

Although Same-origin Policy that regulates access to DOM exists in all modern browsers, there is – as is typical – a difference from Internet Explorer's (IE) implementation, and all other modern browsers. In the browsers except IE, the items that define origin are schema/protocol + domain + port; whereas in IE, the port is not involved when defining origin. This poses a security risk to the applications run on same domain but in different ports.

If we take an example web page of *http://www.example.com/test.html* we can see where the difference lies between browsers:

URL	Browser	Result	Reason
http://www.example.com:81/contact.html	IE	Accessible	Schema and domain match
http://www.example.com:81/contact.html	Chrome + Firefox + Safari + Opera, etc.	Not Accessible	Although schema and domain match, port does not

Let us examine the risk from a security perspective for this table. Sometimes, test sites can be deployed live on the same domain, but on a different port. For example, a test version of the site *http://www.example.com* can be published on *http://www.example.com:8080*. If a vulnerability exists in test version, for instance a Cross-Site Scripting (XSS) vulnerability, the site published on port 8080 can reach the DOM of the site published on port 80 in Internet Explorer, and vice-versa. If the test version can be published via a simple deployment hook (a git push, a Jenkins deployment, etc.), a rogue developer could sneak in a vulnerability then phish users on a legitimate version of the website (possibly even with the real TLS certificate, too, if deployed behind the same web server or load balancer configuration), creating an opportunity for insider threat.

JavaScript Setting: document.domain

Sometimes the strict rules of Same-origin Policy can cause problems when sharing between sites under the same base domain name. For example, if we have *login.example.com, games.example.com, and calendar.example.com*, how would we communicate between them when the full domain paths do not match? It is possible to relax the Same-origin Policy a little for such a case.

Thanks to the the document.domain JavaScript setting, we can expand our domain restriction to allow everything up to the base domain. If we set the following in our code …

```
document.domain = “example.com”;
```

… then this tells the browser that everything up to example.com is considered within the same origin now, including *login.example.com, games.example.com*, and *calendar.example.com*. Placing this in your JavaScript code is mandatory if these sites which to share their resources with one another.

However, as a caveat, this does not immediately mean that the *login.example.com* site can access the DOM of *example.com*. In order to allow this, the site on *example.com* must also declare the same *document.domain* setting of "*example.com*".

The document.domain JavaScript setting can relax Same-origin Policy restrictions on hostname (the sub-domain elements of the domain), however port and schema restrictions remain the same (but as mentioned previously, port does not apply to Internet Explorer). If we use the example of iframes again, we can demonstrate this when attempting to reach the DOM of the iframe. Assume the URL in blue loads the URL in green via an iframe:

URL	document.domain	iframe URL	document.domain	Result
http://www.example.com	example.com	http://login.example.com	example.com	Accessible
http://www.example.com	example.com	http://login.example.com	example.com	Not Accessible, protocol mismatch
http://payment.example.com	example.com	http://example.com	Not Set	Not accessible
http://www.example.com	example.com	http://www.example.com	Not Set	Not accessible

One crucial element to highlight here is that a domain cannot set its origin to a different domain. For example, *login.example.com* cannot set its origin as *example2.com*. The setting must match where it actually comes from.

SAME-ORIGIN POLICY VS. WEB 2.0

Same-origin Policy is only the basic construct since the earliest days of the world wide web. Ever since then, the internet we know today has exploded into a rich ecosystem of cross-domain content, Content Delivery Networks, single-page design, liking and sharing, and more. In order to support all this diversity and change, Same-origin Policy had to also expand and adapt. Simple DOM and JavaScript namespace control was no longer enough, and required expansion with *XmlHTTPRequest, JSONP, XDomainRequest*, and *CORS*.

XmlHTTPRequest

XmlHTTPRequest is an HTTP communication method (or API) that can be set on HTTP calls to make web applications richer. It enables asynchronous communication between resources to avoid having to load the page new each time. In most cases where the content of the page you are viewing changes without loading a new page, such as while scrolling down – think like Facebook or Twitter – this is most often done via what is called an Asynchronous JavaScript and XML (AJAX) request using the *X-Requested-With* HTTP header set to *XmlHTTPRequest*. Given the potential dangers this can yield, XmlHTTPRequest is also an area where Same-origin Policy rules must be strictly applied.

As such, since the creation of the *XmlHTTPRequest* HTTP API, the requirements of Same-origin Policy apply in full. Therefore:

- * An *XmlHTTPRequest* call can be sent to a site in a different origin, but the reply cannot be read
- Responses can be read if the request URL is in the same origin
- Custom headers can be added only to a request made to the same origin

(* It is important to note that since an *XmlHTTPRequest* call does allow data to be sent to a different origin, this does allow for the potential of Cross-Site Request Forgery attacks.)

Obviously, the *XmlHTTPRequest* Same-origin Policy will pose a problem when using resources from different origins, just as it does with normal DOM access. However, as we will demonstrate, this default behavior can be changed.

JSON Padding (JSONP)

Although you can make an asynchronous request to a different origin by using the XmlHTTPRequest object, you cannot read its response. So, how can we use the web services that make the internet so rich? How can we show the currency rates, weather forecasts, album lists, and many other things received from other sites by using asynchronous requests?

Under the principles of the Same-origin Policy, we know that scripts work in the context of the site on which the scripts were loaded. The only criteria to do this is that the file loaded should be a valid script file. When we combine this information and go back to the early 2000s where the JSON technologies developed, we run into JSONP.

JSON, or Javascript Object Notation, is both a data type and an output considered as valid executable JavaScript code. If the calls we made to third parties return a result in the form of JSON dynamically, we can circumvent the Same-origin Policy limitation.

However, it's not enough to have a response with the content-type application/json returned from the service. These results must be bound with a callback function named by the caller. For example, let's say we have JavaScript code that makes a call to the following URL:

http://www.example.com/getAlbums?callback=foobarbaz

```
foobarbaz([{“artist”: “Michael Jackson”, “album”: “Black or White”}{“artist”: “Beatles,
The”, “album”: “Revolution”}]);
```

The response returned may look something like this:

Let us break this down a little bit.

1. We make a call to the resource *http://www.example.com/getAlbums* and set a callback name in the query string of *foobarbaz*
2. When the resource returns its results as a JSON object, they are encapsulated in a function named *foobarbaz*, as was defined in the query string callback setting

The values within this returned result are now available within the origin of the calling script. Therefore, if this JavaScript call was made from *http://www.example2.com*, it is able to use this data even though the resource is in a different domain.

This does, of course, pose a security risk. When the JSONP request is executed, JavaScript will assume that anything returned from that resource is trustworthy. Therefore, you should validate that the site to which you make JSONP request is a trusted one. Do not forget that all code returned from that site will work in your users' browser under your web site's context. Additionally, with the strong importance of trust of the JSONP endpoint, making requests through a secure HTTPS channel is also important to prevent manipulation that could occur in transmission. This would also require the whole web page also be under HTTPS, due to JavaScript's heightened strictness on mixed content.

XDomainRequest and JSONP vs. CORS

With JSONP and XmlHTTPRequest, one may think we have all our bases covered. We can perform asynchronous calls to other resources, and we can also do dynamic calls to external resources on a different domain. So why, then, do we need XDomainRequest or CORS? And what are they?

If you only intend to support the simplicity of browsers before IE9, Opera 12, Firefox 12, and so forth, you can continue to use JSONP just fine. However, with the possibilities Web 2.0 offers, JSONP has started to become incapable of living up to some aspirations that arose from developers and browser vendors to make Same-origin Policy's cross-domain restriction a little more relaxed still, yet secure. The most essential reason was that cross-domain requests made with JSONP are only one-way, read only. Requests with the opportunity to write were still prevented by Same-origin Policy applied to JSONP. JSONP was also sort of a hacked solution to prior problems, so a more refined and properly designed approach was needed.

Vendors took into account these aspirations created a solution, albeit in two different ways. Microsoft had its own idea, and with Internet Explorer 8 and 9, XDomainRequest became their solution. Whereas with Chrome, Firefox, and other browsers, they implemented a more popular alternative feature known as Cross-Origin Resource Sharing (CORS). Microsoft realized the potential and popularity of CORS and later adopted it in IE10 and beyond. Even though there are two variations, the good news is that with only a few minor differences, CORS and XDomainRequest implementations are almost the same.

If we outline a CORS request, when a site in origin A wants to make a request to a site in origin *B*, first origin *A* must declare its origin in the request by setting a custom HTTP header named *Origin*. The site in *origin B* then returns a response with an HTTP header that defines the origins from which it allows CORS requests. This header is the *Access-Control-Allow-Origin* HTTP header.

By Access-Control-Allow-Origin, it is possible to allow only one site. You can also use this to allow subdomains, for example sub.example.com. Note that it is only possible to allow one FQDN at a time:

```
Access-Control-Allow-Origin: www.example.com
```

Permission can also be granted to all domains on the internet:

```
Access-Control-Allow-Origin: *
```

In the next chapter, we will dive deeper into the mechanics and details of CORS requests.

CROSS-ORIGIN RESOURCE SHARING (CORS) IN DETAIL

There are two valid request types for CORS requests – Simple Request and Preflight Request. Most normal CORS requests will fall under the Simple Request category, consisting of typical HTTP headers and actions. Preflight Requests, however, due to their atypical nature that falls outside the normal trusted scope of a Simple Request, require additional validation against the server.

SIMPLE REQUEST

If the request contains one of the methods of *GET, POST,* or *HEAD*, and the message type is set by the *Content-Type* HTTP header to either *application/x-www-form-urlencoded, multipart/form-data*, or *text/plain*, this request is considered a CORS Simple Request and sent directly to the server. The server will signify whether it accepts the CORS request by the returned *Access-Control-Allow-Origin* HTTP header. If the server accepts, the response will be processed by the client. There are also some additional HTTP headers that can be sent in the request that directly apply to the CORS request, including *Accept* (content types to accept), *Accept-Language* (languages the browser accepts), and *Content-Language* (the language the request is being sent in).

Let us evaluate an example CORS Simple Request and the server's response. An example request may look as follows:

```
GET / HTTP/1.1
Host: cors.example.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en,en-US;q=0.5
Origin: http://www.acceptmeplease.com
Connection: keep-alive
```

The server response would look like something similar to this:

```
HTTP/1.1 200 OK
Date: Sun, 24 Apr 2016 12:43:39 GMT
Server: Apache
Access-Control-Allow-Origin: http://www.acceptmeplease.com
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: application/xml
Content-Length: 423

<?xml version=”1.0” encoding=”UTF-8”?>
...
```

In the request, we perform our CORS Simple Request against *cors.example.com* from our site of *http://www.acceptmeplease.com*, also designating this as our origin. The server responds approving our origin, thereby allowing the browser to continue with the request with Same-origin Policy domain restrictions relaxed.

PREFLIGHT REQUEST

A CORS request that does not fall under the restrictions for a Simple Request are considered Preflight Requests. What this means is if the request method is not *GET, POST,* or *HEAD*; or if the request is *POST* but the *Content-Type* header is not one of *application/x-www-form-urlencoded, multipart/form-data,* or *text/plain*; or if a custom HTTP header is added to the request, then it must be validated against the server first. Before the actual CORS request can be sent to the server, the browser sends a *pre-flight* check request using the *OPTIONS* method. This is more expensive than a normal CORS Simple Request because two HTTP calls must be executed instead of just one. While this additional cost may sound burdensome, it is necessary for the additional benefits CORS Preflight Requests provide that work around core behaviors of web calls, such as working in additional methods and headers.

To better understand this, let us evaluate an example CORS Preflight Request and server response. Say we wish to send a normal CORS Simple Request-type *POST* request, but we are adding an additional HTTP header of *X-Token-ID* and a *Content-Type* header of *application/xml*. This makes the request become a Preflight Request. Thus, our CORS request sends an initial *pre-flight check* request to validate that this is acceptable to the server. That pre-flight looks as follows:

```
OPTIONS /resources/post-here/ HTTP/1.1
Host: cors.example.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Connection: keep-alive
Origin: http://www.acceptmeplase.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-TOKEN-ID
```

The parts marked in bold are crucial to this CORS Preflight Request. Notice first that the request method is *OPTIONS*, which asks the server if the HTTP headers sent are acceptable. Next, we declare our origin, but there are now two additional headers defined: *Access-Control-Request-Method* and *Access-Control-Request-Headers*. These are important because they tell the intentions of the browser in its following request, to send a *POST* method with an additional HTTP header defined.

The server, if accepting this *OPTIONS* request, would respond similar to this:

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache
Access-Control-Allow-Origin: http://www.acceptmeplease.com
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-TOKEN-ID
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

Let us look carefully at the bold HTTP headers returned above. First and foremost, the server must return a 200 response code, otherwise the Preflight Request is considered unacceptable. Second, we have three additional HTTP headers that are returned on top of the normal *Access-Control-Allow-Origin* header:

- Access-Control-Allow-Methods – These are the allowed request methods the browser may send.
- Access-Control-Allow-Headers – These are the approved additional HTTP headers the browser may send.
- Access-Control-Max-Age – If this HTTP header is set in the response, the server wants the browser to cache this *OPTIONS* result for the same kind of request. This allows the browser to make similar requests to resources within this maximum age setting without needing to make an *OPTIONS* request before each of them.

The *Access-Control-Max-Age* is an important HTTP response header to be aware of. In the example above, this value was set as 86,400 seconds, or 24 hours. Each browser defines a maximum value for this field. If the maximum age limit is exceeded for the browser, it will ignore this value and instead substitute its maximum allowed value. For example, this value in Chrome browsers is at most 10 minutes. A glance at the Chrome source code explains that this was implemented to prevent cache poisoning.

Now that our *pre-flight* check has been completed via the *OPTIONS* request, our CORS request can continue as it would with a Simple Request. The key differences, however, are that the *Content-Type* HTTP header change is now allowed, and the additional *X-Token-ID* HTTP header is also allowed. Our follow-up CORS request from our browser will then look like the following:

```
POST /resources/post-here/ HTTP/1.1
Host: cors.example.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Connection: keep-alive
X-Token-ID: aabbccddeeff00112233445566778899OO
Content-Type: application/xml; charset=UTF-8
Content-Length: 55
Origin: http://www.acceptmeplease.com
Pragma: no-cache
Cache-Control: no-cache

<?xml version="1.0"?>
...
```

Notice the two HTTP request headers set in bold. The *X-Token-ID* HTTP header is allowed to be sent, as well as the change to *Content-Type*. The server should now respond similarly as it would with a normal CORS Simple Request.

COOKIES

Thus far, we have demonstrated Cross-Origin Resource Sharing with various HTTP headers, but not the *Cookie* HTTP header – an element we may want to share. By default, credentials used in the browser (including cookies, authentications, and certificates) are not sent along with a CORS request, Simple or Preflight. It should also be noted that under XDomainRequest in IE8 and IE9, under no circumstance could any of these credential data be sent.

If, for example, we wish to pass along the cookie data accessible under our origin, we can simply pass along the contents of the cookie (without the metadata of domain, expiration, etc.) using the *Cookie* HTTP header in our CORS request. This can be added to a Preflight Request *OPTIONS* request, but is not required. In response, the server also must accept and acknowledge this credentialed request by returning the *Access-Control-Allow-Credentials* HTTP header, set to a 'true' value. If this HTTP header is not received, the browser considers the entire request as failed.

IMPLEMENTATIONS

We have explored many of the core concepts of Cross-Origin Resource Sharing, but only the raw HTTP requests and responses. Obviously the browser will not be interacting with the web server on such a low level, and vice-versa. So how is CORS actually implemented?

On the client side, there is little necessary to implement a CORS request, Simple or Preflight. This can be done via a simple JavaScript built-in library known as *XMLHttpRequest*. It is named the same as the Same-origin Policy implementation we mentioned earlier, but it does also support CORS requests. Here is a basic example:

```
// Declare the XMLHttpRequest object
var invocation = new XMLHttpRequest();

// We wish to open a POST method request
invocation.open(‘POST’, ‘http://cors.example.com/sendData, true);

// If we set this option, then in-browser credentials (cookies,
// authentication, certificates) will be sent along with the
// request
invocation.withCredentials = true;

// If we set the following two headers, as described previously,
// this will automatically become a CORS Preflight Request, and
// an OPTIONS method pre-flight check request will be done in
// the background, unless a matching one has already been done
// and was within the site’s (and browser’s) maximum age setting
invocation.setRequestHeader(‘X-TOKEN-ID’, ‘aabbccddeeff00112233’);
invocation.setRequestHeader(‘Content-Type’, ‘application/xml’);

// When the response is returned from the server, we must
// process it via a callback function
invocation.onreadystatechange = function(){ … };

// Send the POST content and initiate the request
invocation.send(‘…’);
```

The comments (in blue) describe what each option does. As you can see, in all actuality in the browser, a CORS request is quite simple.

On the server side, however, this can become far more complex, mainly due to the fact that there are a wide variety of languages, applications, and frameworks to choose from for servers. For one excellent implementation example, though, you can refer to the Server Side Access Control document prepared by the Mozilla Developer Network.

CORS ON SECURITY

The “Sharing” part of Cross-Origin Resource Sharing poses a security risk, both to the browser and the server. For instance, the *Access-Control-Allow-Origin* HTTP header should never be set to * (all origins) unless the resource is truly intended to be publicly accessible. Further, the server should take precaution when setting this HTTP header appropriately. All too often, servers simply repeat back whatever the requesting browser set as its *Origin* HTTP header. This level of blind trust by the server can pose a security risk. Rather, if *Access-Control-Allow-Origin* is intended to be restrictive, then this should never be blindly trusted from the browser, and instead some server-side access control policy should be applied.

Origin headers cannot be changed after the fact in the browser (such as via JavaScript), however if the request is not made via a TLS-encrypted connection, it is possible to change these parameters via a man-in-the-middle attack. Origin should not be a sole indicator of trust, and authentication should not be made by taking only origin into consideration. CORS requests, especially credentialed ones, should always be made via a TLS-encrypted connections. This will also prevent any potential mixed-content vulnerabilities (such as if your origin is over a TLS connection, but your CORS request is against a plain HTTP connection).

Even when the external origin is intended to be trusted, that trust scope should always be as highly restrictive as possible. All activity here should be carefully filtered, as it could all too easily introduce a cross-site scripting (XSS) attack. HTTP headers returned should also be scrutinized and given only as much trust as necessary.

SAME-ORIGIN POLICY FOR RICH INTERNET APPLICATIONS

Even with Cross-Origin Resource Sharing, there was still a limit to what developers of richer media could do. The internet consists of a very wide ecosystem, including browser plugins that go outside the normal scope of DOM and JavaScript namespaces. These plugins – namely Flash, Java, and Silverlight – required more than what the previous Same-origin Policy implementations could offer.

JAVA – A SECURITY NOTE

The Sun (later, Oracle) Java language found itself one of the earliest forms of rich media in a browser outside the normal scope of JavaScript and DOM. Java from the start adhered to the core concepts of Same-origin Policy, requiring matching protocol, domain, and port as typical. However, Java deviated in particular with the domain element, considering resources to be within the same origin even if the domains differed entirely, but they resolved to the same IP address. This still holds through even through Java 8 today.

From a security concept, if for example two websites – *example.com* and *attacker.com* – are hosted on the same shared host, sharing the same IP address, then *example.com* could run JavaScript in the origin of *attacker.com*, effectively bypassing SOP.

FLASH AND SILVERLIGHT

Flash and Silverlight chose to tackle problems of Same-origin Policy via files on the cross-origin rather than via HTTP headers. Flash, for example, requires the existence of a file named *crossdomain.xml* on the external site in order to validate external origin requests. Given that Microsoft generally likes to do things a little special from the rest, Silverlight uses the same file, but additionally also requires a *clientaccesspolicy.xml* file for further context that is not provided normally in *crossdomain.xml*. These files are expected to be in the root directory of the external origin. For example, if a Flash file hosted and loaded from website.com makes a cross-origin request against *flash.example.com/path/to/content.txt*, then the *crossdomain.xml* file must exist at *flash.example.com/crossdomain.xml*.

The *crossdomain.xml* file manages what domains can make a cross-origin request, whether there exists nested *crossdomain.xml* files, what headers can be sent along with the request, and whether the cross-origin requests must be done via TLS-only or not.

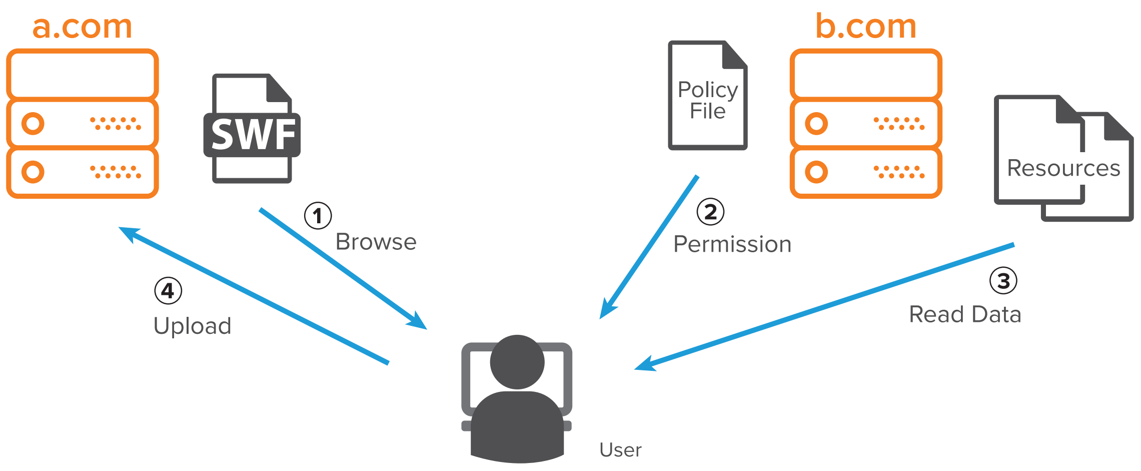
An example *crossdomain.xml* file may look as follows:

```
<?xml version=”1.0”?>
<cross-domain-policy>
  <site-control permitted-cross-domain-policies=”all” />
  <allow-access-from domain=”*” to-port=”” secure=”” />
</cross-domain-policy>
```

Let us break down these tags and their attributes one by one:

- **site-control-permitted-cross-domain-policies** – There are a series of settings that can be defined here
    - **master-only** – Only this *crossdomain.xml* is valid for this origin and all its sub-domains
    - **all** – Sub-directories and sub-domains may have their own nested *crossdomain.xml* settings files
    - **none** – If this is set, *allow-access-from* (explained next) is irrelevant as no cross-origin requests are permitted
- **allow-access-from**
    - **domain** – This attribute defines what domains may use the resources, very much the same as the *Access-Control-Allow-Origin* HTTP header
    - **to-port** – Same-origin Policy by default on most browsers restricts requests to the same port as the origin, however this can be left blank to relax that restriction
    - **secure** – If the request is to be made via TLS-only (HTTPS), setting this to a true value with make Flash fail to load the resource unless the connection is secured

As an example, a Flash cross-origin request may look like this:



1. A user visits *a.com* and their browser loads a Flash SWF file
2. The SWF file from *a.com* wants to make a request to a resource on *b.com*
3. To detect whether this request is valid, the browser loads a request for *crossdomain.xml* from *b.com*
4. If the settings within this *crossdomain.xml* permit this request, then the Flash SWF may continue with its initial request

Security Implications

By using *crossdomain.xml*, it's possible to read data from the target origin hence cross-site request forgery (CSRF) attack prevention can be bypassed! Authentication is also not well managed, and other security measures can be circumvented as well. In effect, the same scenario as outlined at the beginning of this article can take place: Flash could be used to circumvent Same Origin Policies and send attacking traffic to a victim site and read the response, especially if *crossdomain.xml* allows all origins.

Therefore, the *crossdomain.xml* settings should be as restrictive as possible, and carefully managed. Even if the *crossdomain.xml* file in the root web directory does not pose a security risk, if it contains *site-control-permitted-cross-domain-policies="all"*, then *crossdomain.xml* files in sub-directories and sub-domains can open further security risks. The domains allowed should never spread wider in scope than needed, unless this is absolutely intended on being a public endpoint. In that event, the data permitted should be carefully restricted. In general, the policy of least privilege should be applied.

NEXT GENERATION SAME-ORIGIN POLICY

At first we began the "next generation" of the internet with "Web 2.0" – a loosely defined idea of methodologies and visualizations of the new 'modern' web. This was a very fluid and dynamic idea, mostly consisting of concepts rather than standards. Later, with the implementation of HTML5 and the new dynamic this added, the concept of Same-origin Policy also needed to expand.

According to Same-origin Policy, two sites can reach each other's DOM and JavaScript namespace to make asynchronous requests only if their origin matches. However, in HTML5, there is a newly available DOM feature, enabled in JavaScript as *postMessage*, which allows something known as "cross-domain messaging." This is yet another way to allow communication between sites that have a different origin.

The way this works is twofold: First, the starting origin must declare a call to the external origin, and secondly, the external origin must also have a receiver function to handle this event – called a *Message Event*. Let us look at and break down an example of this JavaScript call:

```
otherWindow.postMessage(message, targetOrigin, [transfer])
```

First, we must attach the DOM of the target origin to an object – in this case, the *otherWindow* object. This is a reference value returned from *window.open*, the *contentWindow* property of an iframe, or it can be referenced from a *window.frames* collection. Then, inside the *postMessage()* call, we define three data points:

- **message** – This can be plain text, or it can also be a complex type. This data is transmitted using the HTML5 Structured Clone Algorithm.
- **targetOrigin** – Can be a wildcard (an asterisk, "*"), or a specific value. The receiver's origin is checked against this value, and if valid, the request continues.
- **transfer** (optional) – Enable two-way communication. Channel Messaging can be used here.

The recipient, or external origin, must have defined within its JavaScript namespace an event handler for this *postMessage* event. Here is a brief example of what this could look like:

```
window.addEventListener("message", receiveMessage, false);

function receiveMessage(event){
  if (event.origin == "http://www.example.com")
     event.source.postMessage("Your message: " + event.data, "http://www.example.com");
}
```

In this example, we define an event listener named "message" and tie it to a function named *receiveMessage*. This function gets a single parameter, event, which holds objects such as the sender (*event.source*) and the data sent (*event.data*). Here you can see where we validate the origin of the sender, and how we can even interact back with the sender of this event. It is important to note that *event.origin* specifies the origin at the time when *postMessage* was executed. During the message transmission or communication, the document origin of the windows reached through *event.origin* can be changed.

SECURITY PERSPECTIVE

Cross Domain Messaging, as will any implementation of relaxed Same-origin Policy, comes with its own security risks. The easiest and simplest method to mitigate these risks is to simply not declare a message event handler. However, if one is absolutely needed, much caution needs to be taken to reduce the security risks.

For example, you should almost never use a wildcard (asterisk, "*") for an allowed origin. As mentioned earlier, during message transmission or communication, the location of the target window can be changed. Say two sites, *foobank.com* and *login.foobank.com*, communicate with each other through the *postMessage* API. Upon visiting *foobank.com*, the website opens a new window that navigates to *login.foobank.com* and requests sensitive information. As mentioned earlier during message transmission or communication, the window's location can be changed at any time. As we continue, assume that shortly after *foobank.com* send its message, the location of window changed immediately to an attacker controlled site -- we will call this site badbank.com. The *login.foobank.com* site checked the origin of the message in its message handler function, so it indeed did come from *foobank.com*. However, since it already checked the origin when it received the message, the website does not restrict or specify which sites are allowed to receive the message because it used "*" (wildcard) instead of specific origin (*foobank.com*). However, since the original site got redirected to an attacker controlled site, the *postMessage* API call with the sensitive info will now be sent to the attacker controlled page!

This can also pose a security risk for subdomains of the same primary domain. Say you have a shared hosting site where each user gets their own sub-domain – e.g. *user1.example.com*, *user2.example.com*, etc. Without setting explicit origin validation – such as if you only check that "example.com" exists within the origin, e.g. with a JavaScript *indexOf()* conditional test – this also can open a security vulnerability.

Finally, as mentioned multiple times previously, trust should always be within a very limited scope when dealing with external resources, especially when you are not in control of them. For example, if the data returned is simply pasted to the browser as raw HTML, this easily opens a security vulnerability of XSS and other problems. Instead, data returned should be checked and should be used with safe APIs as explained in our DOM XSS article, rather than blindly trusted and used as-is.

FINAL THOUGHTS AND CONCLUSION

Same-origin Policy is an ever-evolving construct of the world wide web. We can see this, for example, with the evolution of cookies. Cookies were invented before DOM and JavaScript, and so their Same-origin Policy adaptations were tacked on later. We can see this in the contrast with normal Same-origin Policy, where with cookies the schema and port are not considered. After JavaScript was invented, there was an *httpOnly* flag added which deemed a cookie usable only via HTTP headers and not within the JavaScript namespace. Only later was the *secureOnly* flag added to tackle the problem of cookies within a TLS-only scope.

Same-origin Policy, while at the center of client-side web security, is wide and quite varied, differing in important details from browser to browser and between technological implementations. It is a crucial concept to understand, but more importantly, the differences and pitfalls must be equally taken into consideration. When developers (and indeed security engineers) understand and properly implement the Same-origin Policy that suits their needs, it makes for a richer web, indeed.

Author:
Ziyahan Albeniz

Translators:
Alex Baker
Emre İyidogan