**SOP, CORS & CSP**

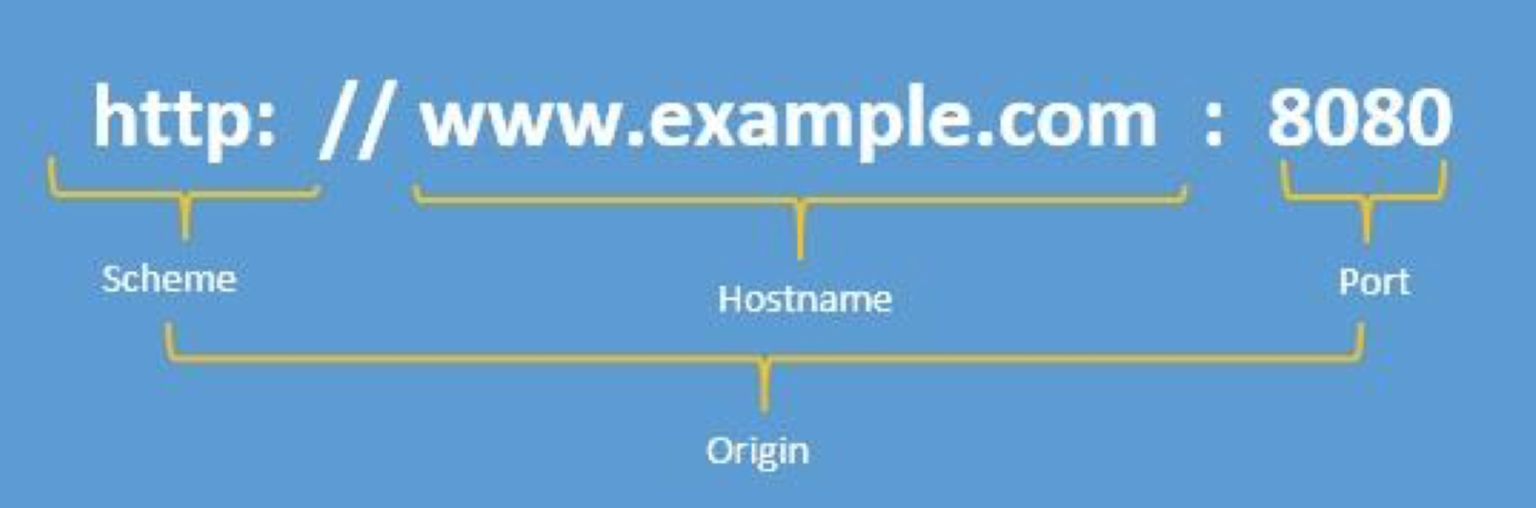
**SOP – Same Origin Policy -** The same origin policy was created in 1995 by Netscape Navigator

One of the core and most important web Security Policy –

* Restriction policy on the resources interaction - which controls how JavaScript accesses documents across different origins
* To restrict the cross-site issues with origin

*Origin –*

Same Origin – URI Scheme (Protocol), Host and port number (if one is specified & Excluding IE) are the same for both pages



One more thing should be noted here that SOP is inconsistent and vendor specific policy and different browsers have different implementations when it comes to SOP. A good example would be in case of Internet explorer an origin consists of a scheme and a host and it does not take ports into consideration.

So, if there is a SOP bypass for one browser, it is not likely to affect other browsers unless the vulnerability exists in a plugins such as flash or java plugins.

Example – Origin comparison

<http://merchant.store.com/folder/page.html>

|  |  |  |
| --- | --- | --- |
| **URL** | **Outcome** | **Reason** |
| http://merchant.store.com/folder2/other.html | Success | - |
| http://merchant.store.com/folder/inner/another.html | Success | - |
| https://merchant.store.com/secure.html | Failure | Protocol mismatch |
| http://merchant.store.com:81/folder/etc.html | Failure | Port mismatch |
| http://products.store.com/folder/other.html | Failure | Host mismatch |

*Common Misconception –*

“*SOP means that one page cannot send data to a different server*” – WRONG !

“*SOP means that one page cannot use resources from a different server*” – WRONG !

“*SOP means that pages from one site are completely immune to tampering by other sites” –* Again WRONG !

*Live example of blocking SOP for GET request >> Demo TBD..*

<http://mysite1.com> <http://mysite2.com>

ajax

Normal BLOCKED Response

**Same origin policy for accessing DOM**

A webpage inside the frame or iFrame is not allowed to modify or access the DOM of its parent or top page and vice-versa if both pages don’t belong to same origin.

There are three ways of bypassing this restriction

* *window.document.domain*
* *Proxy*
* *Cross Document Messaging*

**Same Origin Policy for AJAX**

A webpage cannot make a AJAX Request to an another page if they both don’t belong to the same origin policy. There are basically three ways of bypassing this restriction

* *Proxy server*
* *JSONP*
* *Cross-Origin Resource Sharing*

**Same Origin Policy for Cookies**

Same Origin Policy for cookies works with different manner. A page can set a cookie for its own domain or any parent domain.

For example :

* www.example.com can set a cookie for example.com.
* www.blog.example.com can set cookies for blog.example.com and example.com.
* blog.example.com cannot set cookie for news.example.com because news.example.com is not a parent of blog.example.com.

Note that a page cannot set cookie for public suffix. For example www.example.com cannot set cookie for .co, .uk, .in etc.

The browser will make a cookie available to the given domain including any sub-domains.

For example:  
A cookie of example.com will be send to example.com, www.example.com and www.blog.example.com.

**Same Origin Policy for data storage**

Data stored in the browser (inside localStorage and IndexedDB) are allowed to be accessed only from the same origin which stored it. Two webpages (parent and frame) from different origins can access each other’s local storage by using Cross Document Messaging API or window.name variable.

**Security Applications –**

* Critical security mechanism - Isolation of potential malicious documents
* It protects the site which uses authenticated sessions – A potential risk example of banking site logged-in in same browser running with malicious website requests the data from the banking site
* CSRF - Since the <script> tag doesn’t respect the same-origin policy, it is possible for a malicious website to get sensitive data using the same URL. Using the example above, a malicious page could download the same JSON data and perform some unkind act with it. This is known as a Cross Site Request Forgery (CSRF) attack. There are countermeasures to prevent CSRF attacks include using tokens or cookies for validation, and limiting the lifetime of such tokens.

**Now How to Relax Same origin Policy –**

*Changing Origin*

Sometimes, the SOP may block requests between subdomains on the same domain. The easiest way to solve this problem is to set document.domain from within JavaScript. For example:

document.domain = 'example.com';

*Using a Web Proxy*

Although specifying document.domain helps to contact subdomains of the website, now what would we do if we needed to access data from a different domain altogether?

So, there is a way to configure a web proxy on the webserver –

Instead of sending a request directly from our domain (http://www.example.com/) to a new domain (<http://www.newexample.com/>), Instead send a request to our own server (http://www.example.com/connect/), which in turns sends a request to the new domain (<http://www.newexample.com/>).

*Using WebSockets*

Nowadays most of the web 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.

*Cross-document messaging*

A 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.

*Using JSONP*

Another way of implementing cross browser requests is by using JSONP, or “JSON with padding.” JSONP takes advantage of the fact that <script> tags are not subject to the same-origin policy. For example, we can include a library like jQuery on our page even if it is hosted on any other CDN.

*Switching to Cross-Origin Resource Sharing – CORS*

Although JSONP can be used to accomplish most tasks with relative ease, there are several shortcomings. You can only send HTTP GET requests using JSONP. Along with this there was another security concern about use of custom proxy.

This extends HTTP with a new Origin request header and a new Access-Control-Allow-Origin response header. 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.

Hence switching to CORS – works by modifying HTTP headers in the requests to access resources on a different domain. In IE8+, simple CORS requests using the XDomainRequest (instead of the XMLHttpRequest) are permitted.

A simple example is shown below.

*request = new XDomainRequest();*

*request.open(method, url);*

*request.onload = function() {*

*callback(req.responseText);*

*};*

*request.send(data);*

**CORS – Cross origin resource sharing**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Cross-Origin Resource Sharing (CORS) is a technique for relaxing the same-origin policy, allowing Javascript on a web page to consume a REST API served from a different origin.

Cross-Origin Resource Sharing is a mechanism that uses additional [HTTP](https://developer.mozilla.org/en-US/docs/Glossary/HTTP) headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin. A web application makes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, and port) than its own origin.

CORS vs JSONP

CORS can be used as a good alternative to the [JSONP](https://en.wikipedia.org/wiki/JSONP). As JSONP supports only the GET request method, CORS also supports other types of HTTP requests. Using CORS enables a web developer to use regular XMLHttpRequest, which supports better error handling than JSONP. On the other hand, JSONP works on legacy browsers which doesn’t support CORS. CORS is supported by most modern web browsers. Also, while JSONP can cause [cross-site scripting (XSS)](https://en.wikipedia.org/wiki/Cross-site_scripting) issues when the external site is compromised, CORS allows websites to manually parse responses to ensure security.

CORS is handled by browser but must be managed by Server.

Major Players in CORS Lifecycle:

1. Server: The server supports CORS
2. Browser: The browser enforces CORS on behalf of client.

Browsers that supports CORS:

The below browsers support CORS:

* Chrome 3+
* Firefox 3.5+
* Opera 12+
* Safari 4+
* Internet Explorer 8+

The CORS Lifecycle flow

The CORS lifecycle is as follows:

1. Client makes a CORS request
2. Browser enforces CORS
3. Server support CORS (using response headers)
4. Browser returns a valid response back to client.

mysite.com mysite-api.com

GET/get-info

Origin: mysite.com

Over AJAX

Access-Control-Allow-Origin: \*

{

“info: public info”

}

**Access Control Scenarios**

About Simple Requests

Those request doesn’t trigger a CORS preflight, those called “Simple Request” -

Talking about Simple request –

• The only allowed methods are:

* + *GET*
  + *HEAD*
  + *POST*

• Apart from the headers set automatically by the user agent (for example, Connection, User-Agent, or any of the other headers with names defined in the Fetch spec as a “forbidden header name”), the only headers which are allowed to be manually set are those which the Fetch spec defines as being a “CORS-safelisted request-header”, which are:

* + *Accept*
  + *Accept-Language*
  + *Content-Language*
  + *Content-Type (but note the additional requirements below)*
  + *Last-Event-ID*
  + *DPR*
  + *Save-Data*
  + *Viewport-Width*
  + *Width*

• The only allowed values for the Content-Type header are:

* + *application/x-www-form-urlencoded*
  + *multipart/form-data*
  + *text/plain*

The request includes an **Origin header** that indicates the origin of the client code.

The server will consider the request's Origin and either allow or disallow the request. If the server allows the request, then it will respond with the requested resource and an **Access-Control-Allow-Origin header** in the response. This header will indicate to the client which client origins will be allowed to access the resource. Assuming that the *Access-Control-Allow-Origin* header matches the request's Origin, the browser will allow the request.

Otherwise, if *Access-Control-Allow-Origin* is missing in the response or if it doesn't match the request's Origin, the browser will disallow the request.

For example –

For example, suppose that client code served from foo.client.com were to send the following request for a resource at bar.server.com:

*var invocation = new XMLHttpRequest();*

*var url = 'http://bar.server.com/greetings/';*

*function callOtherDomain() {*

*if(invocation) {*

*invocation.open('GET', url, true);*

*invocation.onreadystatechange = handler;*

*invocation.send();*

*}*

This will lead to handle below client server request with CORS privileges –

Client Server

Simple request

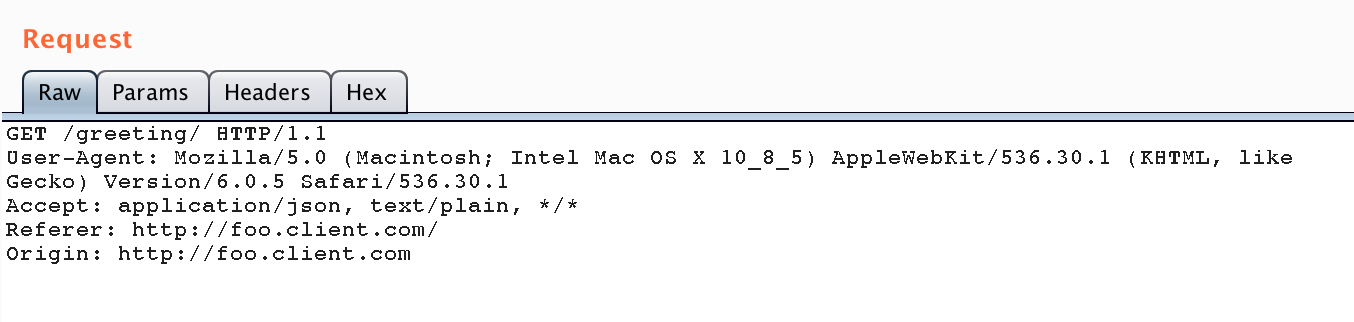
GET / doc HTTP/1.1

Origin: foo.client.com

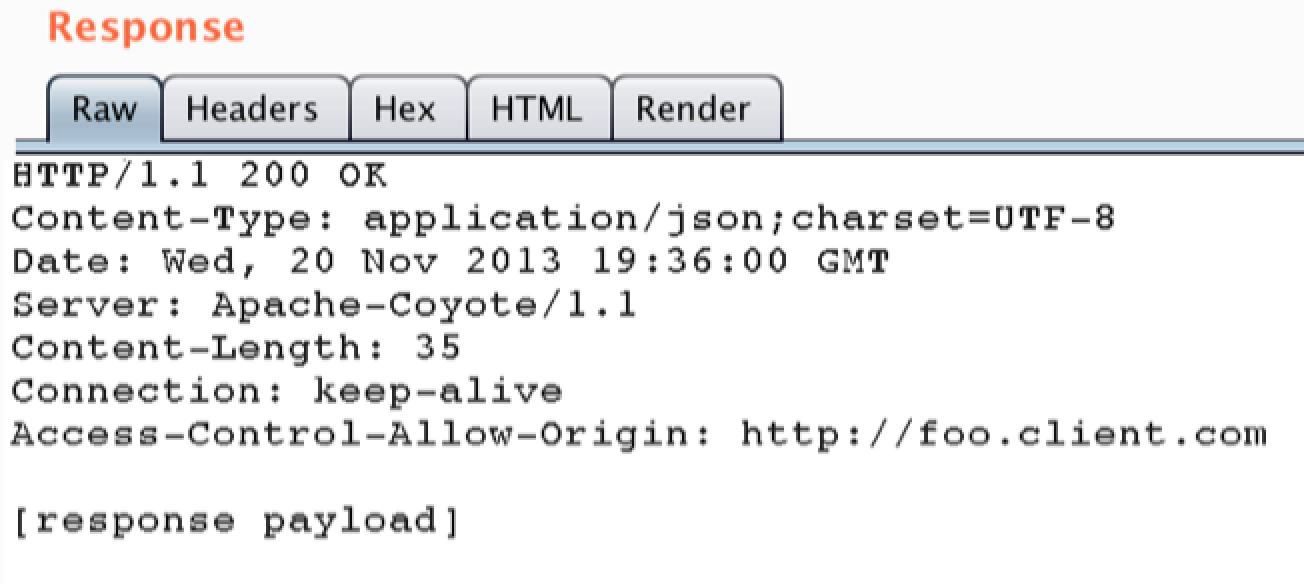
HTTP/1.1 200 OK

Access-Control-Allow-Origin: foo.client.com

Let’s see how browser will send the request and server responds to it –



The Origin header tells the server that the client code originated from <http://foo.client.com>. So it checks its same-origin policies and determines that it can serve the request. The response might look like this:



The **Access-Control-Allow-Origin** indicates that "<http://foo.client.com>" is allowed access, but no other origins will be allowed access.

So the above code will be like –

*if($\_SERVER[“HTTP\_ORIGIN”] == “http://foo.client.com”)*  
*{*  
*header(“Access-Control-Allow-Origin: http://foo.client.com”);*  
*}*

Also,**Access-Control-Allow-Origin** may be set to "**\***" to indicate that all client origins are allowed. This is considered an unsafe practice, however, except in special cases where an API is completely public and is expected to be consumed by any client

What about Not-So-Simple Requests – “preflight” requests

Browser

Server

JavaScript Code

xhr.send()

preflight request (if necessary)

preflight response (if necessary)

actual request

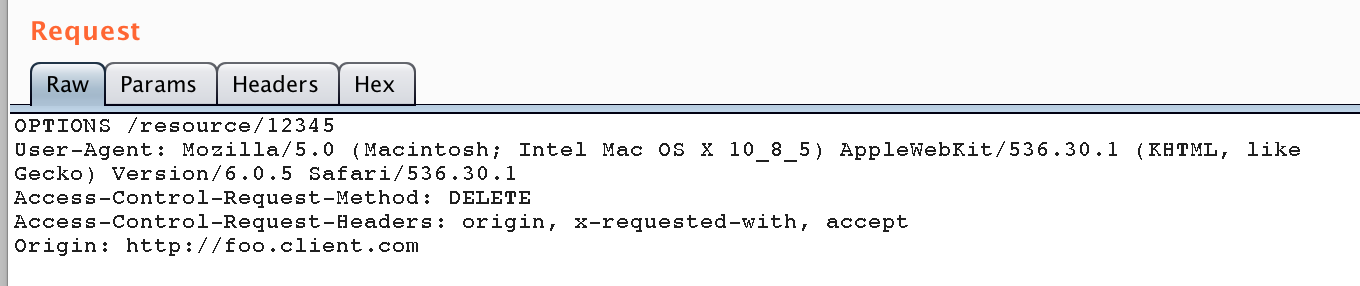
actual response

file onload() or onerror()

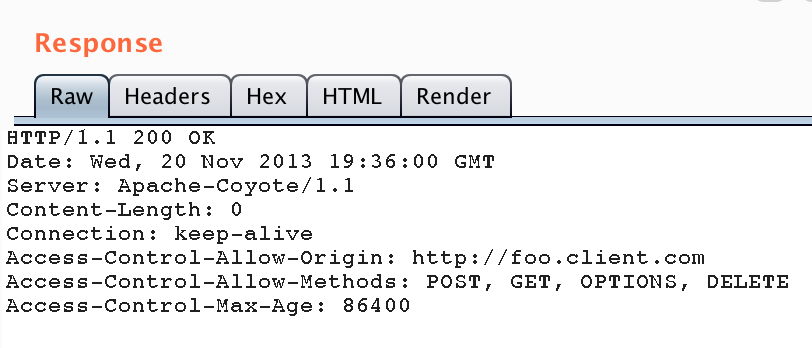
CORS introduced the preflight request model to support backward compatibility. Let’s say that your web server does not support CORS, but browsers have implemented CORS. This means that your web server will get CORS requests that it does not know how to respond to.

To avoid the such type of surprises, the browser sends preflight request and ask servers if they support CORS and allow requests with that origin, containing methods and headers. If not, the browser will not make the actual request. GET, POST, HEAD and OPTIONS are all requests that server understands, so no preflight request are initiated from browser.

For example, suppose that a client served from foo.client.com performs a DELETE request against a resource at bar.server.com. The preflight request takes the form of an OPTIONS request with the following headers:



The preflight request is essentially asking the server if it would allow the DELETE request, without actually sending the DELETE request. If the server allows the original request, then it will respond to the preflight request like this:



The response to the preflight request indicates (in the **Access-Control-Allow-Methods** header) that the client is allowed to issue a DELETE request for the given resource.

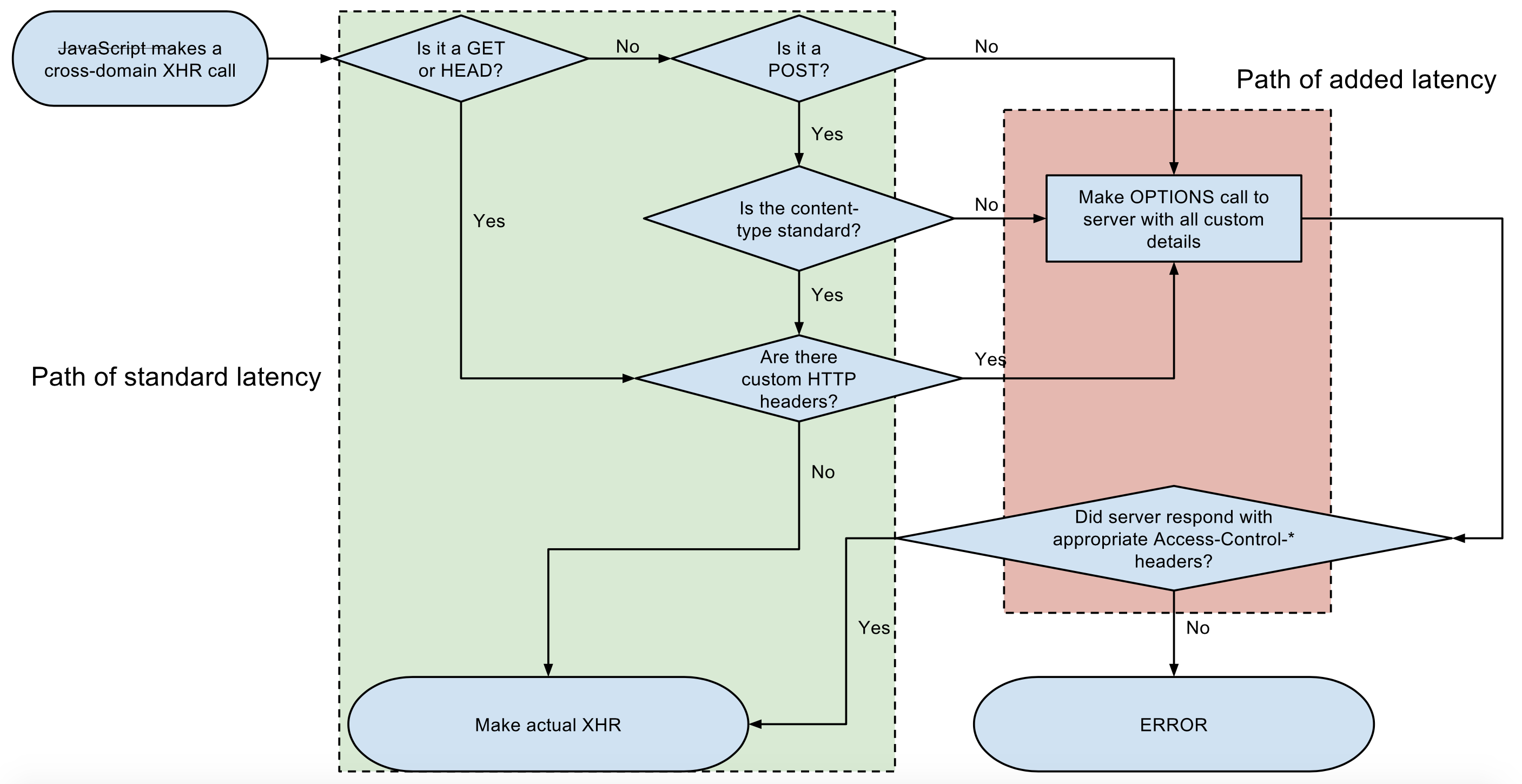
The **Access-Control-Max-Age** indicates that this preflight response is good for 86,400 seconds, or 1 day, after which a new preflight request must be issued.

In the meantime, the client will be allowed to send the original DELETE request for the resource.

*Limitations with IE – IE >=10*

*Some javascript libraries, angularJS, Snotch by default supports to preflight requests only.*

**Basic difference for preflight check is required or not -**



*[image source - wikipedia]*

**While sending Cookies to a cross domain using CORS**

As, we have made simple and preflights request. However, in neither have we sent cookies in a cross-domain request.

By default, CORS requests do not send or set cookies. If you need to send cookies as part of the request, you will need to set the XHR. **WithCredentials** property to **True**.

As we have seen in previous demos, the server must allow Origin, Method and Headers that the client sends. Else, in the browser, the preflight response server will not make the actual request. Similarly, before sending cookies, the server must be enabled by setting the **Access-Control-Allow-Credentials** response header to True.

The **.withCredentials** property sends and sets cookies from the remote domain in the request. However, cookies will continue to honor the SOP, which means that you cannot access cookies from the client JavaScript code; only the remote domain will be able to control these cookies.

The HTTP request headers

**Origin -**

The Origin header indicates the origin of the cross-site access request or preflight request.

*Origin: <origin>*

**Access-Control-Request-Method -**

The Access-Control-Request-Method is used when issuing a preflight request to let the server know what HTTP method will be used when the actual request is made.

*Access-Control-Request-Method: <method>*

**Access-Control-Request-Headers -**

The Access-Control-Request-Headers header is used when issuing a preflight request to let the server know what HTTP headers will be used when the actual request is made.

*Access-Control-Request-Headers: <field-name>[, <field-name>]\**

**Access-Control-Max-Age -**

The Access-Control-Max-Age header indicates how long the results of a preflight request can be cached. For an example of a preflight request, see the above examples.

*Access-Control-Max-Age: <delta-seconds>*

**Access-Control-Allow-Credentials -**

The Access-Control-Allow-Credentials header Indicates whether or not the response to the request can be exposed when the credentials flag is true. When used as part of a response to a preflight request, this indicates whether or not the actual request can be made using credentials. Note that simple GET requests are not preflighted, and so if a request is made for a resource with credentials, if this header is not returned with the resource, the response is ignored by the browser and not returned to web content.

*Access-Control-Allow-Credentials: <true>*

**Access-Control-Allow-Methods –**

The Access-Control-Allow-Methods header specifies the method or methods allowed when accessing the resource. This is used in response to a preflight request. The conditions under which a request is preflighted are discussed above.

*Access-Control-Allow-Methods: <method>[, <method>]\**

**Best Test Cases to consider for CORS security Misconfiguration -**

* POOR Implementation, but BEST case for attack:

*Access-Control-Allow-Origin: https://attacker.com*

*Access-Control-Allow-Credentials: true*

* POOR Implementation, but EXPLOITABLE:

*Access-Control-Allow-Origin: null*

*Access-Control-Allow-Credentials: true*

* BAD Implementation, but not exploitable:

*Access-Control-Allow-Origin: \**

*Access-Control-Allow-Credentials: true*

or simply

*Access-Control-Allow-Origin: \**

As this is not good implementation due to own rules of CORS if *Access-Control-Allow-Origin* set to \* [we don’t get benefit](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS#Requests_with_credentials) *Access-Control-Allow-Credentials: true* it means no cookie access of the victim.

Also sometimes we can’t exploit even if above misconfigurations are present:

* Presence of any custom header in the request which is getting used to authenticate the user.
* Presence of any unique/authentication/key in the request URI

Security Concerns with CORS misconfiguration –

* CORS vulnerability falls under OWASP Top 10 mentioned Security Misconfiguration, The HTTP response header ‘Access-Control-Allow-Origin’ is not configured correctly and this creates the issue.
* Misconfigured Access-Control-Allow-Origin to \* headers gives chance to attacker to retrieve sensitive information, if applicable.
* Cross site scripting like attacks if whitelisted domain/ sub-domains are vulnerable to XSS, if even any one of the subdomains of the whitelisted domain is vulnerable to other exploits such as XSS then it can enable CORS exploitation.
* CSRF protection bypass with misconfigured Access-Control-Allow-Origin header to \*
* Possible MitM attacks due to Cross Origin Resource Sharing (CORS) being enabled over the unencrypted HTTP protocol.

**CSP – Content Security policy**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

CSP is developed by the World Wide Web Consortium (W3C), CSP is a new standard that allows developers to define restrictions on the behaviour of a website or application. For example: from which external locations is it allowed to load scripts, stylesheets or images? Is the browser allowed to load the website in an iframe from a different website?

**Why**

A well-adjusted CSP can prevent multiple attacks and vulnerabilities such as *cross-site scripting (XSS)* or *clickjacking*, data injection attacks and packet sniffing attacks.

**Security Applications**

*Preventing cross site scripting*

Important role of CSP is to mitigate and report XSS attacks. XSS attacks exploit the browser's trust of the content received from the server. Malicious scripts are executed by the victim's browser because the browser trusts the source of the content, even when it's not coming from where it seems to be coming from.

CSP makes it possible for developers to reduce or eliminate the attack vectors by which XSS can occur by specifying the domains that the browser should consider to be valid sources of executable scripts. A CSP compatible browser will then only execute scripts loaded in source files received from those whitelisted domains, ignoring all other script

*Mitigating packet sniffing attacks*

Along with to restrict the domains from which content can be loaded, the server can specify which protocols are allowed to be used; for example (and ideally, from a security standpoint), a server can specify that all content must be loaded using HTTPS. A complete data transmission security strategy includes not only enforcing HTTPS for data transfer, but also marking all cookies with the secure flag and providing automatic redirects from HTTP pages to their HTTPS counterparts. Sites may also use the *Strict-Transport-Security* HTTP header to ensure that browsers connect to them only over an encrypted channel**.**

**How to set up CSP**

To enable CSP, you need to configure your web server to return the [Content-Security-Policy](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy) HTTP header (sometimes you will see mentions of the X-Content-Security-Policy header, but that's an older version and you don't need to specify it anymore).

Alternatively, the <*meta*> element can be used to configure a policy, for example: <meta http-equiv="*Content-Security-Policy*" content="default-src 'self'; img-src https://\*; child-src 'none';">

Webservers can add a special *header* named **Content-Security-Policy** to every response. For example, this website has set the following header:

Content-Security-Policy: default-src https://example.com https://example.com; script-src 'none'; frame-ancestors 'none'; block-all-mixed-content; report-uri /csp-violations/;

**What's the meaning of the attributes?**

There are multiple attributes can be defined in CSP header. Refer the below source to find list of available headers,

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>

Some of them important as below –

* *default-src:* this is the fallback for when other attributes have not been defined. Most of the times this attribute has the value 'self'indicating that only resources from the website itself can be loaded.
* *script-src:* restricts the locations external scripts can be loaded from. Value can be set to 'none' if the website or application doesn't use client side scripts.
* *img-src:* restricts the locations images can be loaded from.
* *media-src:* restricts the locations media (such as videos) can be loaded from.
* *object-src:* restricts the locations plugins can be loaded from.
* *frame-ancestors:* this attribute restricts the locations that may load the website using frame, iframe, object, embed or applet elements. In due course frame-ancestors should replace the HTTP response header X-Frame-Options.
* *form-action:* restricts the URLs that can be used as part of a <form> action. In other words, form-action restricts where the browser can send the data from completed forms. *It's important to note that form-action does not fallback on the default-src, so be sure to define this if your website or application uses form-elements.*

CSP can also be used to enforce that resources can only ever be loaded using HTTPS by adding the https:// protocol to every value defined for the \*-src attributes. This prevents resources from being loaded over non-encrypted HTTP connections. The same effect can be achieved by adding the attribute block-all-mixed-content.

Example usage of the common use cases

1. A web developer wants all content to come from the site's own origin (this excludes subdomains.)

***Content-Security-Policy:*** *default-src 'self'*

1. A web developer wants to allow content from a trusted domain and all its subdomains (it doesn't have to be the same domain that the CSP is set on.)

***Content-Security-Policy:*** *default-src 'self' \*.trusted.com*

1. A web developer wants to allow users of a web application to include images from any origin in their own content, but to restrict audio or video media to trusted providers, and all scripts only to a specific server that hosts trusted code.

***Content-Security-Policy:*** *default-src 'self'; img-src \*; media-src media1.com media2.com; script-src userscripts.example.com*

1. A web developer for an online banking site wants to ensure that all its content is loaded using SSL, in order to prevent attackers from eavesdropping on requests.

***Content-Security-Policy:*** *default-src* [*https://onlinebanking.jumbobank.com*](https://onlinebanking.jumbobank.com)

1. A web developer of a web mail site wants to allow HTML in email, as well as images loaded from anywhere, but not JavaScript or other potentially dangerous content.

***Content-Security-Policy:*** *default-src 'self' \*.mailsite.com; img-src \**

*default-src -* means that scripts can be loaded only from the originating server.

**Server Side Configuration**

Any server side programming environment should allow you to send back a custom HTTP response header. You can also use your web server to send back the header.

***Apache Content-Security-Policy Header***

Add the following to your httpd.conf in your VirtualHost or in an .htaccess file:

*Header set Content-Security-Policy "default-src 'self';*"

***Nginx Content-Security-Policy Header***

In your server {} block add:

*add\_header Content-Security-Policy "default-src 'self';";*

***IIS Content-Security-Policy Header***

You can use the HTTP Response Headers GUI in IIS Manager or add the following to your web.config:

*<system.webServer>*

*<httpProtocol>*

*<customHeaders>*

*<add name="Content-Security-Policy" value="default-src 'self';" />*

*</customHeaders>*

*</httpProtocol>*

*</system.webServer*>

**References:**

[**https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\_policy**](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy)

[**https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS**](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS)

[**https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP**](https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP)

[**https://www.owasp.org/**](https://www.owasp.org/)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*