Cross-App Security III

Cross-Origin Resource Sharing

Cross-Origin Resource Sharing

A mechanism for servers to relax default browser protections applied to cross-origin requests, permitting additional request types and data reading capabilities

Fetch API

Defined in Fetch Standard (WHATWG)

“To allow sharing responses cross-origin and allow for more versatile fetches than possible with HTML’s form element, the CORS protocol exists. It is layered on top of HTTP and allows responses to declare they can be shared with other origins. “

https://fetch.spec.whatwg.org/

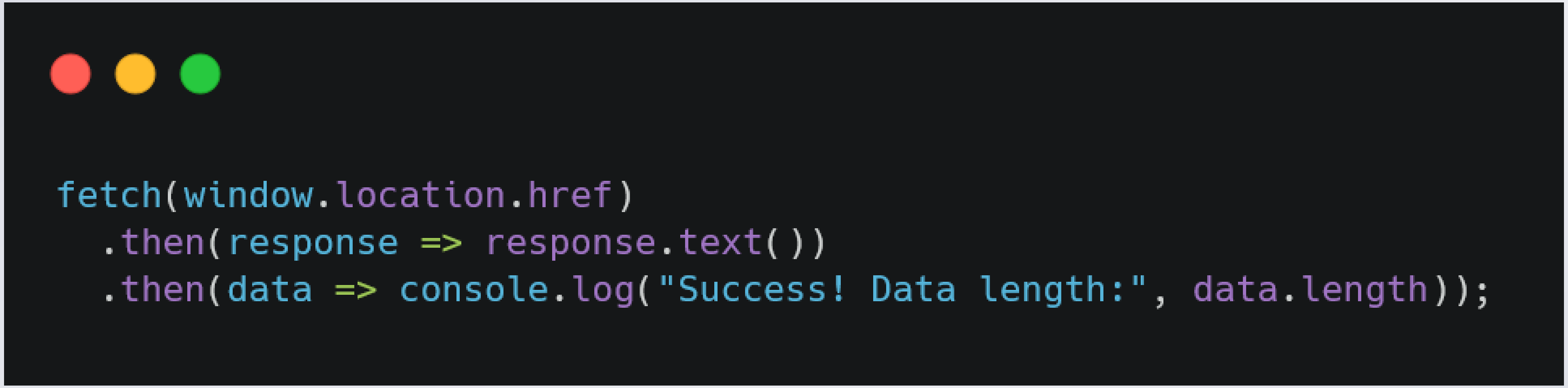
Fetch API

- Modern JavaScript API for sending HTTP requests, replacing XMLHttpRequest (XHR), which is still common
- Implemented and consumed by browsers (primarily)
- Browsers enforce security rules
- Fetch is promise-based and asynchronous
- Fulfills the role of AJAX (asynchronous JavaScript + XML)
- Configurable to send a wide-range of request types

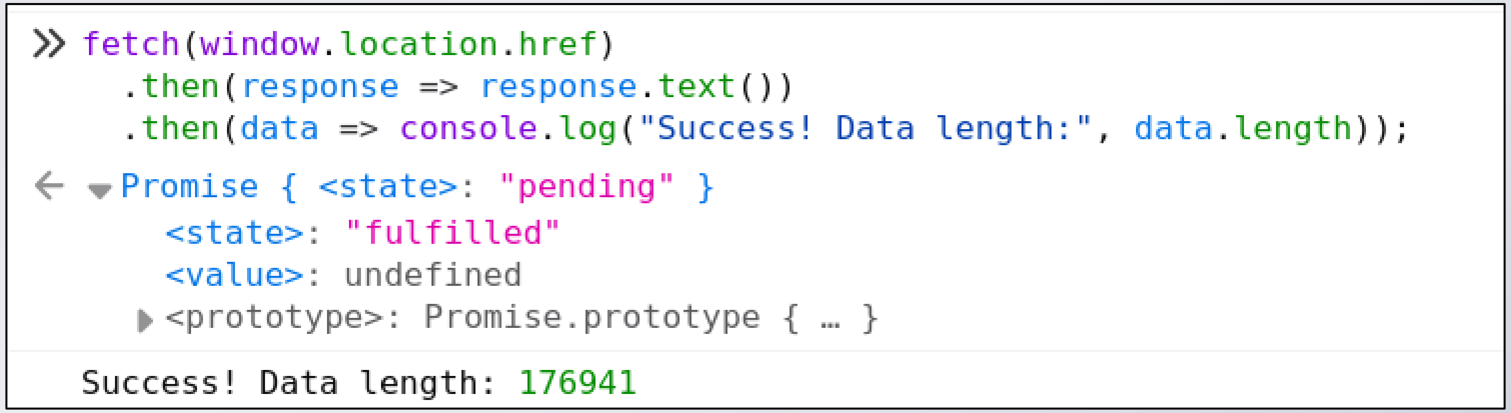
Note: we are ignoring XHR for this lesson as it follows the same rules as Fetch for security

Fetch API: Same-Origin Request

Requesting Origin: https://www.google.com
Target Origin: https://www.google.com

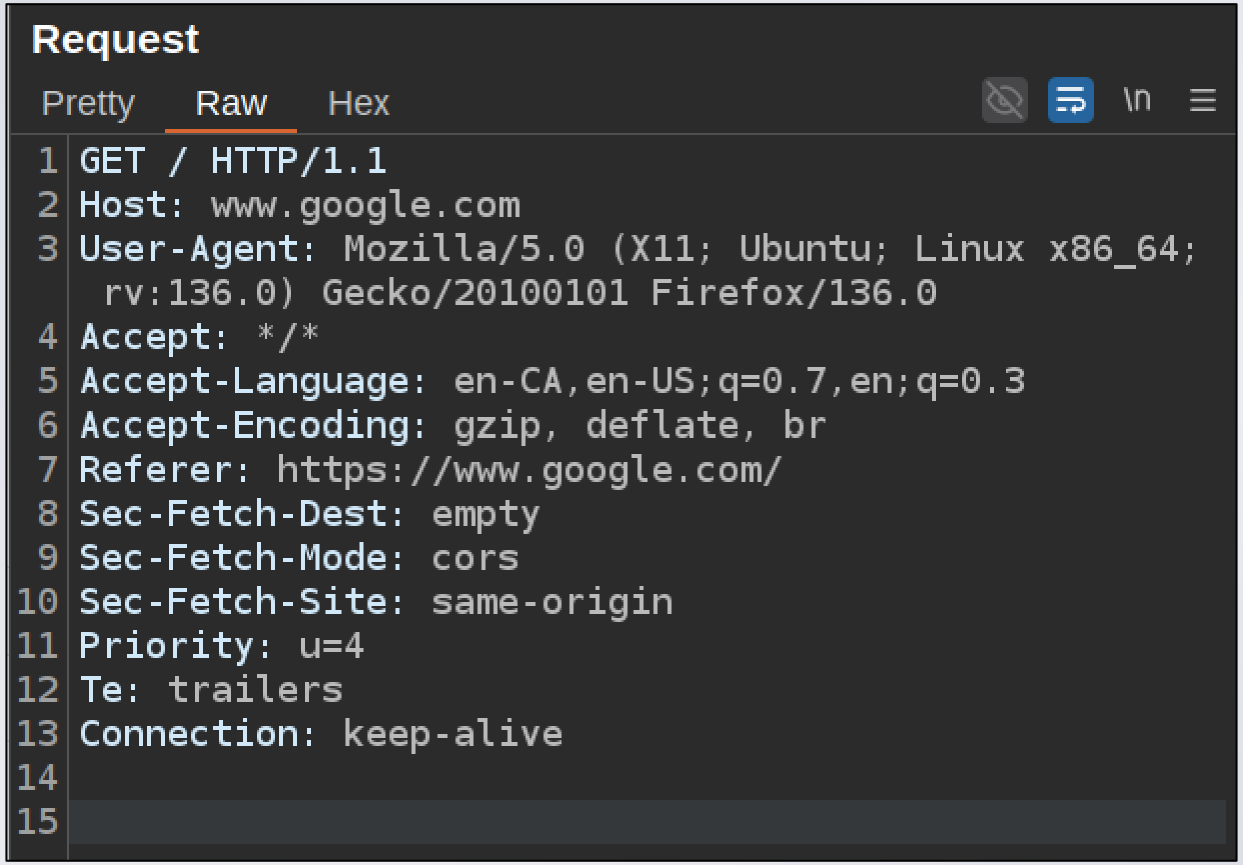


```
fetch(window.location.href)
  .then(response => response.text())
  .then(data => console.log("Success! Data length:", data.length));
```



```
» fetch(window.location.href)
    .then(response => response.text())
    .then(data => console.log("Success! Data length:", data.length));
← ▼Promise { <state>: "pending" }
      <state>: "fulfilled"
      <value>: undefined
    ▶<prototype>: Promise.prototype { … }
  Success! Data length: 176941
```

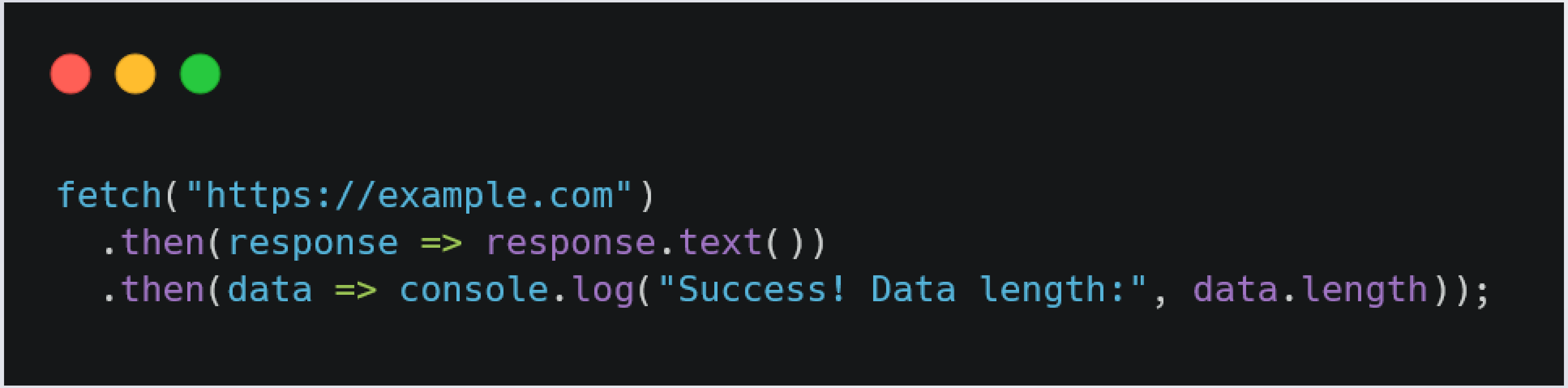
Fetch API: Same-Origin Request



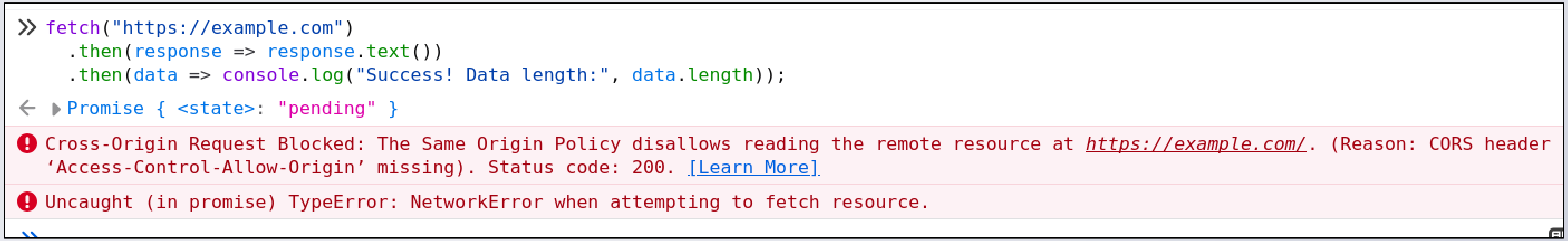
```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:136.0) Gecko/20100101 Firefox/136.0
Accept: */*
Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate, br
Referer: https://www.google.com/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Priority: u=4
Te: trailers
Connection: keep-alive


```

Fetch API: Cross-Origin Request

Requesting Origin: https://www.google.com
Target Origin: https://example.com

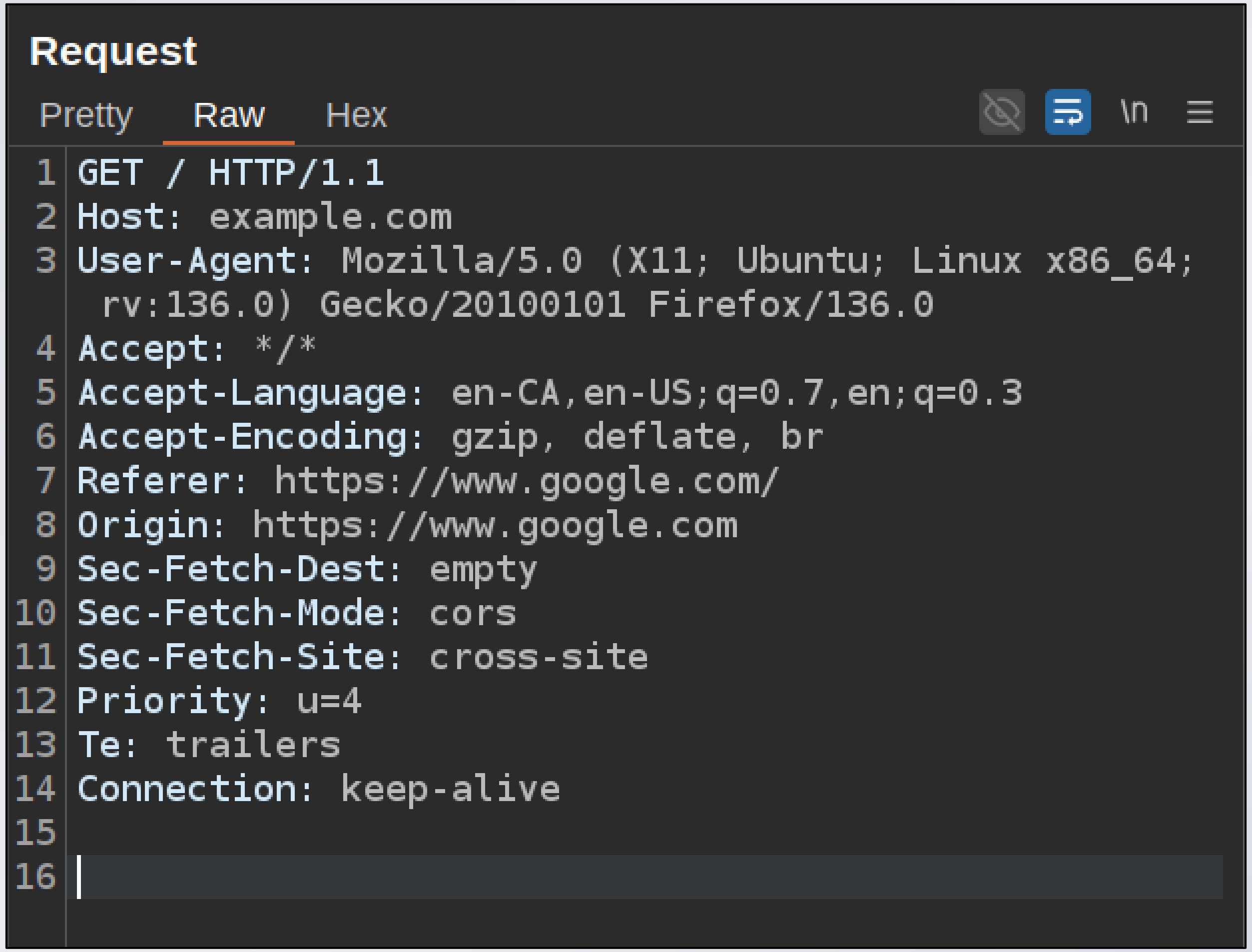


```
fetch("https://example.com")
  .then(response => response.text())
  .then(data => console.log("Success! Data length:", data.length));
```



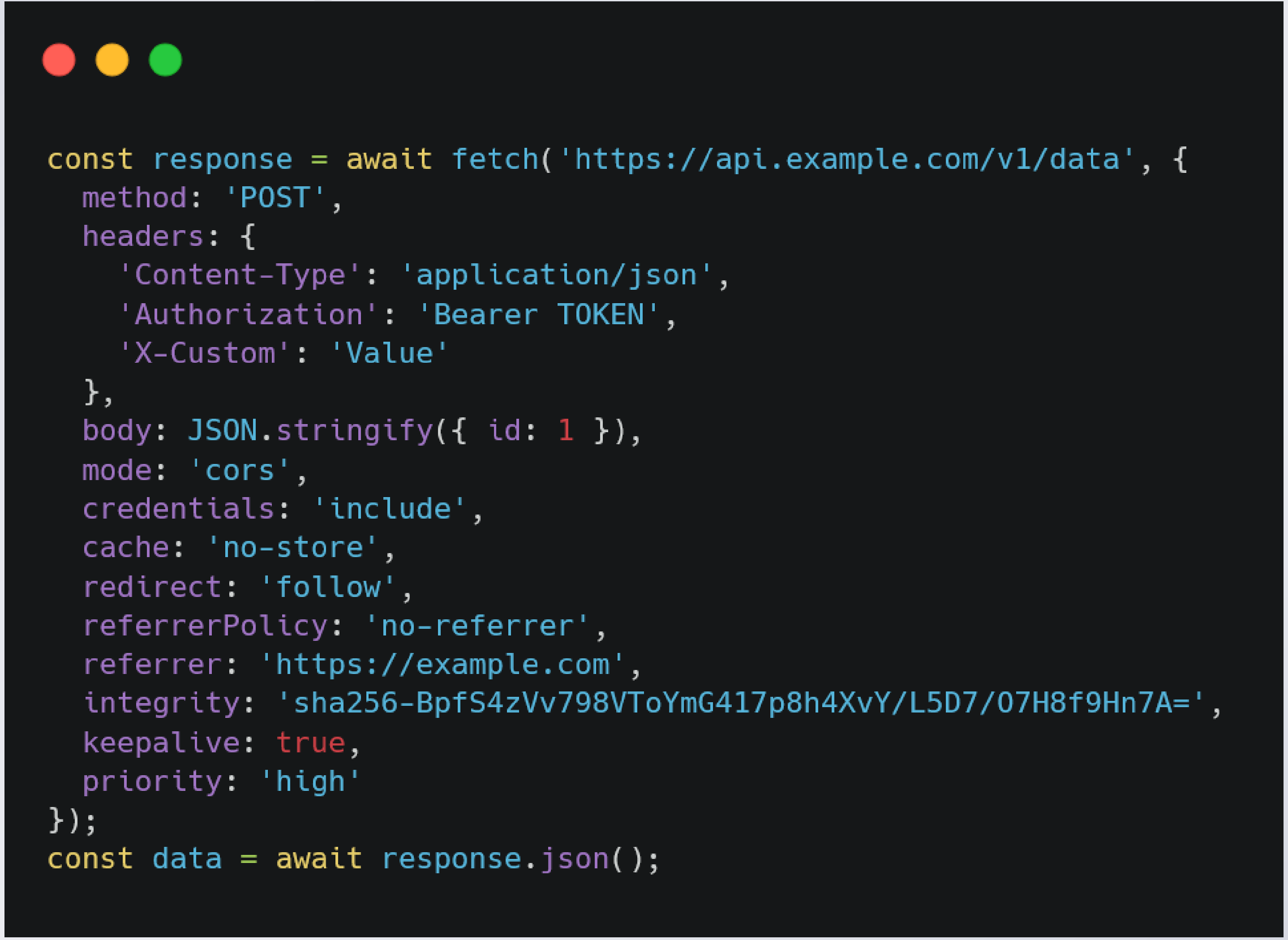
```
» fetch("https://example.com")
    .then(response => response.text())
    .then(data => console.log("Success! Data length:", data.length));
← ▸ Promise { <state>: "pending" }
❗ Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://example.com/. (Reason: CORS header
  'Access-Control-Allow-Origin' missing). Status code: 200. [Learn More]
❗ Uncaught (in promise) TypeError: NetworkError when attempting to fetch resource.
```

Fetch API: Cross-Origin Request



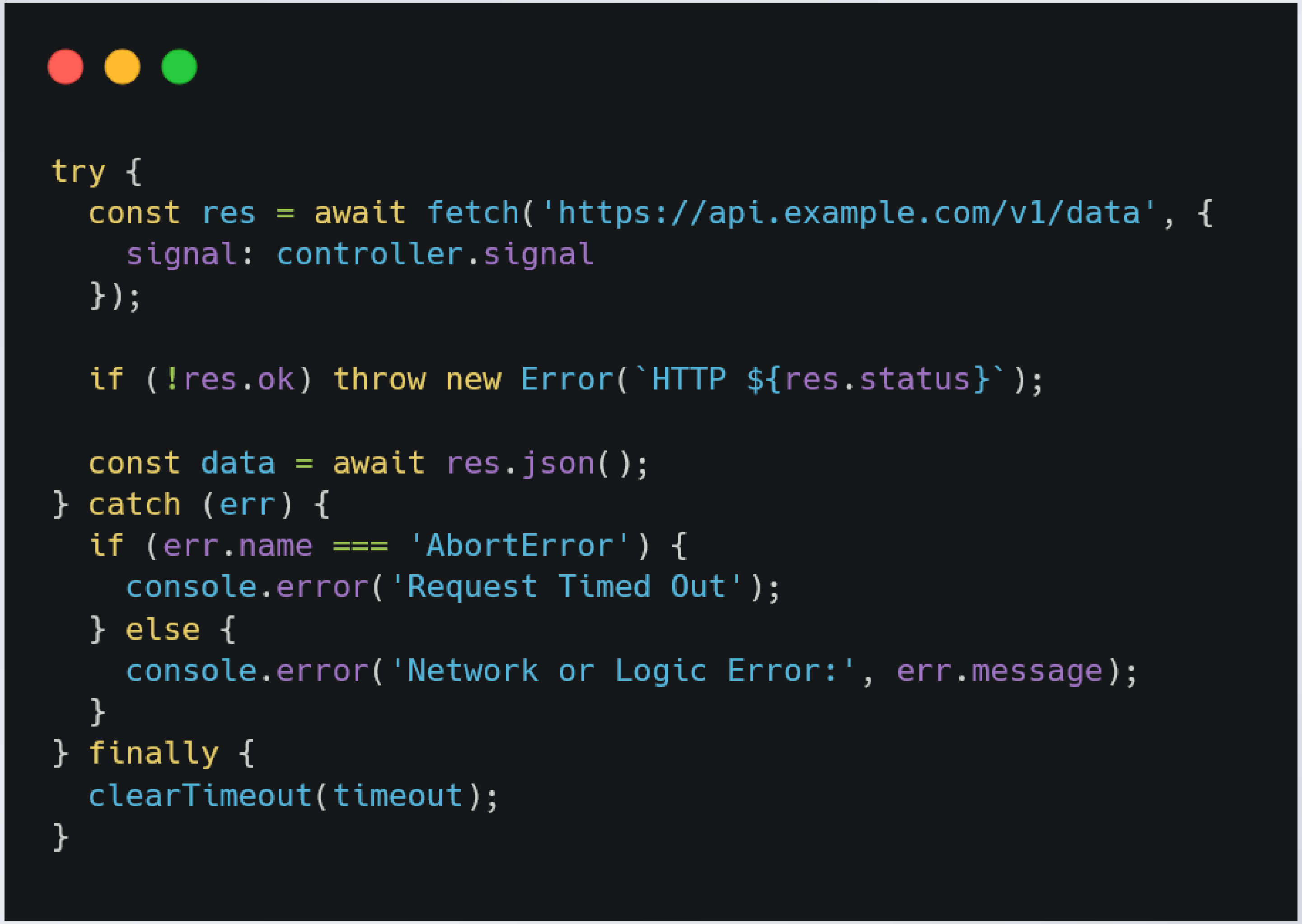
```
GET / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64;
 rv:136.0) Gecko/20100101 Firefox/136.0
Accept: */*
Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate, br
Referer: https://www.google.com/
Origin: https://www.google.com
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
Priority: u=4
Te: trailers
Connection: keep-alive


```

Fetch API: Complex!



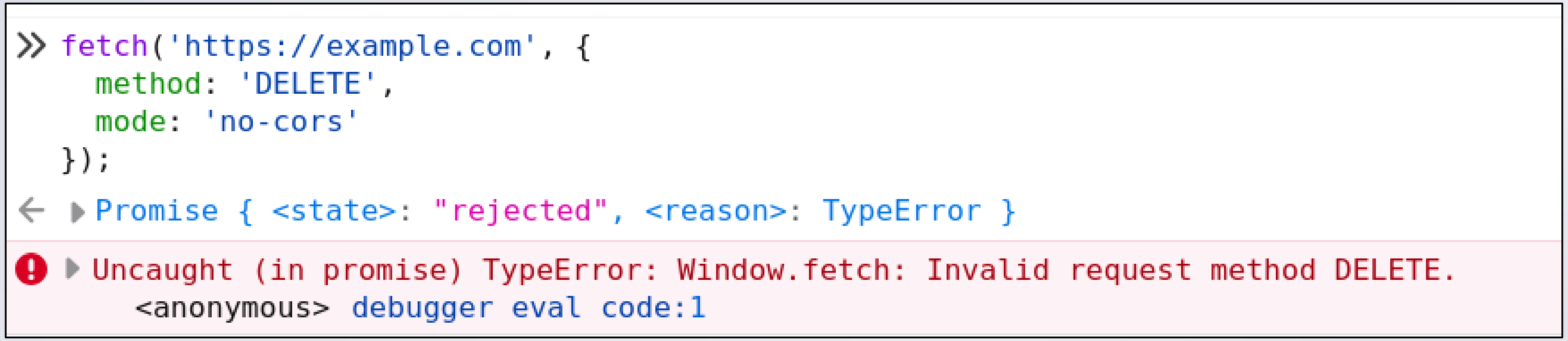
```
const response = await fetch('https://api.example.com/v1/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer TOKEN',
    'X-Custom': 'Value'
  },
  body: JSON.stringify({ id: 1 }),
  mode: 'cors',
  credentials: 'include',
  cache: 'no-store',
  redirect: 'follow',
  referrerPolicy: 'no-referrer',
  referrer: 'https://example.com',
  integrity: 'sha256-BpfS4zVv798VToYmG417p8h4XvY/L5D7/O7H8f9Hn7A=',
  keepalive: true,
  priority: 'high'
});
const data = await response.json();
```

Fetch API: Catching Errors



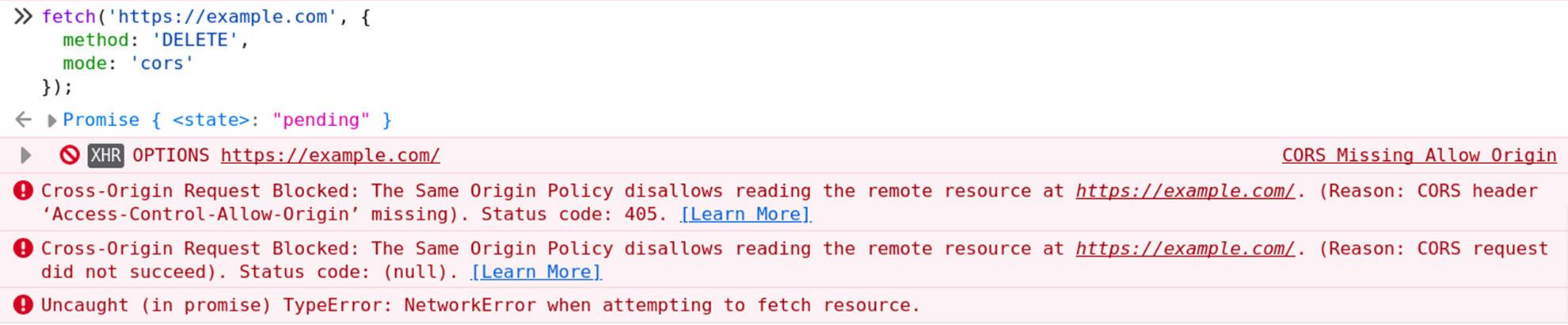
```
try {
  const res = await fetch('https://api.example.com/v1/data', {
    signal: controller.signal
  });

  if (!res.ok) throw new Error(`HTTP ${res.status}`);

  const data = await res.json();
} catch (err) {
  if (err.name === 'AbortError') {
    console.error('Request Timed Out');
  } else {
    console.error('Network or Logic Error:', err.message);
  }
} finally {
  clearTimeout(timeout);
}
```

Fetch API: Restrictions Without CORS

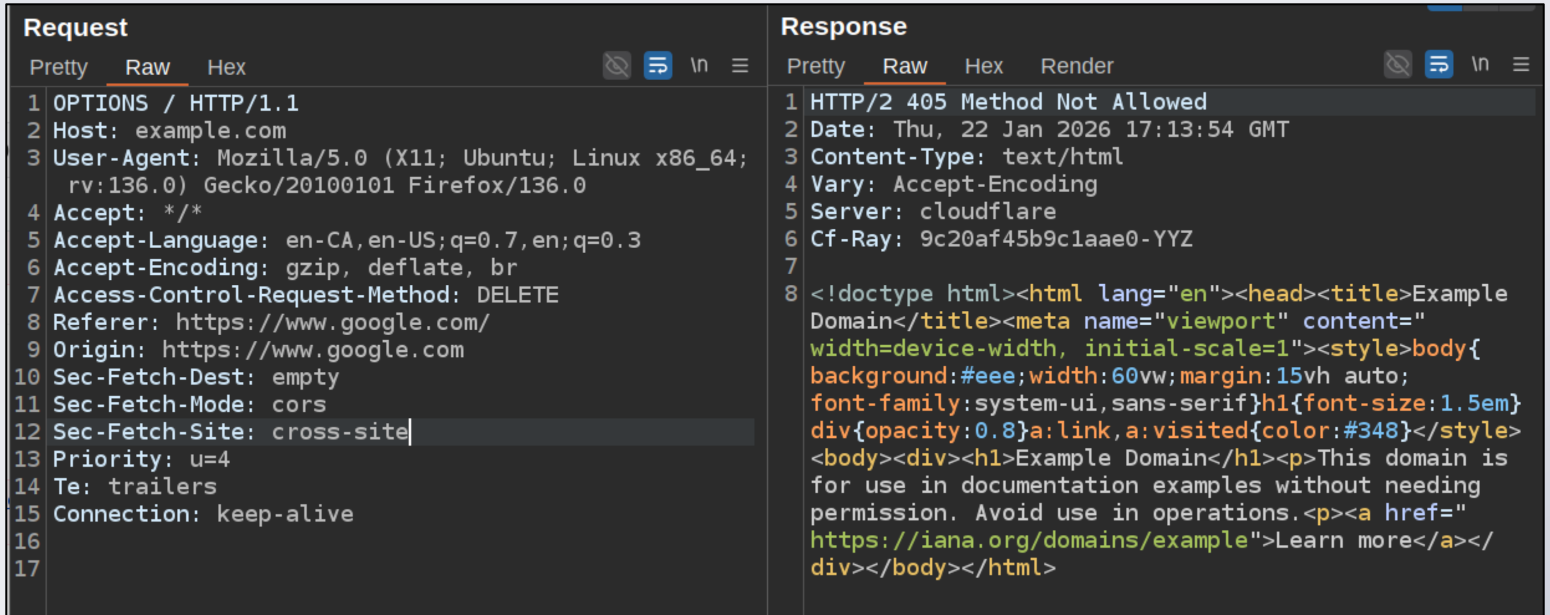


```
» fetch('https://example.com', {
    method: 'DELETE',
    mode: 'no-cors'
  });
← ▶ Promise { <state>: "rejected", <reason>: TypeError }
❗ ▶ Uncaught (in promise) TypeError: Window.fetch: Invalid request method DELETE.
      <anonymous> debugger eval code:1
```

Fetch API: Restrictions Without CORS



```
>> fetch('https://example.com', {
     method: 'DELETE',
     mode: 'cors'
   });
<- ▶ Promise { <state>: "pending" }
▶ 🚫 XHR OPTIONS https://example.com/                                     CORS Missing Allow Origin
❗ Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://example.com/. (Reason: CORS header
   'Access-Control-Allow-Origin' missing). Status code: 405. [Learn More]
❗ Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://example.com/. (Reason: CORS request
   did not succeed). Status code: (null). [Learn More]
❗ Uncaught (in promise) TypeError: NetworkError when attempting to fetch resource.
```



```
OPTIONS / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64;
  rv:136.0) Gecko/20100101 Firefox/136.0
Accept: */*
Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate, br
Access-Control-Request-Method: DELETE
Referer: https://www.google.com/
Origin: https://www.google.com
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
Priority: u=4
Te: trailers
Connection: keep-alive


```

```
HTTP/2 405 Method Not Allowed
Date: Thu, 22 Jan 2026 17:13:54 GMT
Content-Type: text/html
Vary: Accept-Encoding
Server: cloudflare
Cf-Ray: 9c20af45b9c1aae0-YYZ

<!doctype html><html lang="en"><head><title>Example
Domain</title><meta name="viewport" content="
width=device-width, initial-scale=1"><style>body{
background:#eee;width:60vw;margin:15vh auto;
font-family:system-ui,sans-serif}h1{font-size:1.5em}
div{opacity:0.8}a:link,a:visited{color:#348}</style>
<body><div><h1>Example Domain</h1><p>This domain is
for use in documentation examples without needing
permission. Avoid use in operations.<p><a href="
https://iana.org/domains/example">Learn more</a></
div></body></html>
```

Fetch Metadata (HTTP Headers)

- Distinct spec: https://w3c.github.io/webappsec-fetch-metadata/
- Browser Enriches HTTP requests with contextual information
- Included in requests as unique HTTP headers
- Includes:
  - Sec-Fetch-Site: indicates relationship between source and destination
  - Sec-Fetch-Mode: indicates the *mode* of a request
  - *Sec-Fetch-User: set to “?1” when request is trigger by “user activation”
  - Sec-Fetch-Dest: context of the request in originating app

* Not used by Safari as of January 2026: https://caniuse.com/?search=sec-fetch

Fetch Metadata: Sec-Fetch-Site

Value	Meaning
cross-site	This request is cross-origin (implies cross-origin)
same-origin	This request is same-origin (implies same-site)
same-site	This request is same-site (implies cross-origin)
none	A user initiated this operation directly (such as opening a URL)

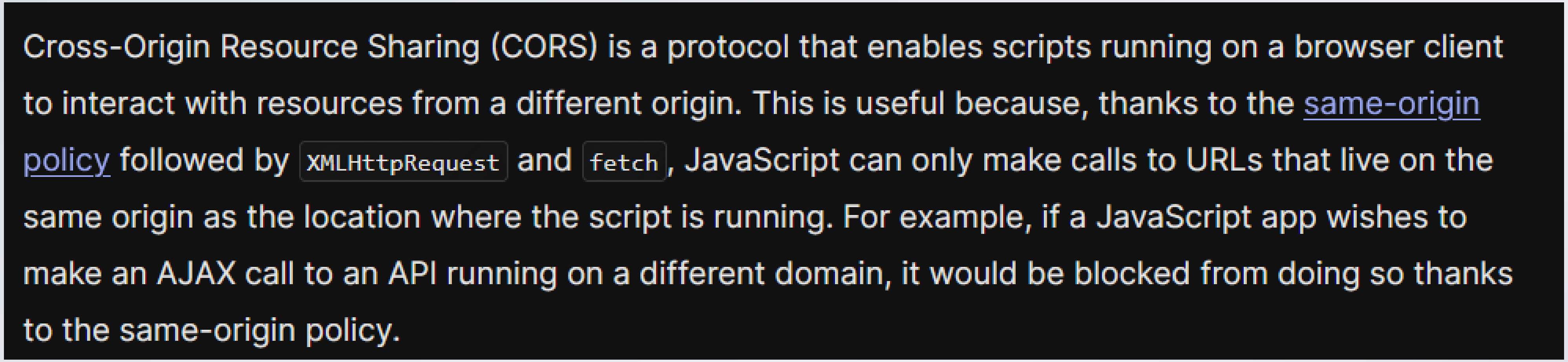
Fetch Metadata: Sec-Fetch-Mode

Value	Meaning
cors	Request is a CORS request
no-cors	Request is a no-cors request
navigate	Request initiated by navigation
same-origin	Request is made from same-origin
websocket	Request is initiating WebSocket connection

Fetch Metadata: Sec-Fetch-Dest

Value	Meaning (simplified)
iframe	Request originating from HTML <iframe>
script	Request originating from HTML <script>
style	Request originating from HTML <link rel=stylesheet>
video	Request originating from HTML <video>
…	Many more values…

No One Understands CORS!

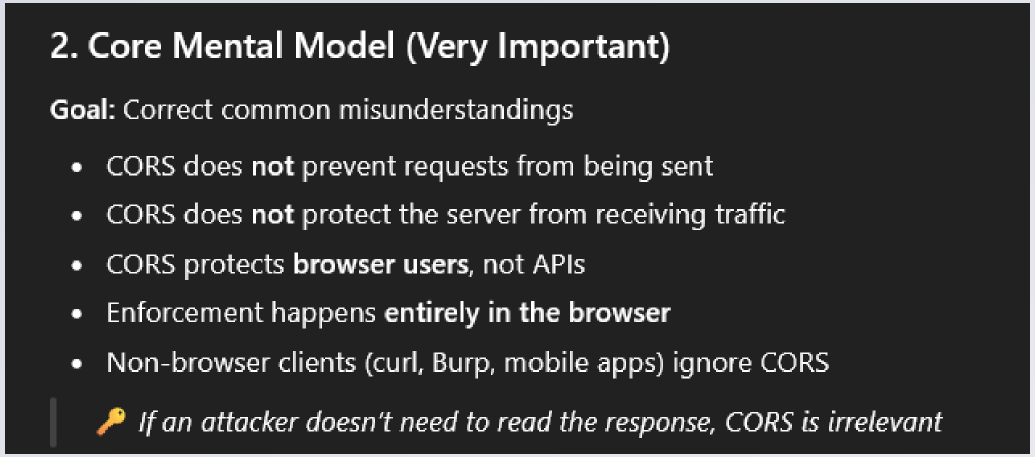


Cross-Origin Resource Sharing (CORS) is a protocol that enables scripts running on a browser client to interact with resources from a different origin. This is useful because, thanks to the same-origin policy followed by `XMLHttpRequest` and `fetch`, JavaScript can only make calls to URLs that live on the same origin as the location where the script is running. For example, if a JavaScript app wishes to make an AJAX call to an API running on a different domain, it would be blocked from doing so thanks to the same-origin policy.

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

auth0

No One Understands CORS!

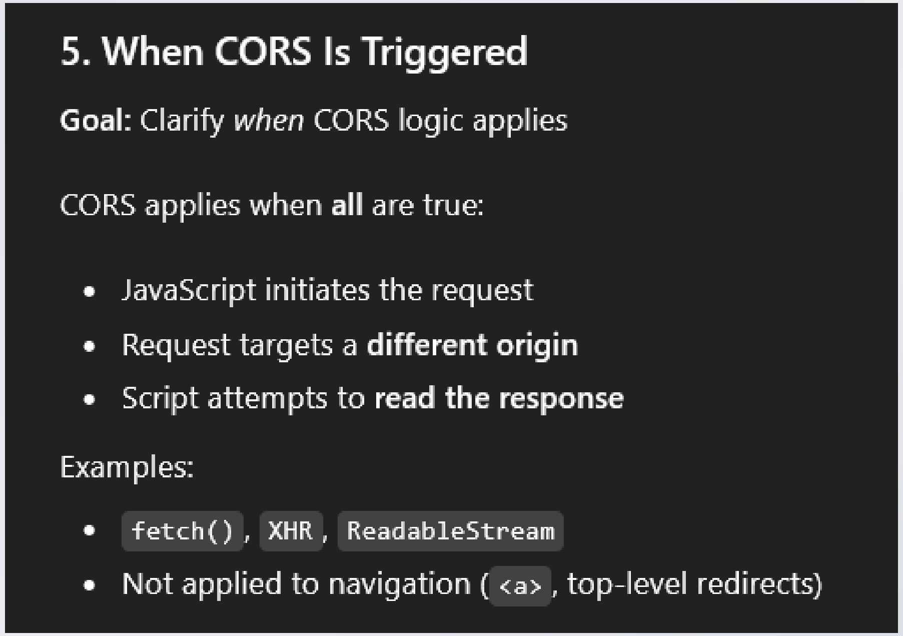


2. Core Mental Model (Very Important)

Goal: Correct common misunderstandings

- CORS does **not** prevent requests from being sent
- CORS does **not** protect the server from receiving traffic
- CORS protects **browser users**, not APIs
- Enforcement happens **entirely in the browser**
- Non-browser clients (curl, Burp, mobile apps) ignore CORS

> 🔑 *If an attacker doesn't need to read the response, CORS is irrelevant*



5. When CORS Is Triggered

Goal: Clarify *when* CORS logic applies

CORS applies when **all** are true:

- JavaScript initiates the request
- Request targets a **different origin**
- Script attempts to **read the response**

Examples:

- `fetch()`, `XHR`, `ReadableStream`
- Not applied to navigation (`<a>`, top-level redirects)

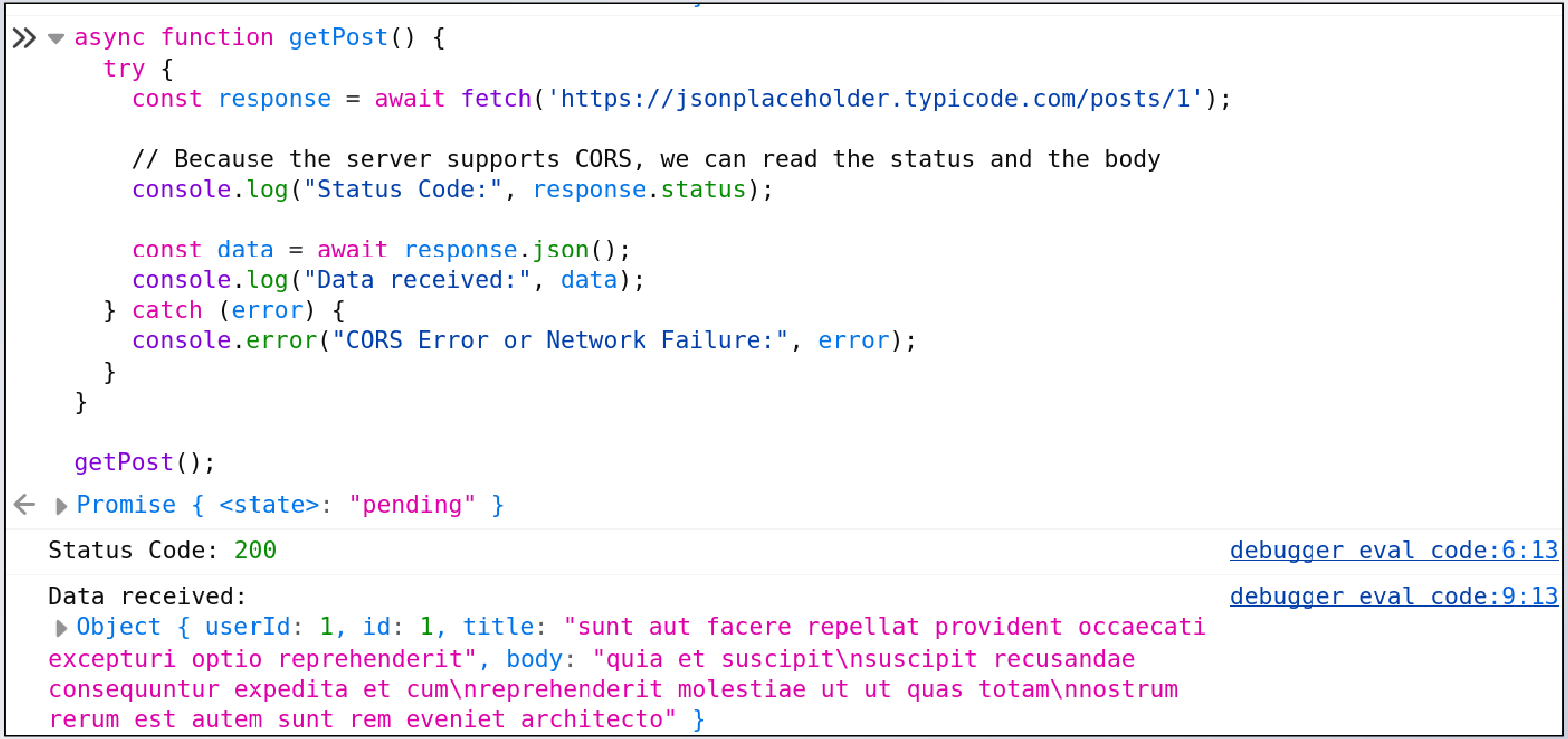
ChatGPT

Cross-Origin Resource Sharing (CORS)

- Can ONLY relax SOP and related security mechanisms
- Permits cross-origin access to read potentially sensitive data
- Opt-in by the server via HTTP headers (Access-Control-Allow-*)
- Enforced by the browser largely through Fetch API
- Often considered to *only* permit cross-origin *reads* but can also permit additional HTTP requests to be sent by the browser
- A modern replacement for old hacks like JSONP

CORS: Simple Example

Requesting Origin: www.google.com
Target Origin: jsonplaceholder.typicode.com

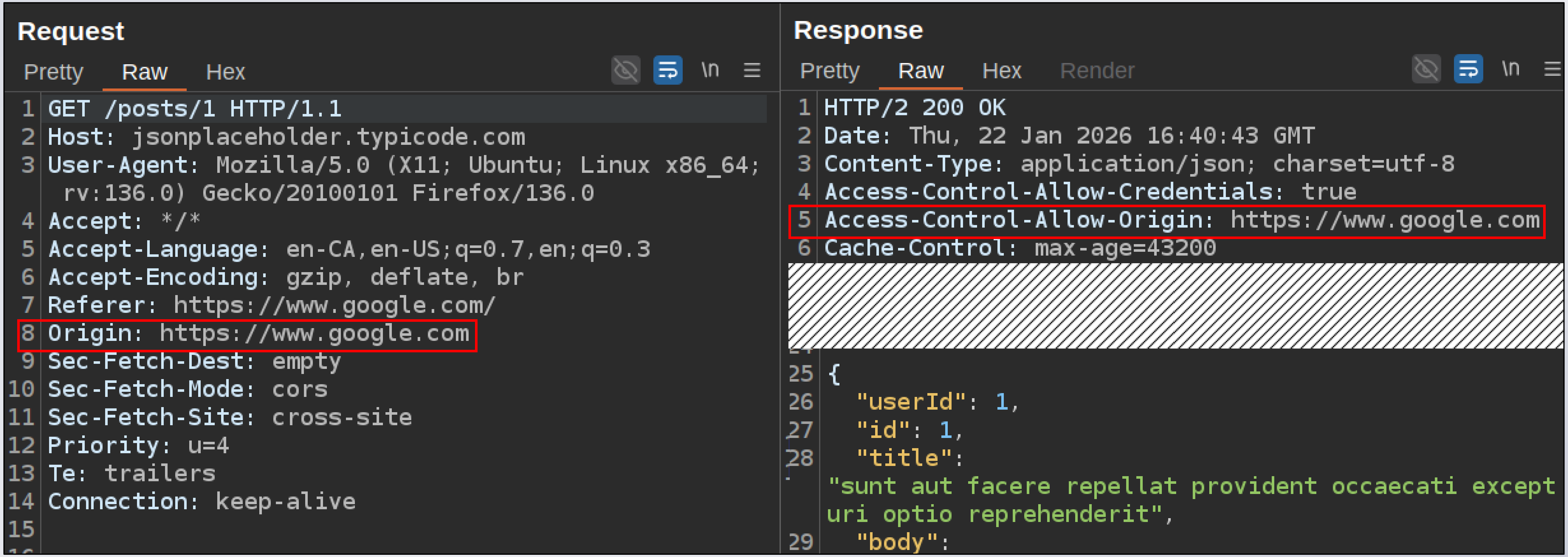


```
async function getPost() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');

    // Because the server supports CORS, we can read the status and the body
    console.log("Status Code:", response.status);

    const data = await response.json();
    console.log("Data received:", data);
  } catch (error) {
    console.error("CORS Error or Network Failure:", error);
  }
}

getPost();
```

```
← Promise { <state>: "pending" }

Status Code: 200                                        debugger eval code:6:13

Data received:                                          debugger eval code:9:13
Object { userId: 1, id: 1, title: "sunt aut facere repellat provident occaecati
excepturi optio reprehenderit", body: "quia et suscipit\nsuscipit recusandae
consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum
rerum est autem sunt rem eveniet architecto" }
```

CORS: Simple Example

Requesting Origin: https://www.google.com
Target Origin: https://jsonplaceholder.typicode.com

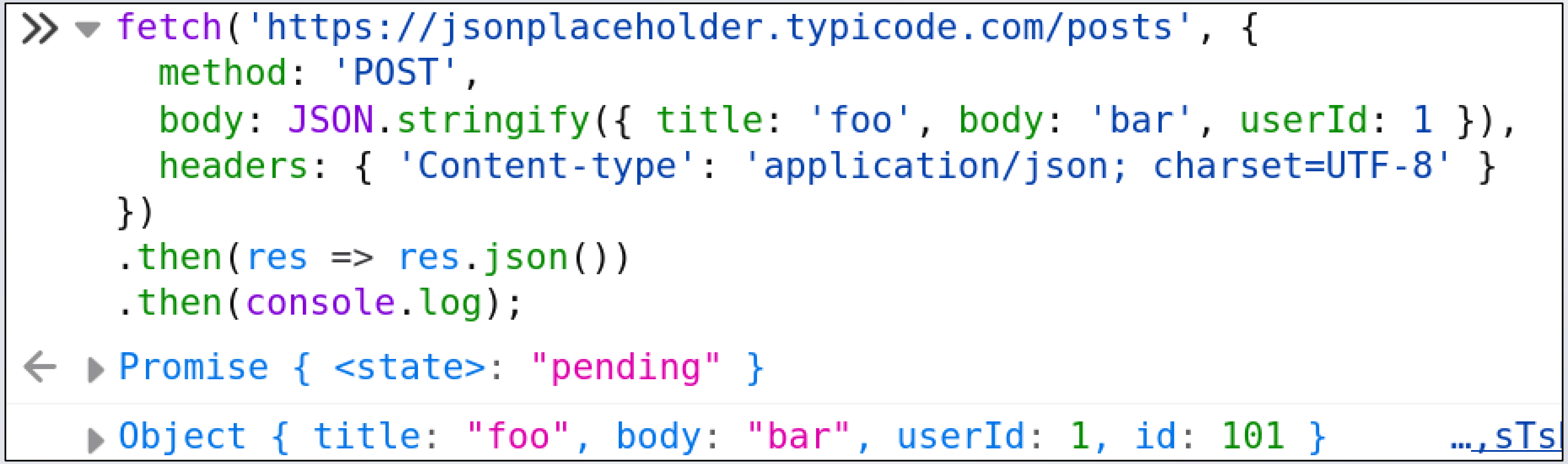


```
GET /posts/1 HTTP/1.1
Host: jsonplaceholder.typicode.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64;
 rv:136.0) Gecko/20100101 Firefox/136.0
Accept: */*
Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate, br
Referer: https://www.google.com/
Origin: https://www.google.com
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
Priority: u=4
Te: trailers
Connection: keep-alive

```

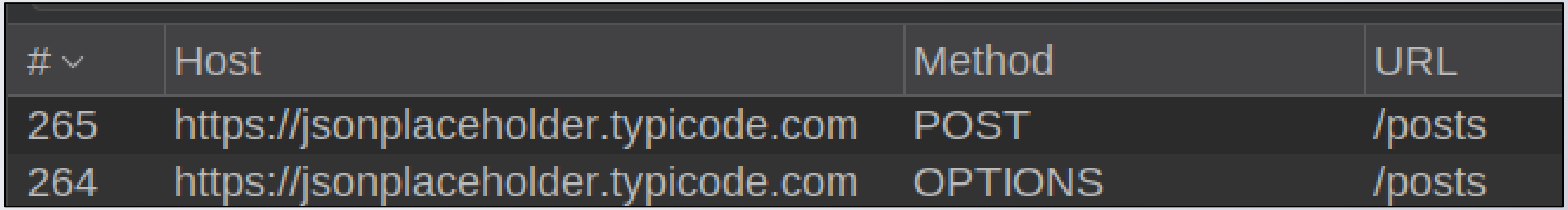
```
HTTP/2 200 OK
Date: Thu, 22 Jan 2026 16:40:43 GMT
Content-Type: application/json; charset=utf-8
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: https://www.google.com
Cache-Control: max-age=43200
[redacted]
{
  "userId": 1,
  "id": 1,
  "title":
"sunt aut facere repellat provident occaecati except
uri optio reprehenderit",
  "body":
```

CORS: Preflight Example

Requesting Origin: https://www.google.com
Target Origin: https://jsonplaceholder.typicode.com



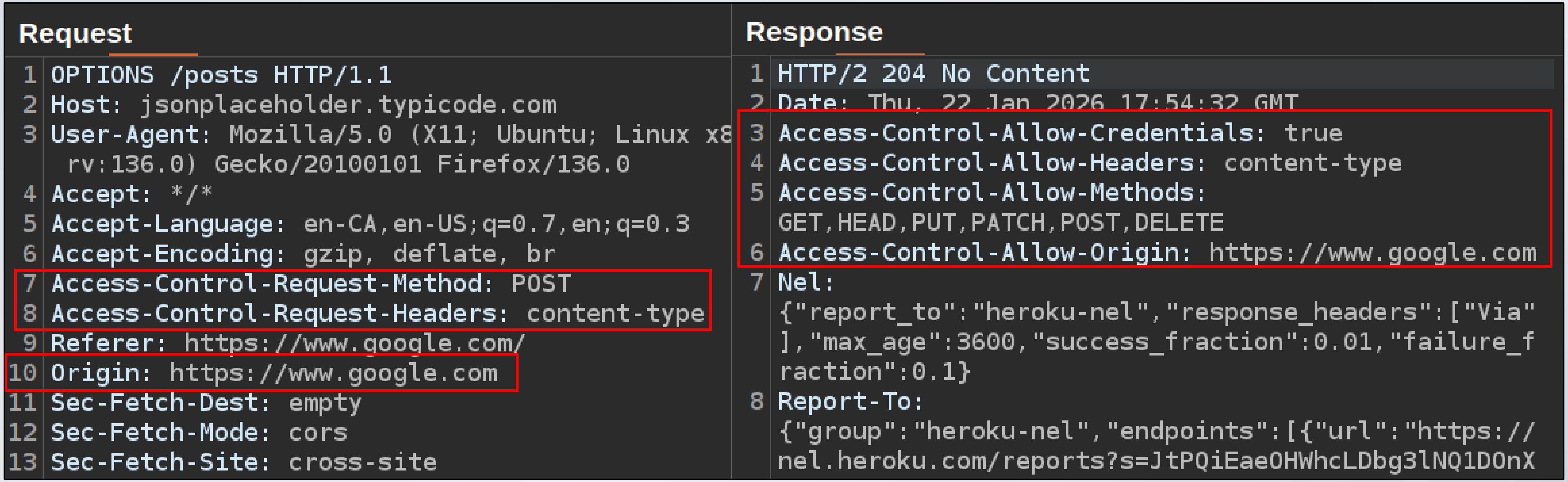
```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  body: JSON.stringify({ title: 'foo', body: 'bar', userId: 1 }),
  headers: { 'Content-type': 'application/json; charset=UTF-8' }
})
.then(res => res.json())
.then(console.log);
Promise { <state>: "pending" }
Object { title: "foo", body: "bar", userId: 1, id: 101 }        …, sTs
```



#	Host	Method	URL
265	https://jsonplaceholder.typicode.com	POST	/posts
264	https://jsonplaceholder.typicode.com	OPTIONS	/posts

CORS: Preflight Example

Requesting Origin: https://www.google.com
Target Origin: https://jsonplaceholder.typicode.com



Request

```
OPTIONS /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x8
  rv:136.0) Gecko/20100101 Firefox/136.0
Accept: */*
Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate, br
Access-Control-Request-Method: POST
Access-Control-Request-Headers: content-type
Referer: https://www.google.com/
Origin: https://www.google.com
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
```

Response

```
HTTP/2 204 No Content
Date: Thu, 22 Jan 2026 17:54:32 GMT
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: content-type
Access-Control-Allow-Methods:
GET,HEAD,PUT,PATCH,POST,DELETE
Access-Control-Allow-Origin: https://www.google.com
Nel:
{"report_to":"heroku-nel","response_headers":["Via"
],"max_age":3600,"success_fraction":0.01,"failure_f
raction":0.1}
Report-To:
{"group":"heroku-nel","endpoints":[{"url":"https://
nel.heroku.com/reports?s=JtPQiEaeOHWhcLDbg3lNQ1DOnX
```

CORS: Key Client-Side Concepts

Simple vs. Non-Simple Requests

Simple requests can be sent without a Preflight request, but non-simple require preflight

Preflight

An HTTP OPTIONS method request to enumerate permissions via *Access-Control-Allow-** HTTP Response headers (sent for non-simple requests)

Mode: ‘cors’ vs ‘no-cors’

Explicitly or implicitly declares whether the client has an *intent* to *read* data (as a request property), ultimately determining whether CORS will be utilized (if not, response is *opaque*)

Opaque Response

Browser withholds response data (empty/null *body* and *status* of 0)

(response.type === “opaque”)

Simple Requests (No Preflight)

Permitted HTTP Request Methods (all others blocked):

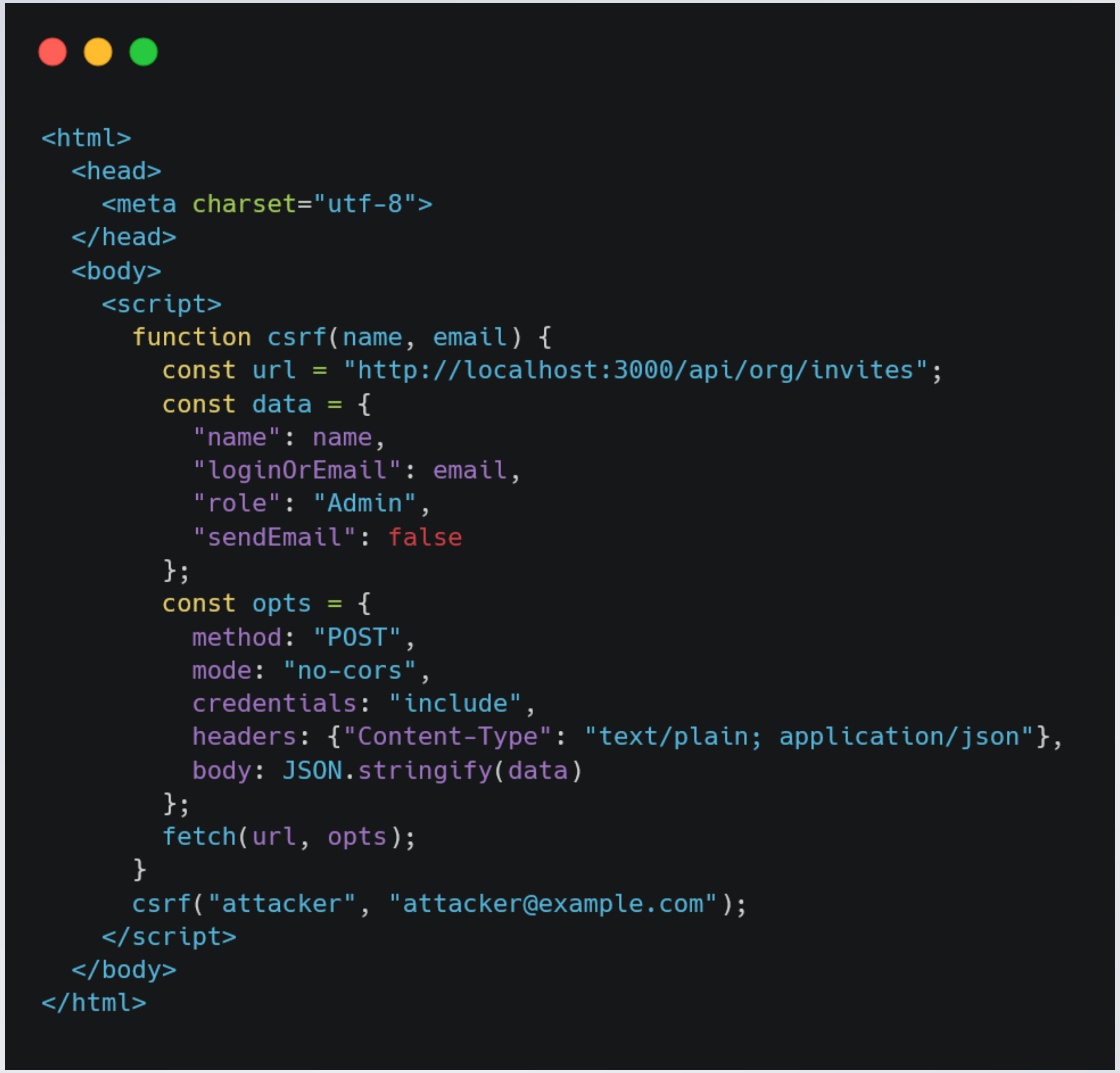
- **GET**
- **POST**
- **Head**
- **OPTIONS (as a side effect)**

Permitted HTTP Request Headers (all others blocked):

- **Accept**
- **Accept-Language**
- **Content-Language**
- **Content-Type, limited to:**
  - **application/x-www-form-urlencoded**
  - **multipart/form-data**
  - **text/plain**

Note: some exceptions to the above

Exception: Content-Type Smuggling



```
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <script>
      function csrf(name, email) {
        const url = "http://localhost:3000/api/org/invites";
        const data = {
          "name": name,
          "loginOrEmail": email,
          "role": "Admin",
          "sendEmail": false
        };
        const opts = {
          method: "POST",
          mode: "no-cors",
          credentials: "include",
          headers: {"Content-Type": "text/plain; application/json"},
          body: JSON.stringify(data)
        };
        fetch(url, opts);
      }
      csrf("attacker", "attacker@example.com");
    </script>
  </body>
</html>
```

https://jub0bs.com/posts/2022-02-08-cve-2022-21703-writeup/#bypassing-content-type-validation-and-avoiding-cors-preflight

Simple Requests (cors vs. no-cors mode)

cors mode: fetch('https://api.example.com/data', { mode: ‘cors’ })

no-cors mode: fetch('https://api.example.com/data', { mode: ‘no-cors’ })

HTTP Header	Mode: ‘cors’	Mode: ‘no-cors’
Origin	https://origin-app.com	(typically omitted for GET)*
Sec-Fetch-Mode	cors	no-cors
Sec-Fetch-Site	cross-site	cross-site

* Note: Origin header should always be sent for cross-origin HTTP POST requests due to its historical use in protecting against CSRF attacks (POST = unsafe method)!

CORS and Embedded (HTML) Resources

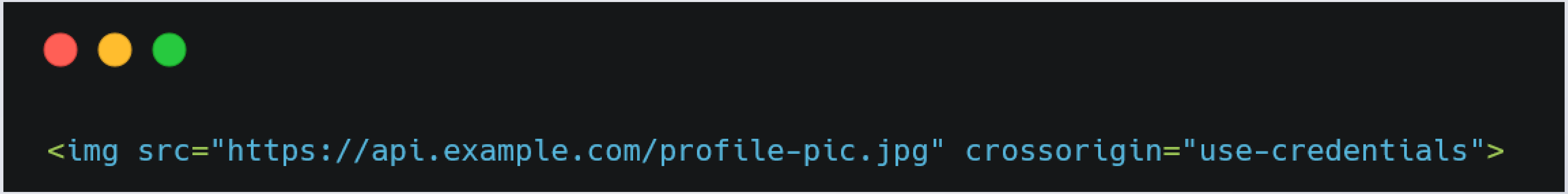
- By default, HTML tags (‘<img>’) are effectively ‘*no-cors*'
- The <form> tag is the only HTML element that can send POST reqs
- Elements embedding resources from another app are *tainted* and cannot be read (canvas)
- HTML elements can opt-in to CORS using *crossorigin* attribute with values:
  - *anonymous*: CORS is used and credentials are only included when *same-origin*
  - *use-credentials*: CORS is used with credentials
- EXCEPTION! *@font-face* (within CSS) requires CORS (additional exceptions exist)
- Also: *Subresource Integrity* requires CORS! (Needed to read raw bytes)

Use Credentials

Browser will include credentials for the target domain, such as:

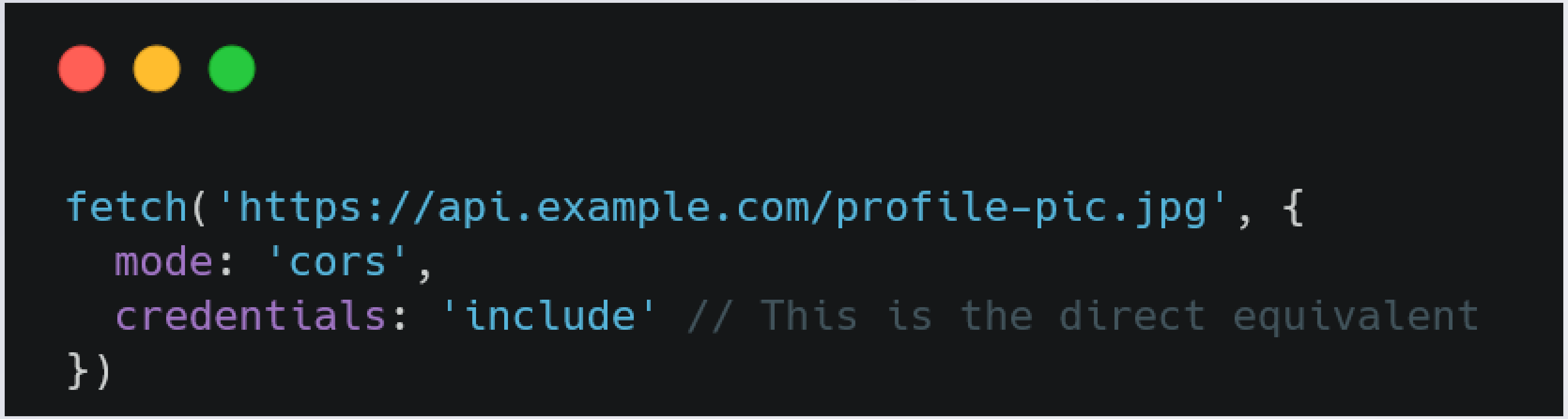
- HTTP Cookies
- HTTP Authentication (BASIC, DIGEST, NTLM)
- TLS certificates

Does NOT include Bearer tokens or other custom mechanisms!
Many modern apps are therefore inherently secure against credentialed CORS.



```html
<img src="https://api.example.com/profile-pic.jpg" crossorigin="use-credentials">
```

Fetch API: *credentials* Property



```
fetch('https://api.example.com/profile-pic.jpg', {
  mode: 'cors',
  credentials: 'include' // This is the direct equivalent
})
```

credentials Value	Behavior
omit	Never send credentials in request or include in response
same-origin (default)	Send and include credentials in same-origin requests
include	Always include credentials

Credentialed Request Requirements

- Credentials are automatically included for same-origin requests
- Credential inclusion also depends on credential configuration and browser rules
- Setting Fetch *credentials* to *include* will send them cross-origin
  - Exception: cookies are set with *SameSite Strict* or *Lax*
- Simple requests include credentials without preflight
- Non-simple requests first send non-credentialed preflight (OPTIONS)
  - If credentialed permission is not granted, no subsequent request is sent!
- The target application can return CORS headers to define permissions
- So, how is permission granted to the client?

Essential CORS HTTP Response Headers

Returned by the server to grant CORS permissions.

Access-Control-Allow-Origin

- Controls which source origins can access (and read) resource
- Values:
    - <origin>
    - *
    - null

Access-Control-Allow-Credentials

- Tells browser whether credentialed requests are actually permitted
- Values:
    - true
    - (otherwise omit header!)

ACAO Behavior When Using <Origin>

ACAO Setting	ACAO Value (Example)	Behvaior
Multiple ACAO Headers	Access-Control-Allow-Origin: https://www.site1.com Access-Control-Allow-Origin: https://www.site2.com	Request will fail (violation)
Missing protocol	Access-Control-Allow-Origin: www.site.com	Request will fail (violation)
Insecure protocol	Access-Control-Allow-Origin: http://www.site.com	Permitted if the requesting site is using http:// (otherwise, the origin is different)
Comma-separated Origins	Access-Control-Allow-Origin: https://www.site2.com, https://www.site1.com	Request will fail (violation)
Subdomain wildcards	Access-Control-Allow-Origin: https://*.site.com	Request will fail (violation)

DBG App Test Finding

Access-Control-Allow-Origin Header Returned Insecure Origin

Allow-Control-Allow-Origin: *

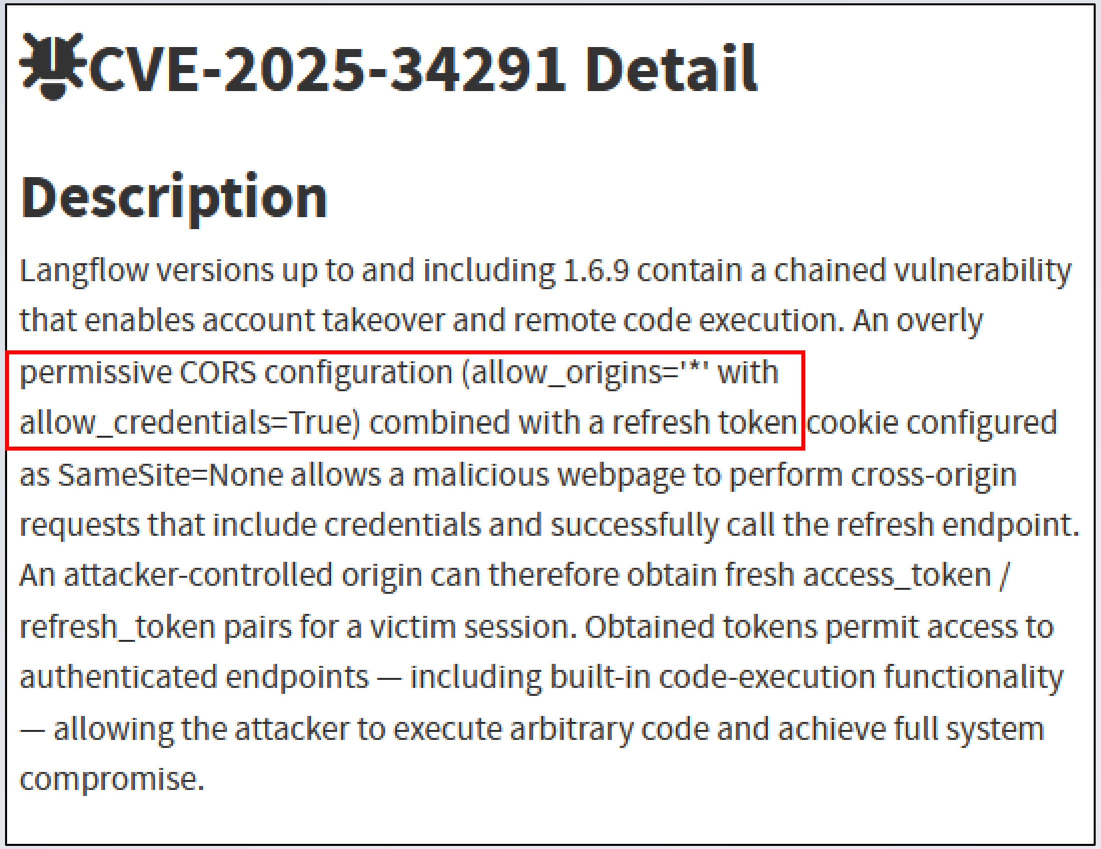
Known as the *wildcard* configuration.

Permits reads from ANY Origin, BUT:

When the wildcard configuration is used, Access-Control-Allow-Credentials CANNOT permit credentialed requests (browser-blocked)

DBG App Test Finding
Wildcard Cross-Origin Resource Sharing Policy

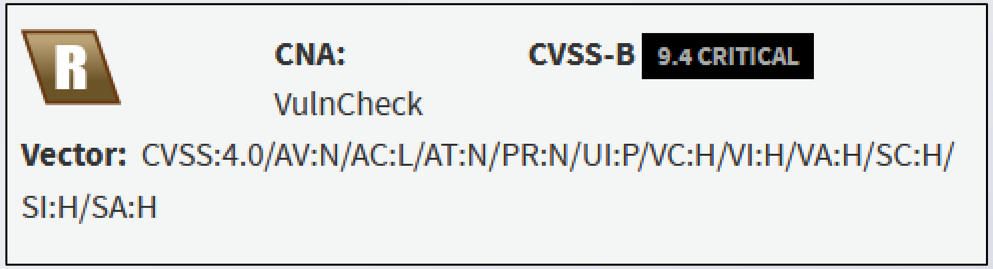
Allow-Control-Allow-Origin: *



CVE-2025-34291 Detail

Description

Langflow versions up to and including 1.6.9 contain a chained vulnerability that enables account takeover and remote code execution. An overly permissive CORS configuration (allow_origins='*' with allow_credentials=True) combined with a refresh token cookie configured as SameSite=None allows a malicious webpage to perform cross-origin requests that include credentials and successfully call the refresh endpoint. An attacker-controlled origin can therefore obtain fresh access_token / refresh_token pairs for a victim session. Obtained tokens permit access to authenticated endpoints — including built-in code-execution functionality — allowing the attacker to execute arbitrary code and achieve full system compromise.



CNA: VulnCheck

CVSS-B 9.4 CRITICAL

Vector: CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:H/VI:H/VA:H/SC:H/SI:H/SA:H

Allow-Control-Allow-Origin: null

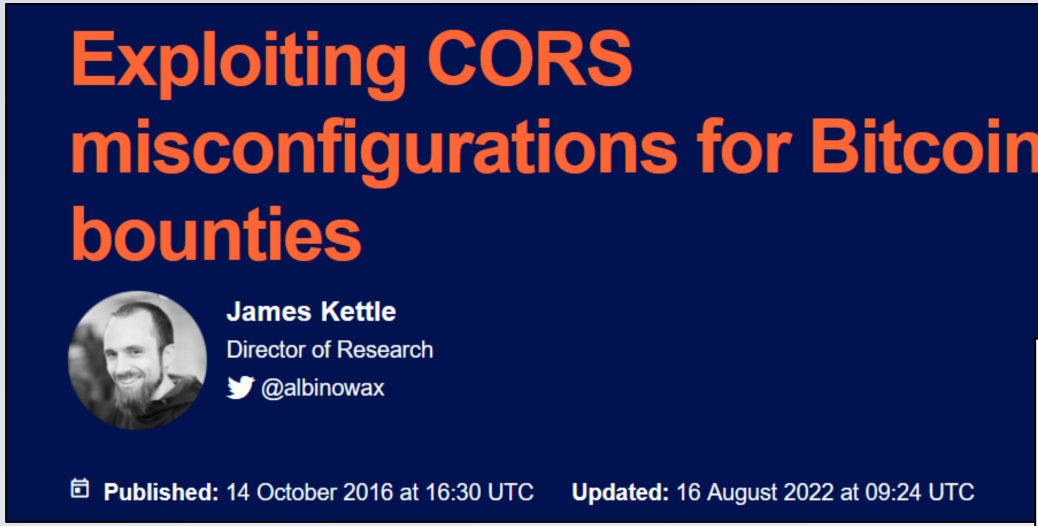
Do NOT return this.

Permits reads from “null” Origins, which may include:

- Redirects across Origins
- Requests from local files (file://)
- Sandboxed iframes
- Requests from data: URLs

DBG App Test Finding
Access-Control-Allow-Origin Header Set to "null"

Allow-Control-Allow-Origin: null

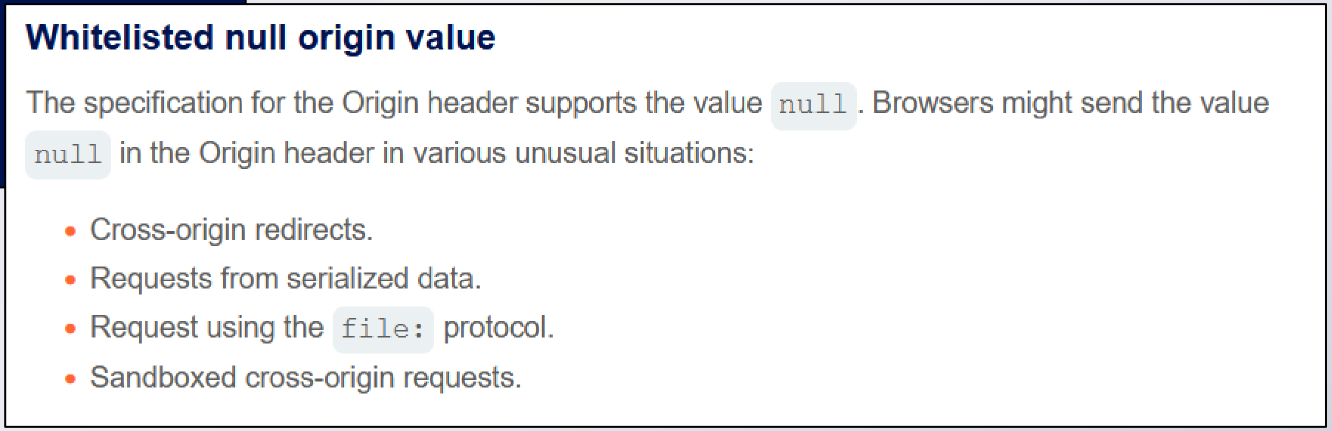


Exploiting CORS misconfigurations for Bitcoins and bounties

James Kettle
Director of Research
@albinowax

Published: 14 October 2016 at 16:30 UTC   Updated: 16 August 2022 at 09:24 UTC

https://portswigger.net/research/exploiting-cors-misconfigurations-for-bitcoins-and-bounties



Whitelisted null origin value

The specification for the Origin header supports the value `null`. Browsers might send the value `null` in the Origin header in various unusual situations:

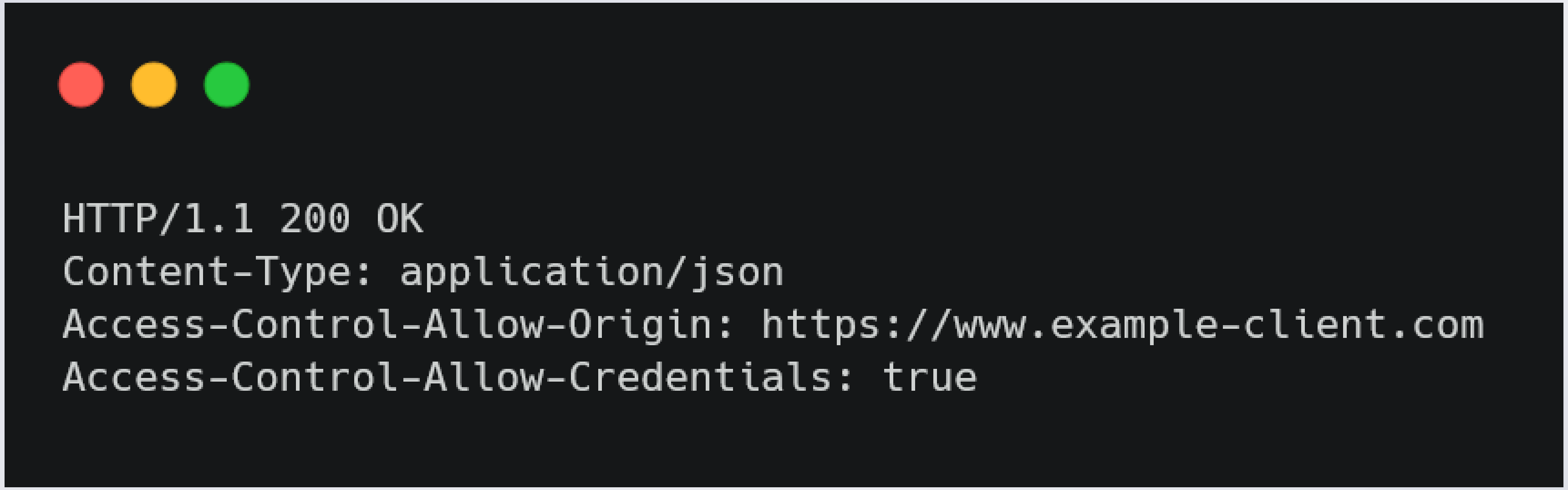
- Cross-origin redirects.
- Requests from serialized data.
- Request using the `file:` protocol.
- Sandboxed cross-origin requests.

https://portswigger.net/web-security/cors

Allow-Control-Allow-Origin/Credentials

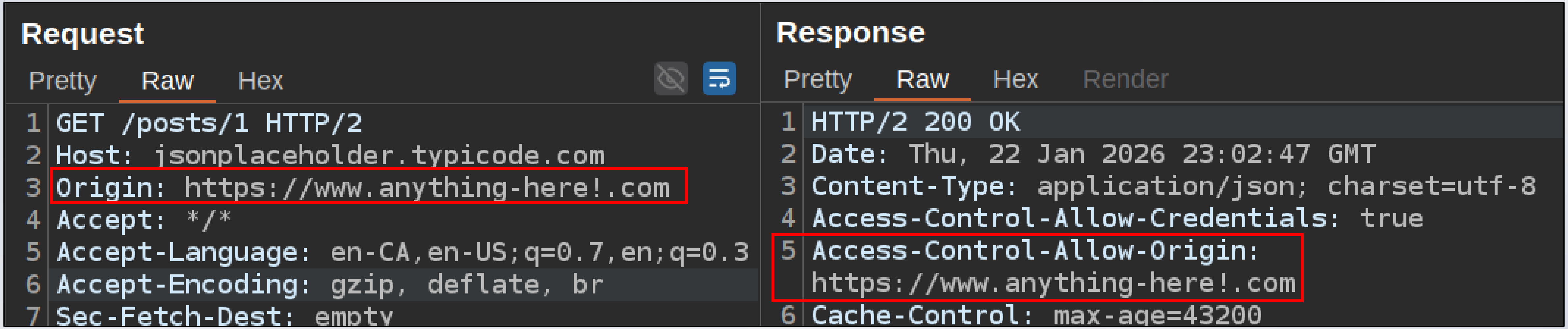
ACAC Value	ACAO Value	Requirements	Implications
(not set)	null	Don’t use this!	Some source contexts may use a *null* Origin and therefore have unintended access to a resource.
(not set)	*	Origin can be anything.	Browser can send uncredentialed (unauth) requests and read the response.
(not set)	<exact origin>	Origin must exactly match to succeed.	Browser can send uncredentialed (unauth) requests and read the response.
true	null	Don’t use this either!	Browser can send uncredentialed (unauth) requests from a *null* Origin context and read the response.
true	*	Forbidden!	Browser will block response.
true	<exact origin>	Origin must exactly match to succeed.	Browser can send credentialed (Auth) requests and read the response!!

Successful Credentialed CORS



```
HTTP/1.1 200 OK
Content-Type: application/json
Access-Control-Allow-Origin: https://www.example-client.com
Access-Control-Allow-Credentials: true
```

Danger: Reflected Origin + Credentials



```
GET /posts/1 HTTP/2
Host: jsonplaceholder.typicode.com
Origin: https://www.anything-here!.com
Accept: */*
Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate, br
Sec-Fetch-Dest: empty
```

```
HTTP/2 200 OK
Date: Thu, 22 Jan 2026 23:02:47 GMT
Content-Type: application/json; charset=utf-8
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin:
https://www.anything-here!.com
Cache-Control: max-age=43200
```

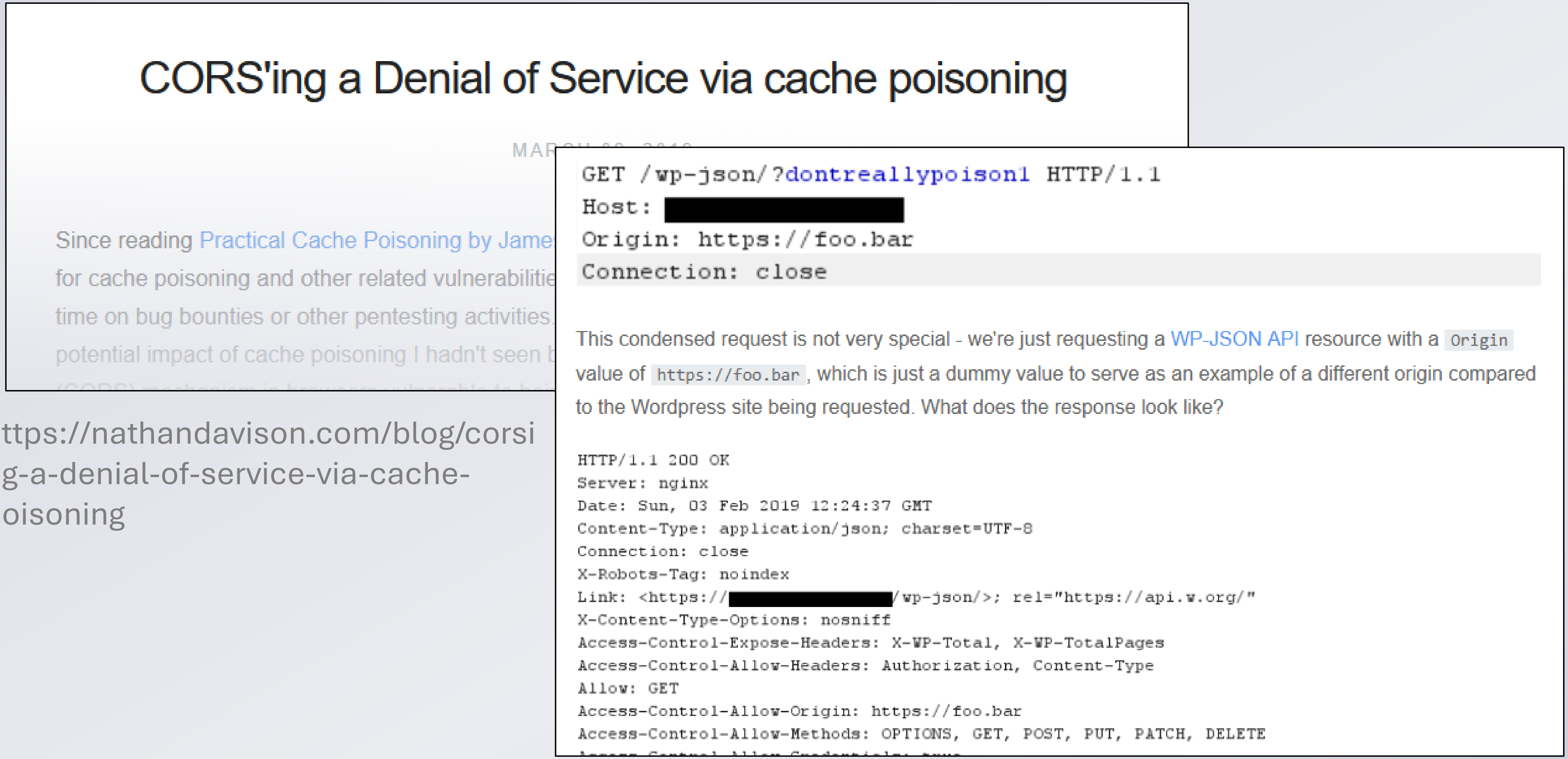
DBG App Test Finding
Permissive Cross-Origin Resource Sharing Policy

CORS and Web Caches

- **Applications supporting multiple client Origins may implement a dynamic ACAO header that uses an allow-list (validated Origin)**
- **Client-side or server-side caches may be poisoned if they do not recognize that a changing Origin header will change response content**
- **Can result in DoS and potentially XSS**
- **Mitigation: caches must recognize Origin as a cache key**
  - **This can be enforced with the HTTP Response header "*Vary: Origin*"**

DBG App Test Finding
Insufficient HTTP Header Cache Key Specification

CORS and Web Caches



CORS'ing a Denial of Service via cache poisoning

MAR

Since reading Practical Cache Poisoning by James
for cache poisoning and other related vulnerabilitie
time on bug bounties or other pentesting activities.
potential impact of cache poisoning I hadn't seen b
(CORS) mechanism in browsers vulnerable to bei

```
GET /wp-json/?dontreallypoison1 HTTP/1.1
Host: [redacted]
Origin: https://foo.bar
Connection: close
```

This condensed request is not very special - we're just requesting a WP-JSON API resource with a `Origin` value of `https://foo.bar`, which is just a dummy value to serve as an example of a different origin compared to the Wordpress site being requested. What does the response look like?

```
HTTP/1.1 200 OK
Server: nginx
Date: Sun, 03 Feb 2019 12:24:37 GMT
Content-Type: application/json; charset=UTF-8
Connection: close
X-Robots-Tag: noindex
Link: <https://[redacted]/wp-json/>; rel="https://api.w.org/"
X-Content-Type-Options: nosniff
Access-Control-Expose-Headers: X-WP-Total, X-WP-TotalPages
Access-Control-Allow-Headers: Authorization, Content-Type
Allow: GET
Access-Control-Allow-Origin: https://foo.bar
Access-Control-Allow-Methods: OPTIONS, GET, POST, PUT, PATCH, DELETE
Access-Control-Allow-Credentials: true
```

https://nathandavison.com/blog/corsing-a-denial-of-service-via-cache-poisoning

Access-Control-Request-*

These HTTP request headers use CORS to request permission to include non-safelisted HTTP methods or headers.

Access-Control-Request-Method

Request one or more HTTP methods

Access-Control-Request-Headers

Request one or more HTTP headers

Access-Control-Request-Private-Network

(Deprecated in favor of Local Network Access)

Additional Access-Control-* Headers

More permission headers servers can return.

Access-Control-Allow-Methods

Permits browser to use specified HTTP Request Methods (comma-sep)

Access-Control-Allow-Headers

Permits browser to include specified HTTP Request Headers (comma-sep)

Access-Control-Expose-Headers

Permits the client application (JavaScript) to read header values (comma-sep)

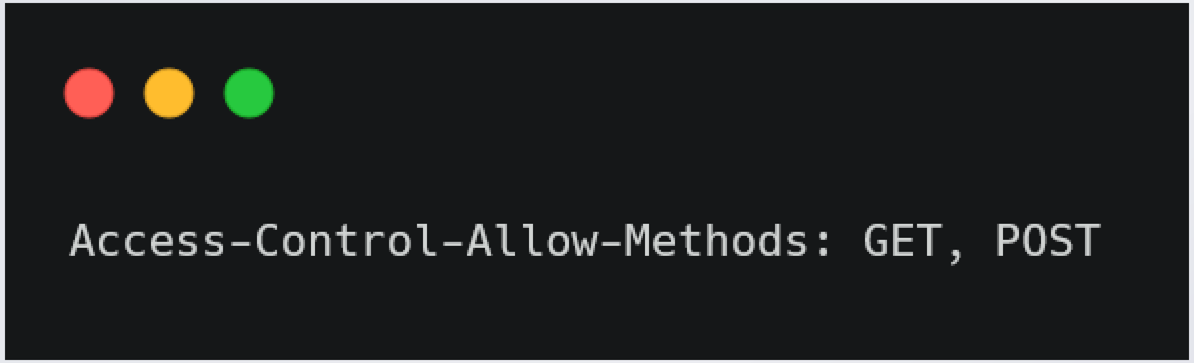
Access-Control-Allow-Private-Network

To be discussed...

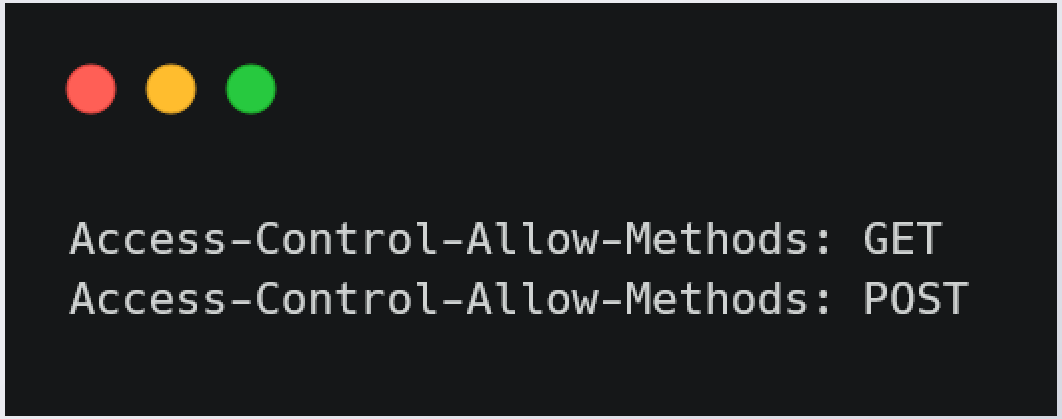
Access-Control-Max-Age...

Aside: Multiple HTTP Headers

- *Some* HTTP headers (List-valued headers) support multiple values
- These can be comma-separated, or divided across headers



```
Access-Control-Allow-Methods: GET, POST
```



```
Access-Control-Allow-Methods: GET
Access-Control-Allow-Methods: POST
```

CORS Permission Caching

Preflight Caching

Browsers cache permissions returned from an initial preflight

Access-Control-Max-Age

This HTTP response header tells the browser how long to cache permissions from a preflight (default is 5 seconds)

The above may be relevant to be aware of in very rare situations...

CORS Abuse: CSRF

To be covered in a CSRF Session…

OWASP ASVS 5.0

V3 Web Frontend Security: V3.4 Browser Security Mechanism Headers

- 3.4.2: Verify that Access-Control-Allow-Origin is fixed or allow-listed

V3 Web Frontend Security: V3.5 Browser Origin Separation

- 3.5.1: Do not rely on CORS preflight to block requests
- 3.5.2: If CORS preflight is relied on to prevent origin use of sensitive functionality, ensure functionality requires preflight

https://github.com/OWASP/ASVS/blob/master/5.0/en/0x12-V3-Web-Frontend-Security.md

Security Testing Considerations

- Ask and you might receive
  - Consider all Access-Control-Request-* headers as the web service may not return all permissions without the appropriate request!
- Insecure configurations (reflected origin + creds) still common
  - Impact depends on session handling (*SameSite*) and functionality
- Evaluate fully any server-side Origin allow-list
  - Subdomains accepted? Simple string match? Regex?
- Application/framework implementation quirks
  - Handling of special chars, whitespace, casing…