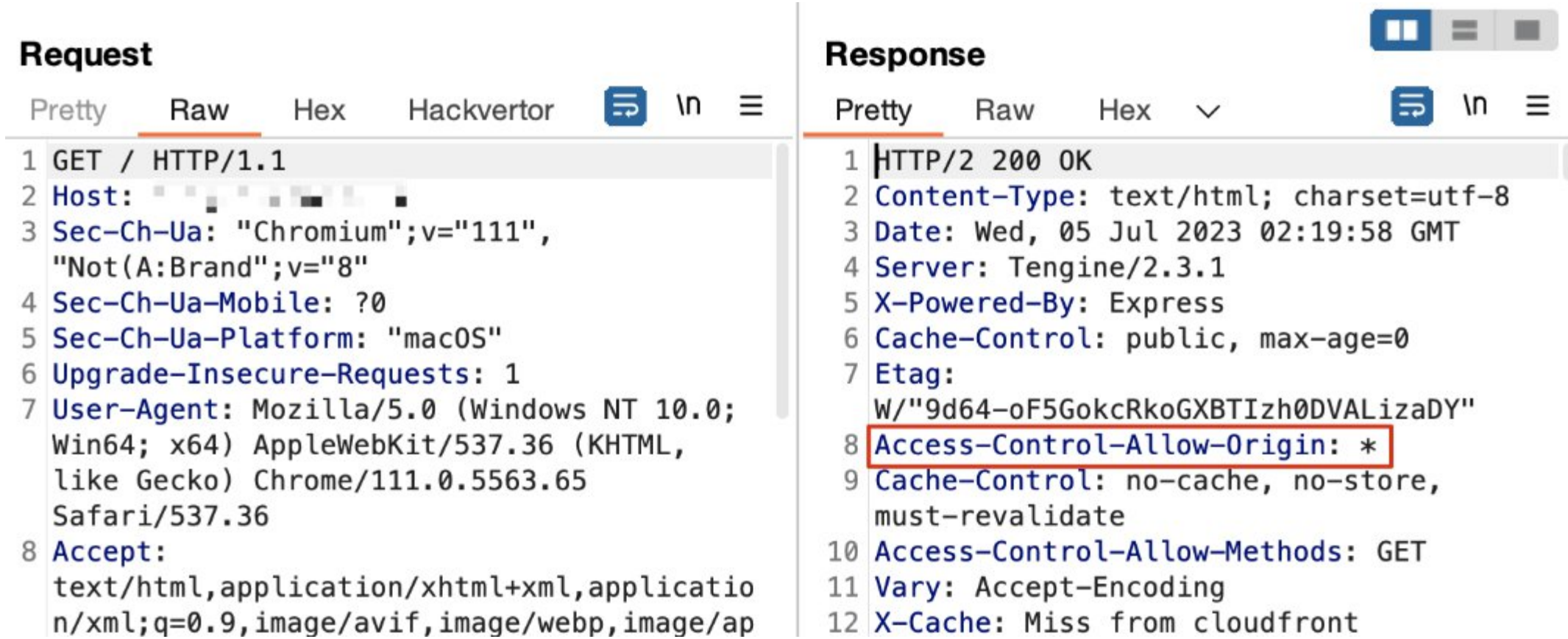
Dissecting CORS (and related vulnerability)

Agenda

- What is CORS?
- When and How to use it?
- Workshop-like, demo based.
- Can it be attacked? (and How?)
  - Is Access-Control-Allow-Origin: * secure?
  - Is Access-Control-Allow-Credentials: true secure?

Is this a risk?



Request

Pretty Raw Hex Hackvertor \n

```
1 GET / HTTP/1.1
2 Host: [redacted]
3 Sec-Ch-Ua: "Chromium";v="111",
  "Not(A:Brand";v="8"
4 Sec-Ch-Ua-Mobile: ?0
5 Sec-Ch-Ua-Platform: "macOS"
6 Upgrade-Insecure-Requests: 1
7 User-Agent: Mozilla/5.0 (Windows NT 10.0;
  Win64; x64) AppleWebKit/537.36 (KHTML,
  like Gecko) Chrome/111.0.5563.65
  Safari/537.36
8 Accept:
  text/html,application/xhtml+xml,applicatio
  n/xml;q=0.9,image/avif,image/webp,image/ap
```

Response

Pretty Raw Hex \n

```
1 HTTP/2 200 OK
2 Content-Type: text/html; charset=utf-8
3 Date: Wed, 05 Jul 2023 02:19:58 GMT
4 Server: Tengine/2.3.1
5 X-Powered-By: Express
6 Cache-Control: public, max-age=0
7 Etag:
  W/"9d64-oF5GokcRkoGXBTIzh0DVALizaDY"
8 Access-Control-Allow-Origin: *
9 Cache-Control: no-cache, no-store,
  must-revalidate
10 Access-Control-Allow-Methods: GET
11 Vary: Accept-Encoding
12 X-Cache: Miss from cloudfront
```

How about this?

Request

Pretty Raw Hex Hackvertor

```
1  GET /api/manage/teams/list HTTP/1.1
2  Host: [redacted]
3  Cookie: LoginType=E; Admin-Token=
   [redacted]
4  [redacted]
5  Accept: application/json, text/plain, */*
6  Sec-Ch-Ua-Mobile: ?0
7  Origin: https://evil.com/
8  Authorization: Bearer
   [redacted]
9  [redacted]
10 Sec-Ch-Ua-Platform: "macOS"
11 Sec-Fetch-Site: same-origin
12 Sec-Fetch-Mode: cors
13 Sec-Fetch-Dest: empty
```

Response

Pretty Raw Hex Render Hackvertor

```
1  HTTP/1.1 200
2  Server: Tengine/2.4.0
3  Date: Wed, 05 Jul 2023 02:24:23 GMT
4  Content-Type: application/json
5  Connection: close
6  Vary: Origin
7  Vary: Access-Control-Request-Method
8  Vary: Access-Control-Request-Headers
9  Access-Control-Allow-Origin: https://evil.com/
10 Access-Control-Allow-Credentials: true
11 X-Content-Type-Options: nosniff
12 X-XSS-Protection: 1; mode=block
13 Cache-Control: no-cache, no-store, max-age=0, mus
14 Pragma: no-cache
15 Expires: 0
16 Content-Length: 23142
17
18 {
     "msg":"操作成功",
     "code":200,
     "data":[
```

What is CORS?

- Cross-Origin Resource Sharing
  - A system, consisting of transmitting HTTP headers, that determines whether browsers block front-end JavaScript code from accessing responses for cross-origin requests.
- Cross-Origin: different origin
  - Origin: scheme://host:port
- Resource: any data, media or functionality fetched form a URL(we only care about data today)

When to use CORS?

- Fetch data from different origins
- Security concern
  - Public APIs? (✔)
  - Private APIs? (✗)

Access-Control-Allow-Origin

- A response header which indicates whether the response can be shared with requesting code from the given origin

- Three types of value:
  - *
  - <origin> a specific origin (cannot be *.example.com)
  - null (generally not recommended, from file:// or sandbox iframe)
- Demo

Access-Control-Allow-Credentials

- A response header tells browsers whether to expose the response to the front-end JavaScript code when the request's credentials mode (Request.credentials) is include.

```
fetch(url, {
  credentials: "include",
});
```

```
const xhr = new XMLHttpRequest();
xhr.open("GET", "http://example.com/", true);
xhr.withCredentials = true;
xhr.send(null);
```

- If Access-Control-Allow-Credential=true, Access-Control-Allow-Origin cannot be `*` (for security reason)
- Demo

Access-Control-Expose-Headers

- A response header which allows a server to indicate which response headers should be accessible for the front-end JavaScript code, in response to a cross-origin request.

- Usage scenario: customized authentication header

- Demo

Access-Control-Request-Headers

- A request header used by browser when issuing a preflight request to let server knows which headers will be sent when the the actual request is made.
- Preflight request: an OPTIONS request before sending the actual non-simple request which meet any of the following:
  - HTTP method other than GET, HEAD, POST
  - Content-Type other than application/x-www-form-urlencode, multipart/form-data, text/plain
  - Customized headers

Access-Control-Allow-Headers

- A response header which is used in response to a preflight request which includes the Access-Control-Request-Headers to indicate which HTTP headers can be used during the actual request.

- Demo

A Few More headers

- Access-Control-Request-Method: a request header used by browsers when issuing a preflight request, to let the server know which HTTP method will be used when the actual request is made.
- Access-Control-Allow-Methods: a response header in response to a preflight request which includes the Access-Control-Request-Method to indicate which HTTP method can be used during the actual request.
- Access-Control-Max-Age: a response header which indicates how long the results of a preflight request can be cached.

Attack Scenario

- Is Access-Control-Allow-Origin: * secure?
- Is Access-Control-Allow-Credentials: true secure?

Is Access-Control-Allow-Origin: * secure?

- * is mainly for public APIs, which is (mostly) publicly accessible
  - Attackers can directly access it

- But what if some sensitive information hosting in the internal website?
  - PNA(Private Network Access)
  - Demo

Is Access-Control-Allow-Credentials: true secure?

- Can carry cookies in the CORS requests now
  - Access to private APIs
  - retrieve sensitive information

- Demo

Stumbling Block: SameSite cookie

- Controls whether or not a cookie is sent with cross-site requests, providing some protection against cross-site request forgery attacks
- What is considered as same site:
  - Same eTLD(effective TLD, registrable domains) + 1 (public suffix)
  - Schemeful, port insensitive

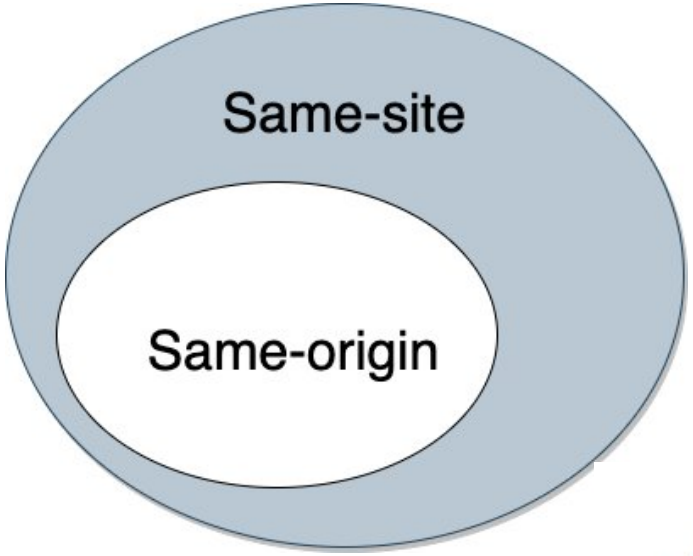
Request from	Request to	Same-site?	Same-origin?
`https://example.com`	`https://example.com`	Yes	Yes
`https://app.example.com`	`https://intranet.example.com`	Yes	No: mismatched domain name
`https://example.com`	`https://example.com:8080`	Yes	No: mismatched port
`https://example.com`	`https://example.co.uk`	No: mismatched eTLD	No: mismatched domain name
`https://example.com`	`http://example.com`	No: mismatched scheme	No: mismatched scheme

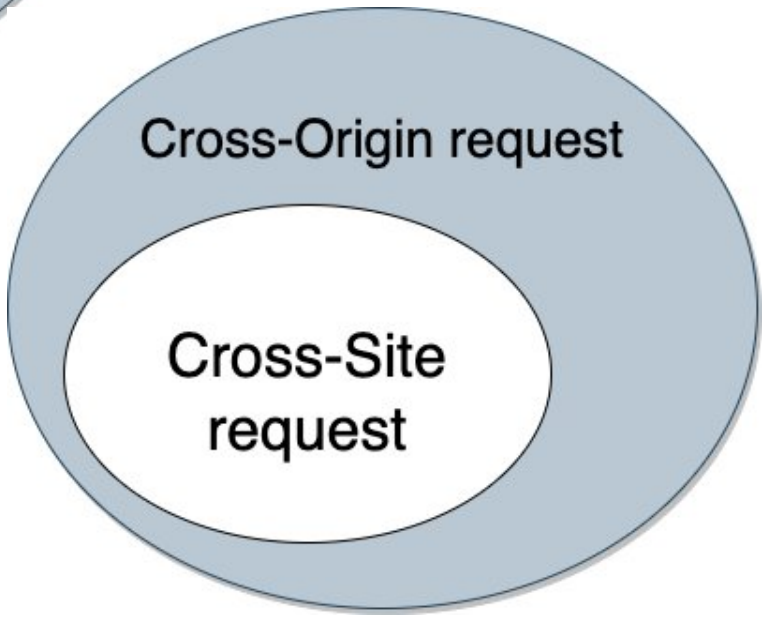
SameSite cookie

- Possible values:
  - None: browser sends the cookie with both cross-site and same-site requests.
  - Lax: browser sends the cookie with top-level navigation cross-site requests (default value for Chrome)
  - Strict: browser sends the cookie only for same-site requests
- Note: None must be used together with Secure=True(cookie can only be sent in secure context) for Chrome
- To test for samesite cookie: https://samesite-sandbox.glitch.me/
- ~~Firefox is a weirdo~~

How does it affect CORS?

- Same-origin VS same-site
- Same-origin:
  - Same scheme
  - Same host
  - Same port
- Same-site:
  - Same Scheme
  - Same eTLD(more relaxed than same host)
  - ~~Same port?~~





How to exploit it?

- If SameSite=None, Secure=True
  - Host a malicious page on your own website
  - Demo

- If SameSite=Lax
  - exploit using cross-origin but same-site request
  - Subdomain takeover, XSS on subdomains etc
  - Demo

How to mitigate it?

- White-list based Access-Control-Allow-Origin
  - Don’t reflect the Origin request header
  - Don’t do substring match
    - extractHost(Origin).endwith(“cors-lab.com”)
      - Bypass with aaaaaaaaacors-lab.com
    - extractHost(Origin).find(“cors-lab.com”) != -1
      - Bypass with cors-lab.com.malicious.com
- Correctly configure SameSite attributes