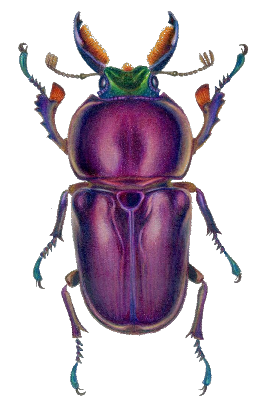
Demystifying Cookies and Tokens

A Security Redefinition

HolyBugx

Me

- Emad Roshan (@HolyBugx)
- Bug Bounty Hunter (~1 years)
- Passionate about Application Security Research
- Run a blog to better explain AppSec at SecurityFlow.io



Agenda

- Authentication & Authorization
- Sessions & Cookies
    - Concepts
    - SameSite Cookies & Demo
- Token Based Authentication
    - Concepts
- Cross-domain attacks (CSRF, CORS, etc.)
- Real world vulnerability explained
- Conclusion

Authentication & Authorization

- Authentication: Verifying users identity (401 Unauthorized)
- Authorization: Verifying users permissions (403 Forbidden)

Authentication Models

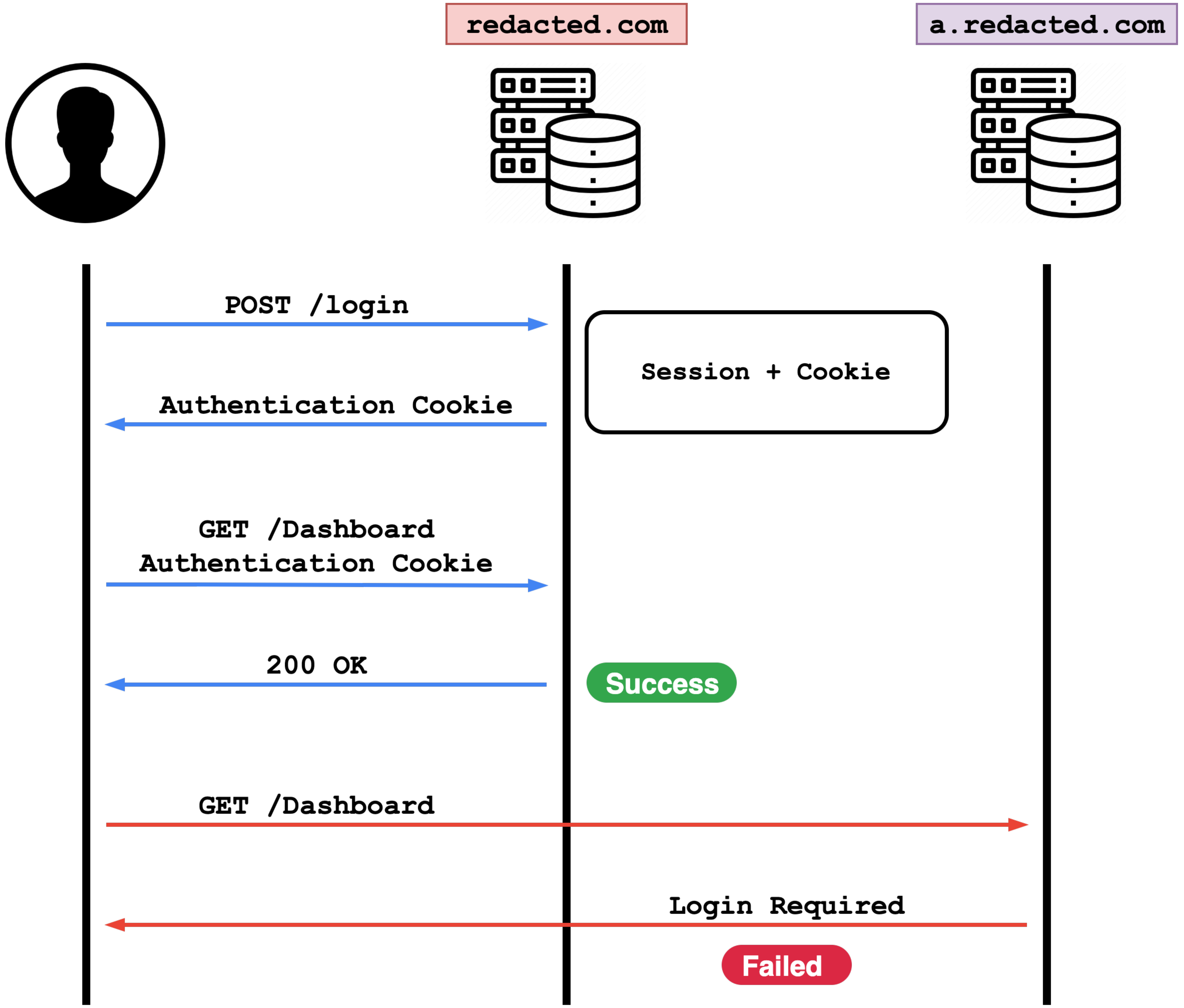
- Cookie-based
- Token-based
    - JWT (JSON Web Token)
    - OAuth
- Single Sign-On
- SAML

Sessions

- User session is stored server-side (stateful)
    - Database e.g. Postgres, MongoDB
    - Cache e.g. Redis, Memcached
    - File system
- User is identified by his session ID.
    - Randomly generated.
    - Carries no sensitive user data.

Cookies

- Used for Session management, Personalization, User Tracking
- Consist of names, values, (optional) attributes
- Set with Set-Cookie header by the server
- With every subsequent request, browser sends it back using the Cookie HTTP header


redacted.com
a.redacted.com
POST /login
Session + Cookie
Authentication Cookie
GET /Dashboard
Authentication Cookie
200 OK
Success
GET /Dashboard
Login Required
Failed

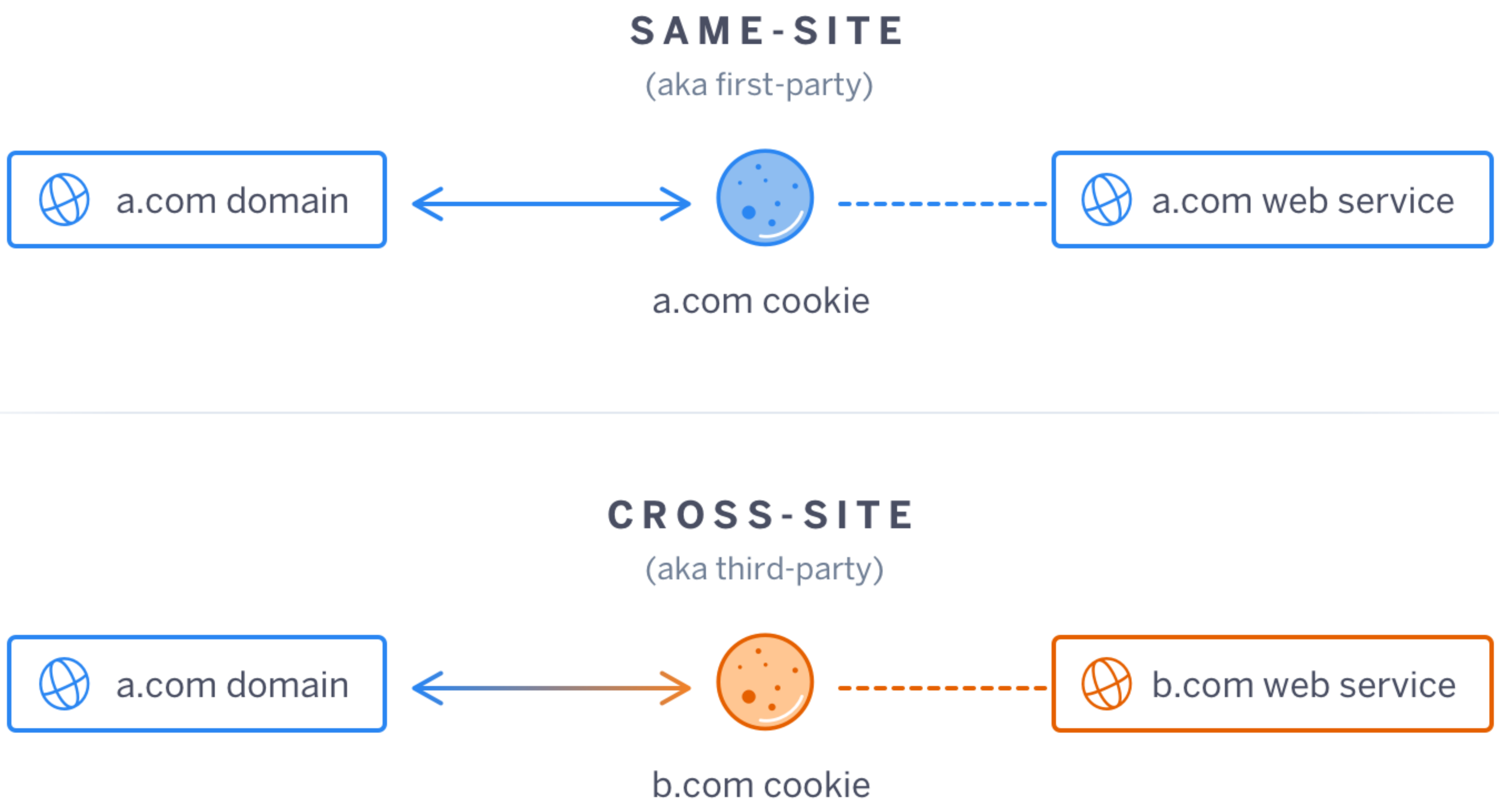
Set-Cookie

```
Set-Cookie: id=a3fWa; Expires=Thu, 21 Oct 2021 07:28:00 GMT; Secure; HttpOnly

Set-Cookie: id=a3fWa; Domain=redacted.com; Path=/; SameSite=Strict
```

Domain Attribute

- Specifies which hosts are allowed to receive the cookie
- If omitted, defaults to the host of the current document URL, not including subdomains.
- Contrary to earlier specifications, leading dots in domain names (.example.com) are ignored.
- Multiple host/domain values are not allowed, but if a domain is specified, then subdomains are always included.

First-Party vs. Third-Party Cookie


SAME-SITE
(aka first-party)
a.com domain
a.com cookie
a.com web service
CROSS-SITE
(aka third-party)
a.com domain
b.com cookie
b.com web service

Same-Origin vs. Same-Site

Origin

$$\underbrace{\text{https}}_{\text{scheme}}\text{://}\underbrace{\text{www.example.com}}_{\text{host name}}\text{:}\underbrace{\text{443}}_{\text{port}}$$

Origin A	Origin B	Explanation of whether Origin A and B are "same-origin" or "cross-origin"
https://www.example.com:443	https://**www.evil.com**:443	cross-origin: different domains
	https://**example.com**:443	cross-origin: different subdomains
	https://**login**.example.com:443	cross-origin: different subdomains
	http://www.example.com:443	cross-origin: different schemes
	https://www.example.com:**80**	cross-origin: different ports
	https://www.example.com:443	**same-origin: exact match**
	https://www.example.com	**same-origin: implicit port number (443) matches**

https://web.dev/same-site-same-origin

Site

$$\text{https://www.}\underbrace{\text{example.}\overset{\text{eTLD}}{\overline{\text{com}}}}_{\text{eTLD+1}}\text{:443}$$

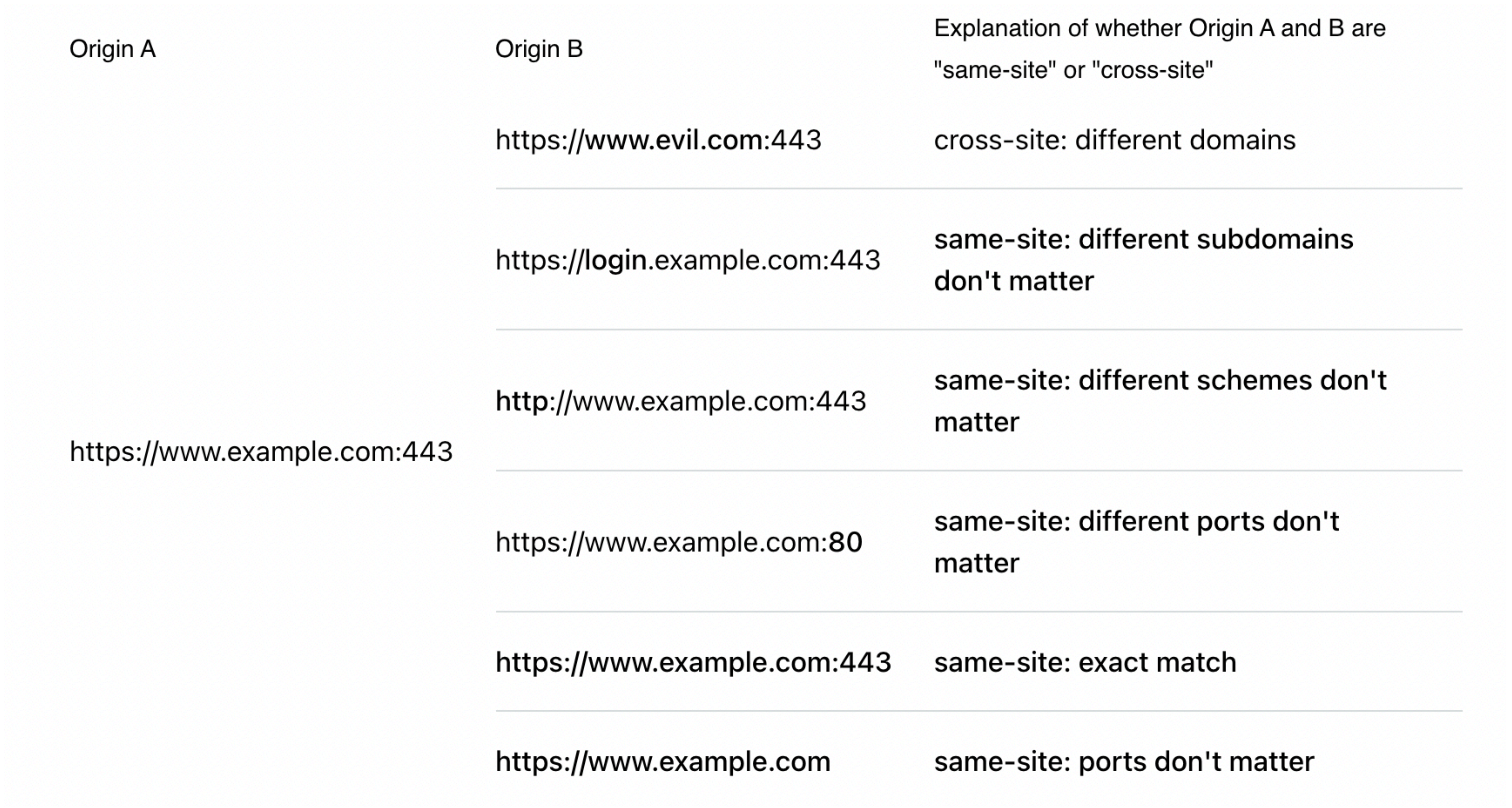
Site

“Site” is the combination of TLD and the domain part just before it.

https://www.$\boxed{\text{example.com}}$:443/login.php

$\downarrow$

Site & eTLD+1



Origin A	Origin B	Explanation of whether Origin A and B are "same-site" or "cross-site"
https://www.example.com:443	https://**www.evil.com**:443	cross-site: different domains
	https://**login**.example.com:443	**same-site: different subdomains don't matter**
	http://www.example.com:443	**same-site: different schemes don't matter**
	https://www.example.com:**80**	**same-site: different ports don't matter**
	https://www.example.com:443	**same-site: exact match**
	https://www.example.com	**same-site: ports don't matter**

https://web.dev/same-site-same-origin

Question

What is the “Site”?

https://www.example.co.uk:443

https://holybugx.github.io:443

What is the “Site”?

$\text{https://www.example}\underbrace{\text{.co.uk}}_{\text{eTLD}}\text{:443}$

$\text{https://holybugx}\underbrace{\text{.github.io}}_{\text{eTLD}}\text{:443}$

What is the “Site”?

Site & eTLD+1

https://www.$\boxed{\text{example.co.uk}}$:443

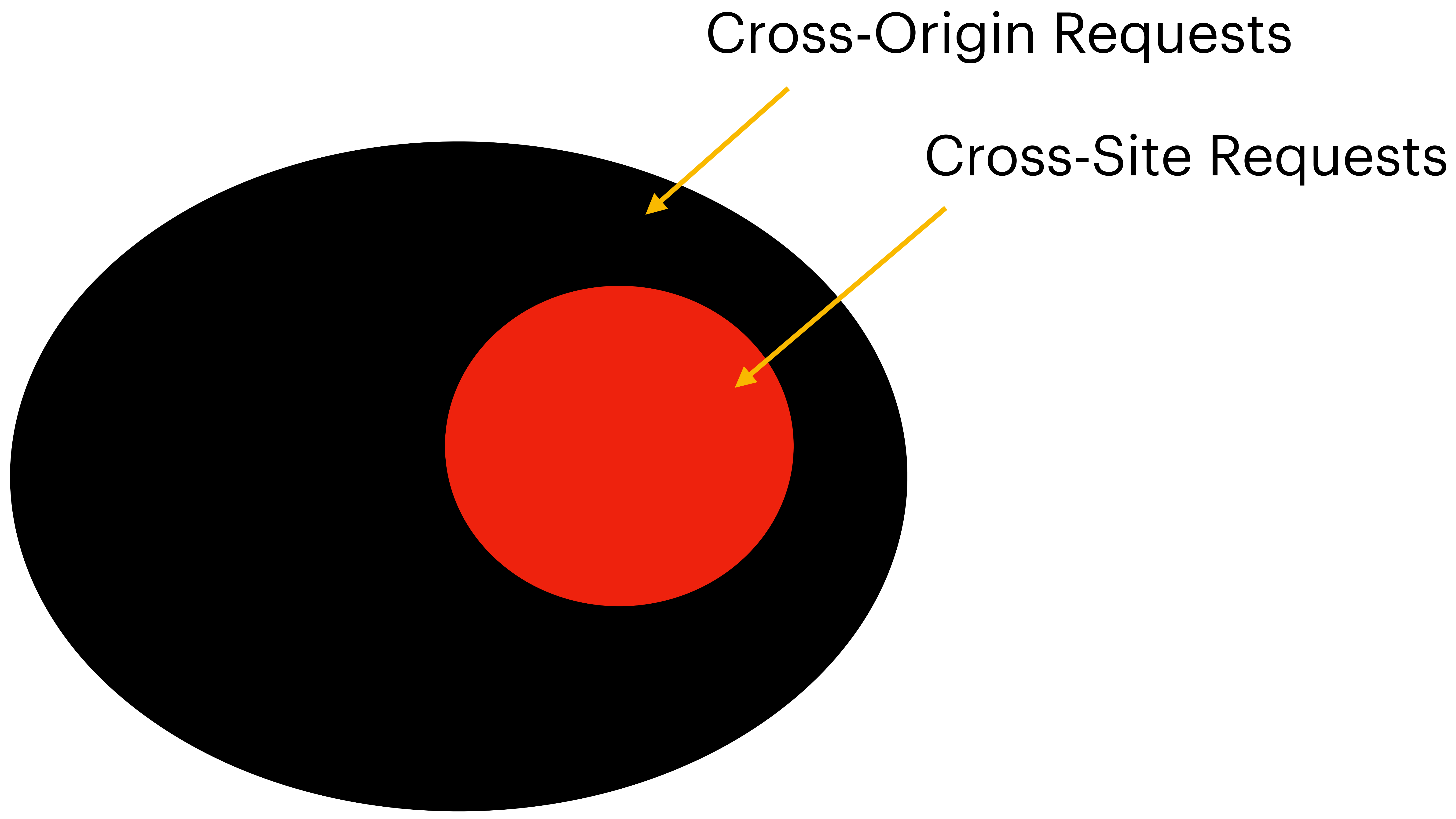
https://$\boxed{\text{holybugx.github.io}}$:443

Site & eTLD+1

eTLDs

- For domains like “.co.uk“ or “.github.io“ just using “.uk” or “.io” is not enough to identify the “Site”
- That’s why eTLDs are created!
- Full list of eTLDs are maintained at Public Suffix List

Conclusion



All Cross-Site Requests are necessarily Cross-Origin

SameSite Cookies

SameSite Cookies

- Controls whether a cookie is sent with cross-origin requests
- SameSite cookies options are:
    - Strict —> The cookie will not be sent along with requests initiated by third-party websites
    - Lax —> The cookie will be sent along with the GET request initiated by third-party websites
    - None —> Allows third-party cookies to track users
        - Needs the Secure flag to work

Lax Notes

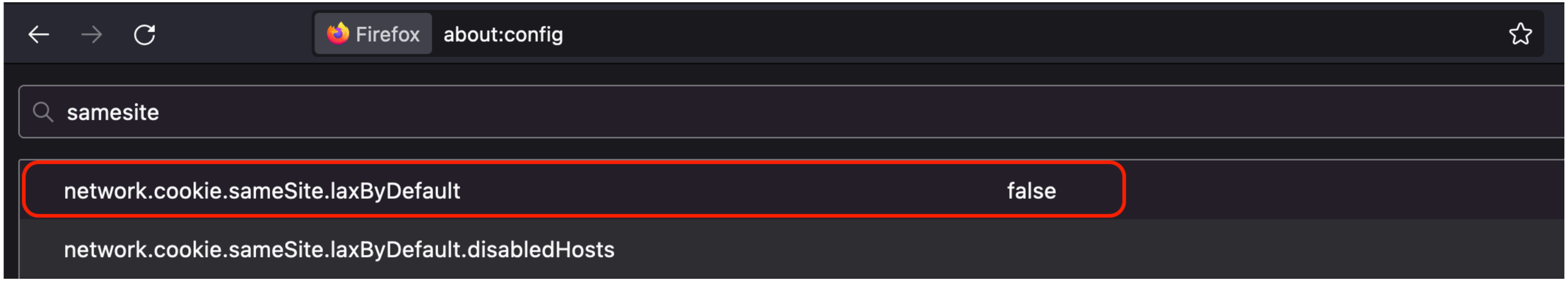
- If not specified, the default will be used as SameSite=Lax
- To send a cookie with a GET request, GET request being made must cause a top-level navigation
- Resources loaded with img, iframe, script tags do NOT cause top-level navigation, thus cookies set to Lax won't be sent with them

Lax vs. None

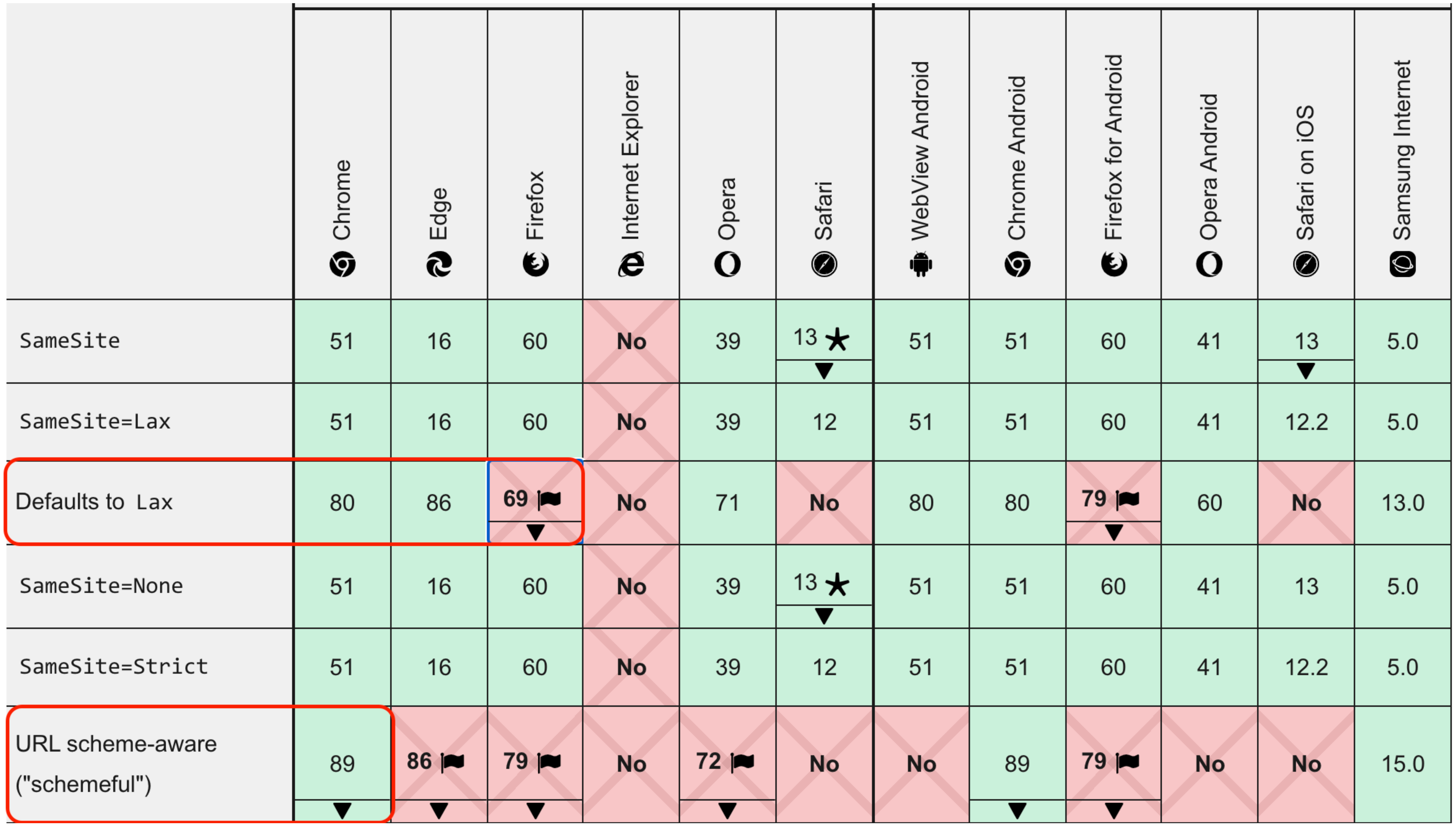
Request Type	Example Code	Cookies sent
Link	`<a href="..."></a>`	Normal, Lax
Perender	`<link rel="prerender" href=".."/>`	Normal, Lax
Form GET	`<form method="GET" action="...">`	Normal, Lax
Form POST	`<form method="POST" action="...">`	Normal
iframe	`<iframe src="..."></iframe>`	Normal
AJAX	`$.get("...")`	Normal
Image	`<img src="...">`	Normal

Schemeful SameSite and Lax Browser Compatibility

- Schemeful SameSite is where the Same-Site term relies on the HTTP scheme as well, but it's only supported on Chrome 89+ at the time.
- Firefox doesn't set the not-specified SameSite cookies to Lax by default, you need to manually adjust it.



```
network.cookie.sameSite.laxByDefault                          false
network.cookie.sameSite.laxByDefault.disabledHosts
```



	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
`SameSite`	51	16	60	No	39	13 ★ ▼	51	51	60	41	13 ▼	5.0
`SameSite=Lax`	51	16	60	No	39	12	51	51	60	41	12.2	5.0
Defaults to `Lax`	80	86	69 ⚑ ▼	No	71	No	80	80	79 ⚑ ▼	60	No	13.0
`SameSite=None`	51	16	60	No	39	13 ★ ▼	51	51	60	41	13	5.0
`SameSite=Strict`	51	16	60	No	39	12	51	51	60	41	12.2	5.0
URL scheme-aware ("schemeful")	89 ▼	86 ⚑ ▼	79 ⚑ ▼	No	72 ⚑ ▼	No	No	89 ▼	79 ⚑ ▼	No	No	15.0

Demo

Cookies Security Issues

- Cross-Site Request Forgery - CSRF
- Cross-Site Scripting - XSS
- Cross-Origin Resource Sharing - CORS
- Other rare attacks e.g Session Fixation, Cookie Tossing, etc.

Cookies are sent by default in all browsers.

Tokens

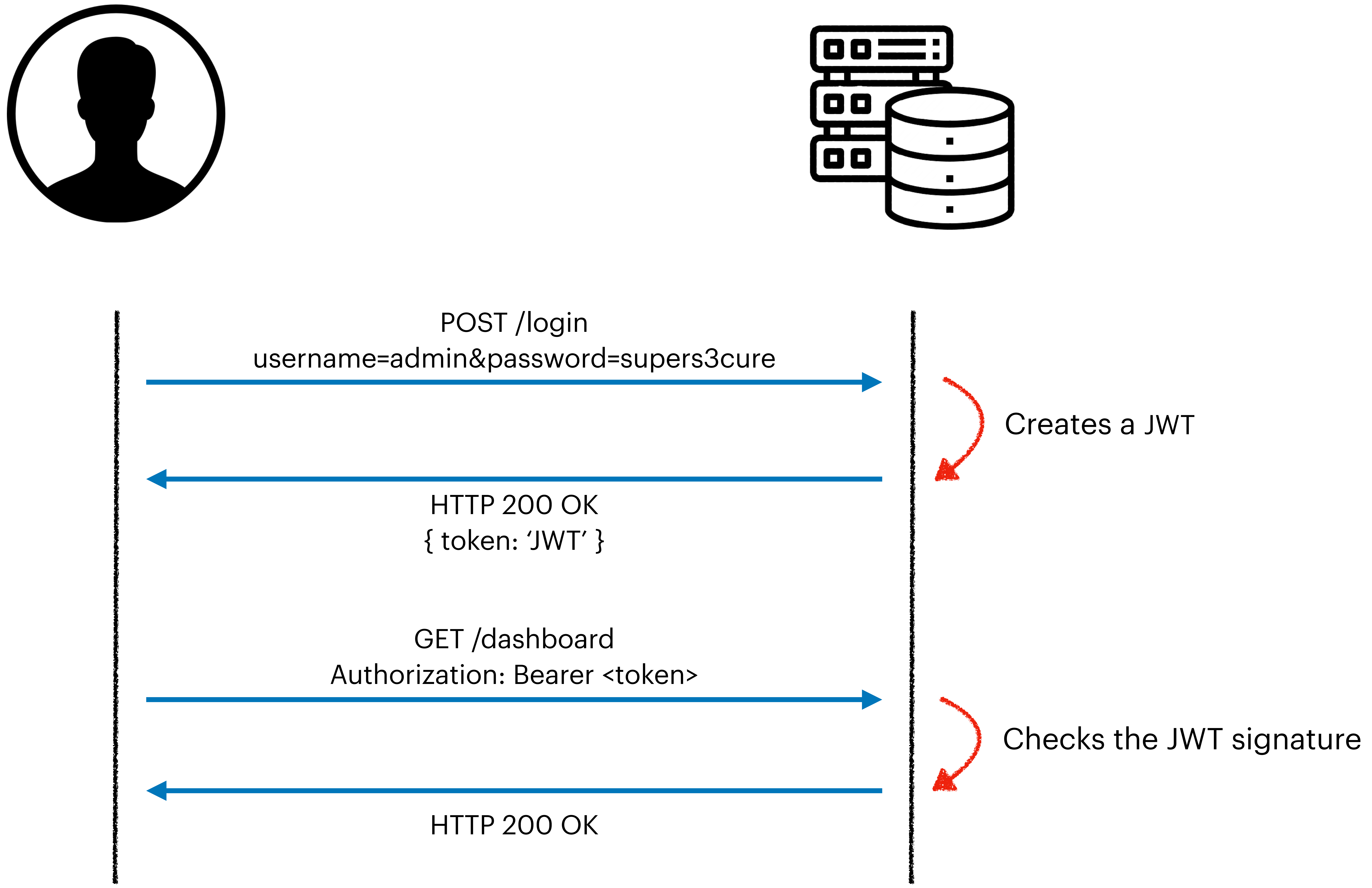
Tokens

- Tokens are not stored server-side (stateless)
- Tokens are signed with a secret (tamper proof)
- Tokens are both opaque and self-contained
- Tokens can simply be revoked
- Tokens are commonly sent in the “Authorization” HTTP header
- Tokens are used in SPAs, APIs, and various Web&Mobile Apps

Storage

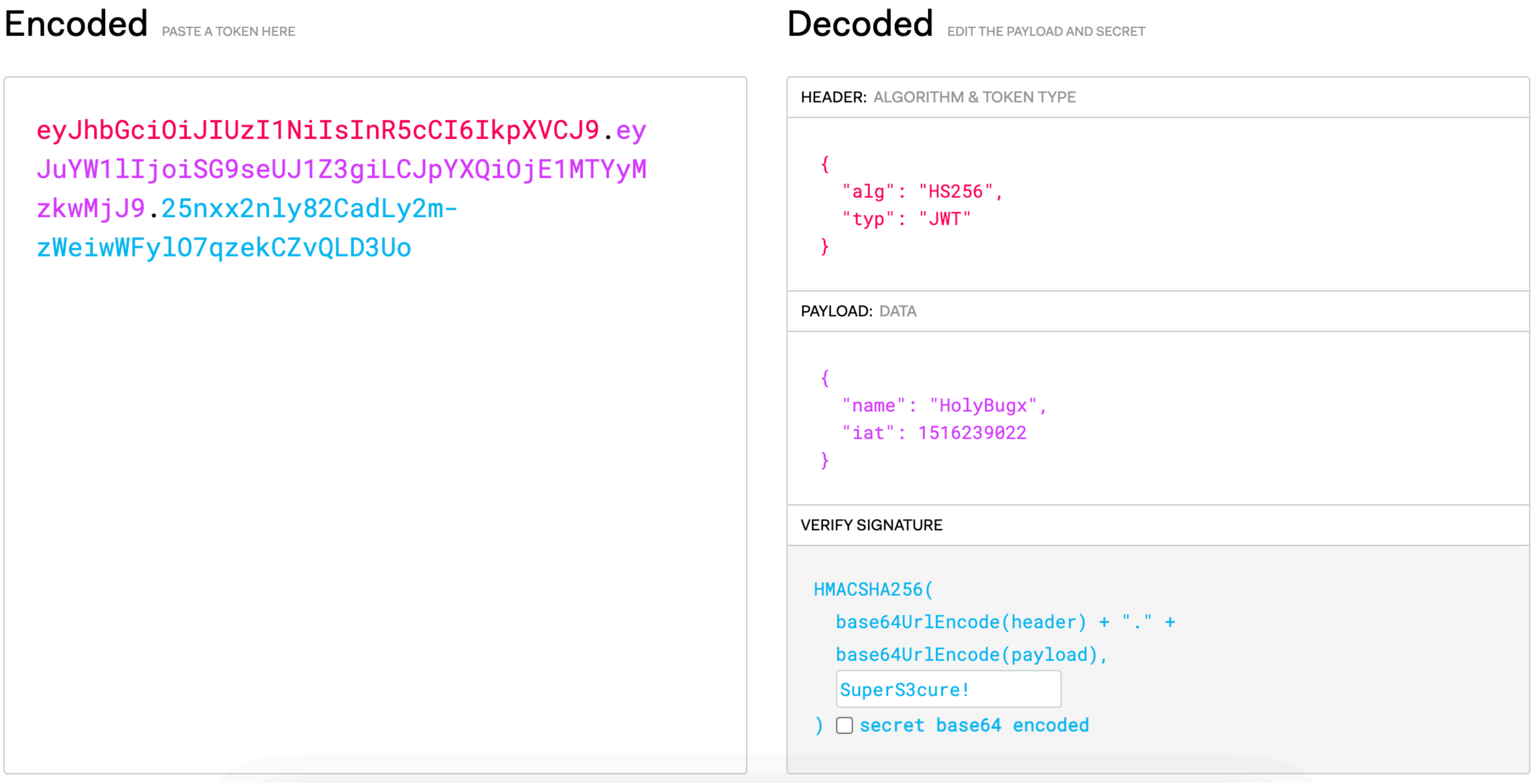
- JWTs are stored in localStorage and sessionStorage
- What's the difference?
    - localStorage: no expiration date
    - sessionStorage: gets cleared after closing the tab (unique per tab)

localStorage is more flexible for web developers!
🙂


POST /login
username=admin&password=supers3cure
Creates a JWT
HTTP 200 OK
{ token: 'JWT' }
GET /dashboard
Authorization: Bearer <token>
Checks the JWT signature
HTTP 200 OK

JSON Web Tokens

- Most famous token based authentication solution
- JWTs consist of three parts separated by dots (.), which are:
    - Header, contains the type of the token and the hashing algorithm
    - Payload, contains the claims
    - Signature, contains the encoded header, the encoded payload, a secret, and the algorithm specified in the header.
- JWTs are usually self-contained, signed and encoded



Encoded PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiSG9seUJ1Z3giLCJpYXQiOjE1MTYyMzkwMjJ9.25nxx2nly82CadLy2m-zWeiwWFyl07qzekCZvQLD3Uo

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "name": "HolyBugx",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  SuperS3cure!
) ☐ secret base64 encoded
```

Token Based Auth Issues (?)

- No CSRF Issues
- No CORS Issues
- XSS is still an issue (localStorage)

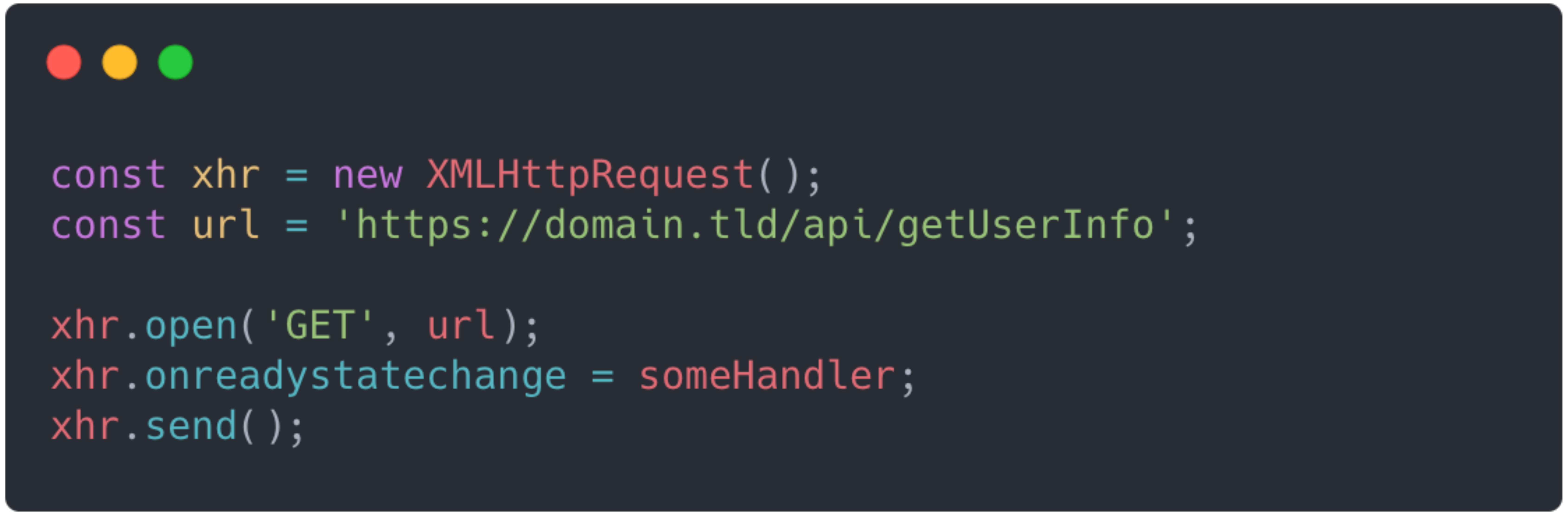
Well, Doesn't Token-Based Authentication fixes all Cross-Domain attacks?
🤔

CORS Preflights

- Some requests are called “Simple” and some are called “Preflight”
- What are the simple requests?
  - If there is no custom HTTP header (anything besides Accept, Accept-Language, Content-Language, Content-Type, DPR, Downlink, Save-Data, Viewport-Width, Width)
  - If HTTP verbs are GET, POST, and Head
  - If HTTP verb is POST and the content-type is text/plain, multipart/form-data, application/x-www-form-urlencoded

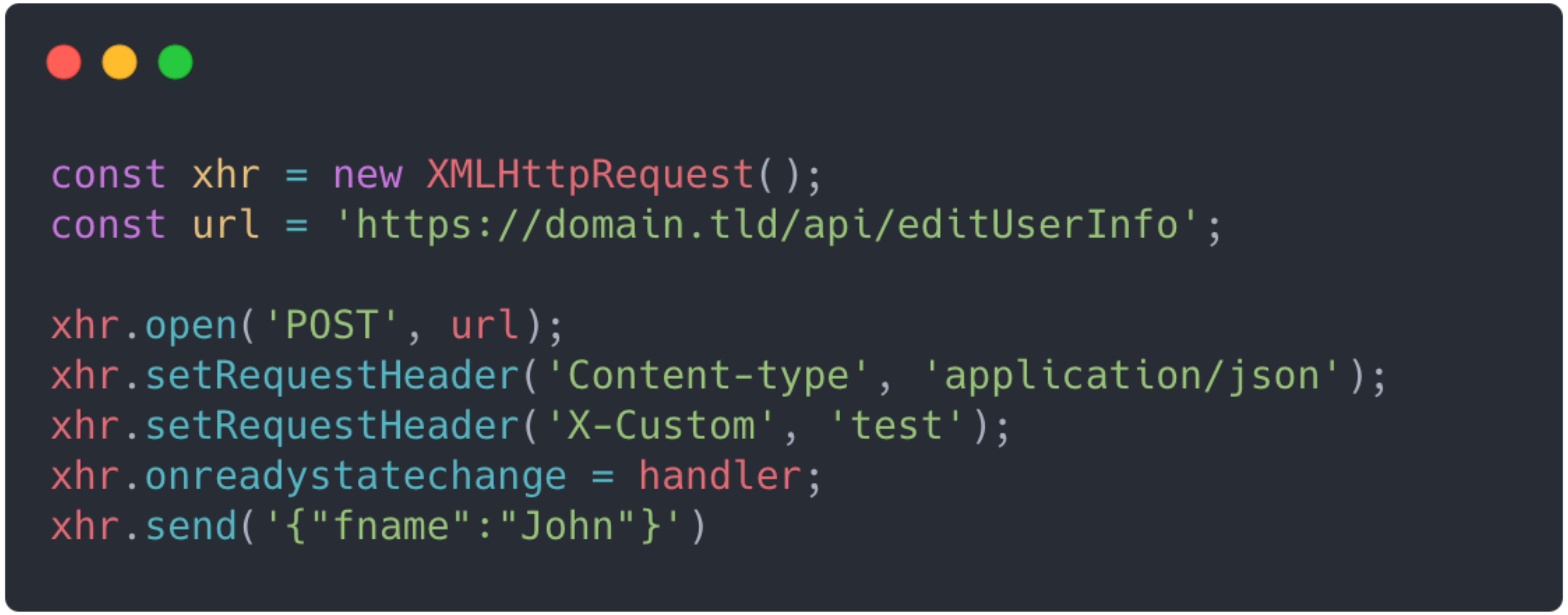
Anything besides these is called preflight request.

Simple Request

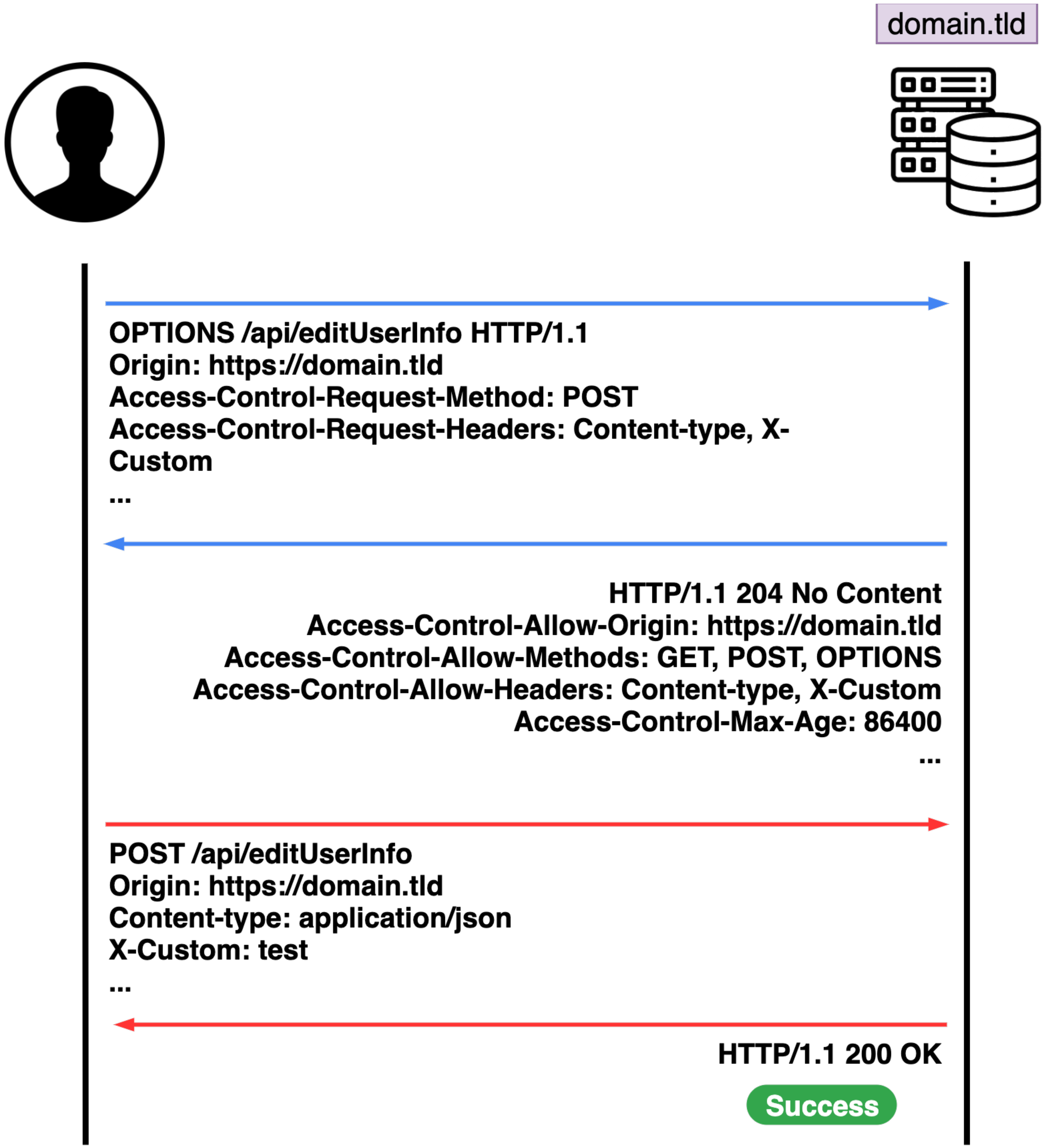


```
const xhr = new XMLHttpRequest();
const url = 'https://domain.tld/api/getUserInfo';

xhr.open('GET', url);
xhr.onreadystatechange = someHandler;
xhr.send();
```

Preflight Request



```
const xhr = new XMLHttpRequest();
const url = 'https://domain.tld/api/editUserInfo';

xhr.open('POST', url);
xhr.setRequestHeader('Content-type', 'application/json');
xhr.setRequestHeader('X-Custom', 'test');
xhr.onreadystatechange = handler;
xhr.send('{"fname":"John"}')
```



```
OPTIONS /api/editUserInfo HTTP/1.1
Origin: https://domain.tld
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Content-type, X-
Custom
...
```

```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://domain.tld
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Allow-Headers: Content-type, X-Custom
Access-Control-Max-Age: 86400
...
```

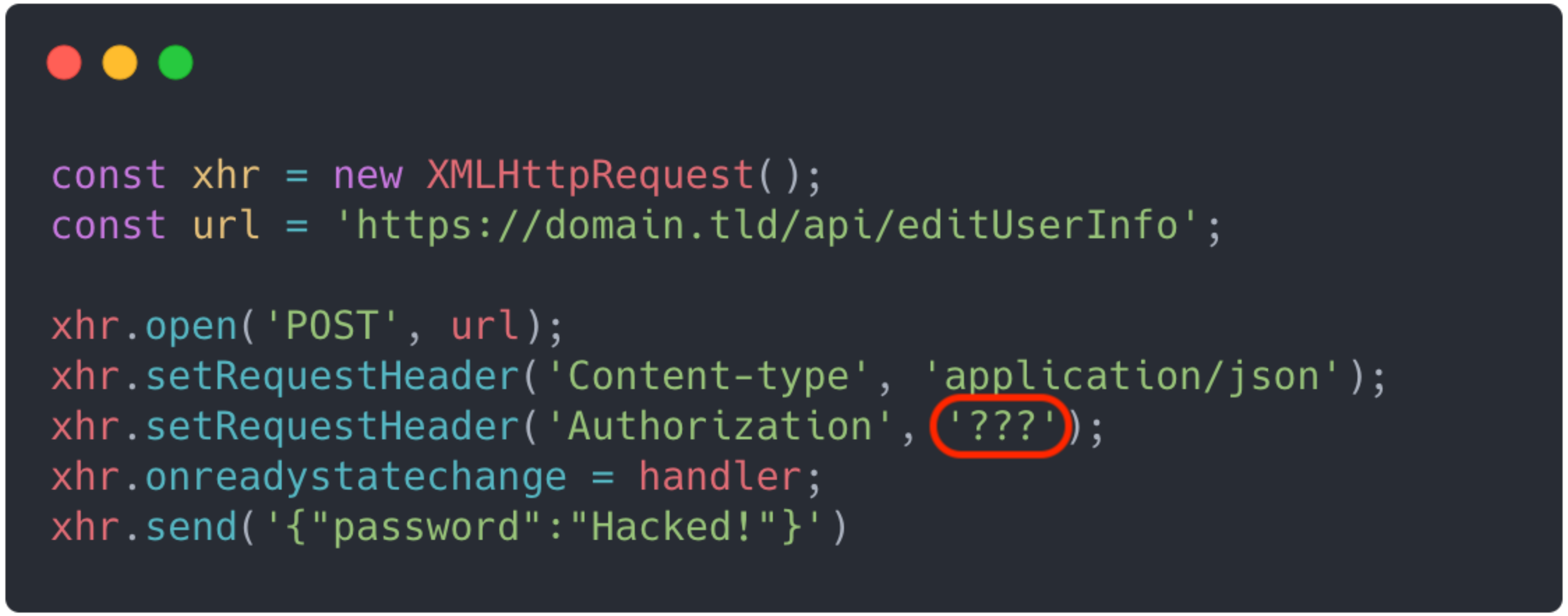
```
POST /api/editUserInfo
Origin: https://domain.tld
Content-type: application/json
X-Custom: test
...
```

```
HTTP/1.1 200 OK
```

As an attacker, you might think:

"The authorization header is not sent by default, but can I force the browser to send it?"

- Authorization headers are not sent by default on the browser
- We can't set the Authorization header to be sent using XHR as we don't know the value



```
const xhr = new XMLHttpRequest();
const url = 'https://domain.tld/api/editUserInfo';

xhr.open('POST', url);
xhr.setRequestHeader('Content-type', 'application/json');
xhr.setRequestHeader('Authorization', '???');
xhr.onreadystatechange = handler;
xhr.send('{"password":"Hacked!"}')
```

What about other methods?

- There are several other authentication & authorization models:
    - OpenID
    - OAuth
    - SSO
    - SAML
- Not enough time :(
- I’m planning to release the v2 version of this talk, including those :)

Real World Vulnerability

How I chained a misconfiguration and an XSS to achieve CSRF

Testing for CSRF

- Application was using CSRF tokens in the HTTP POST body for all state-changing requests
- Most hackers try the following techniques:
  - Removing the CSRF parameter
  - Removing the CSRF parameter value
  - Sharing CSRF tokens
  - Token length tampering
  - Verb tampering

None of the mentioned techniques worked.

But I tried another rare technique.

I removed the X-CSRF-Token from HTTP POST body
And added it to the Cookie header as a parameter

And it worked!

But this is not an issue as we don’t have other user’s tokens :(

I tried the mentioned bypassing methods again in the cookie header...

And I realized the token value is not checked at all.
Only the existence of the parameter was being checked.

And this issue was not possible in the HTTP POST parameter.

This is for sure an issue!

Probably, developers were thinking that an attacker can not set cookies for others.

“Are you sure, my developer friend 🙂?”

The question was: *“How can I force this parameter to be added in other user’s cookies?”*

Using Cross-Site Scripting we have access to the victim’s DOM
And we can tamper with the cookies through document.cookie

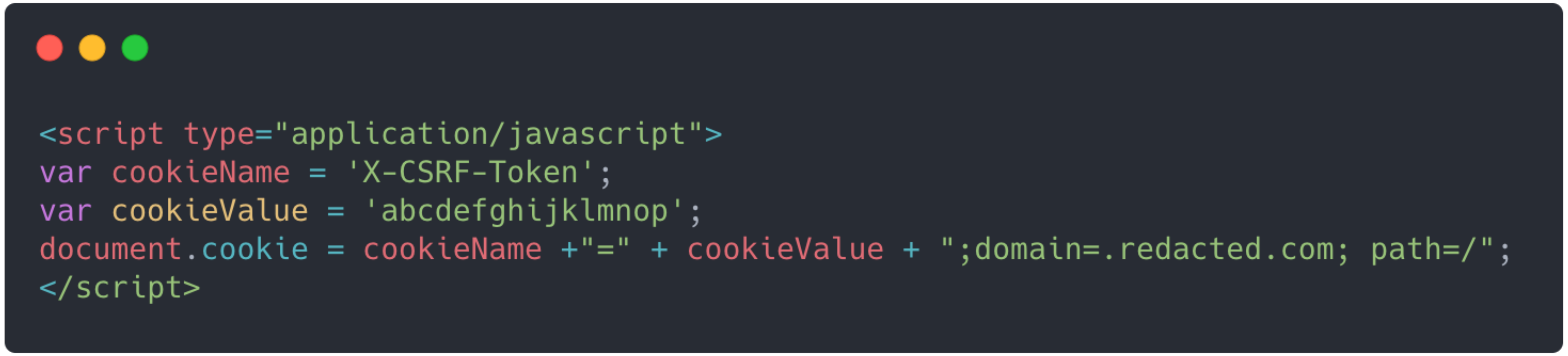
I didn’t find any XSS on their domain :(

“What if I find an XSS on out-of-scope subdomains and use that to set a 'Domain=.redacted.com' attribute. so, all subdomains are included?”

Let’s find out!

I found an XSS on an out-of-scope subdomain in a couple of hours!

Time to prove my point!



```html
<script type="application/javascript">
var cookieName = 'X-CSRF-Token';
var cookieValue = 'abcdefghijklmnop';
document.cookie = cookieName +"=" + cookieValue + ";domain=.redacted.com; path=/";
</script>
```

- Using the XSS on the out-of-scope I was able to overwrite cookies (document.cookie)
- The X-CSRF-Token length was the only check
- The key point of this attack was that I used the “domain” cookie attribute so all in-scope subdomains were affected

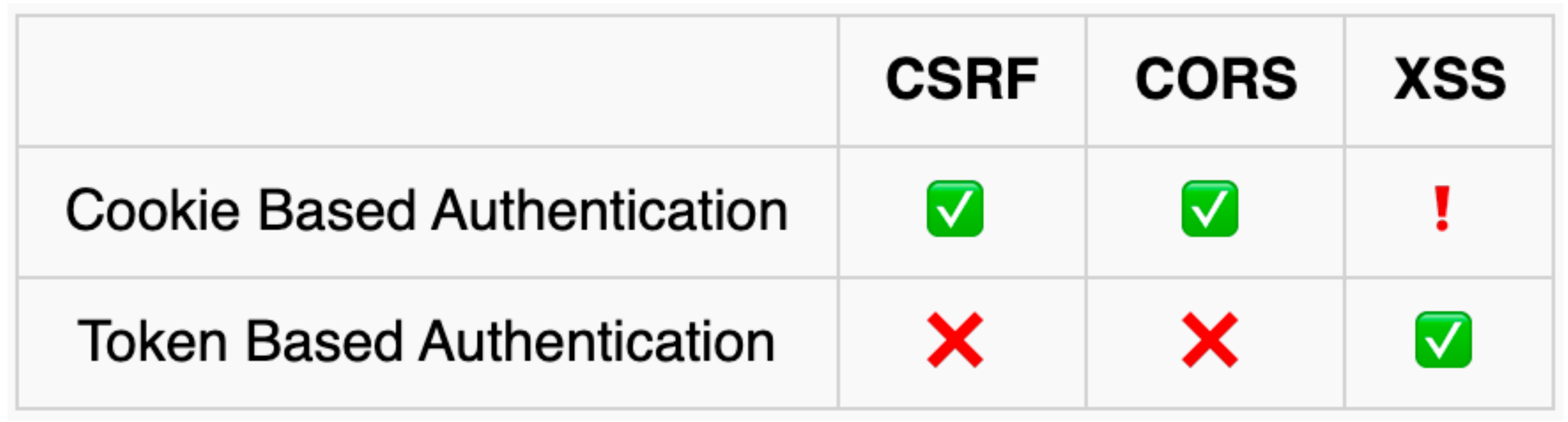
This was how I chained multiple misconfigurations and an XSS to achieve a working CSRF on the main site.

🙂✌🏻

Conclusion

JWT Implementation & Security Issues

- Storing JWTs in “Cookie” header
    - CSRF is still possible! (Without further security like CSRF tokens)
    - XSS can steal the JWT if the cookies are not signed as httponly
    - XSS can lead to sending XHR requests on behalf of other users
- Storing JWTs in “Authorization” header
    - No CSRF (Authorization header is not sent by the browsers)
        - Authorization header will be sent using XHR
        - CSRF Tokens doesn’t provide “*extra*” security
    - XSS can lead to token theft from localStorage



	CSRF	CORS	XSS
Cookie Based Authentication	✅	✅	❗
Token Based Authentication	❌	❌	✅

- The table is based on Authorization: Bearer <token> implementation.
- XSS in cookie-based applications doesn’t lead to direct account takeover if httponly is set. however, the attacker has access to the victim’s DOM.
- XSS in token-based authentication usually leads to direct account takeover.
- CORS and CSRF is not possible in properly implemented token-based applications.

Thanks for watching!

HolyBugx

SecurityFlow.io