

# Protect against Cross-Site Request Forgery (CSRF)

Peter Cosemans - Michiel Olijslagers



# What is CSRF



# What is CSRF

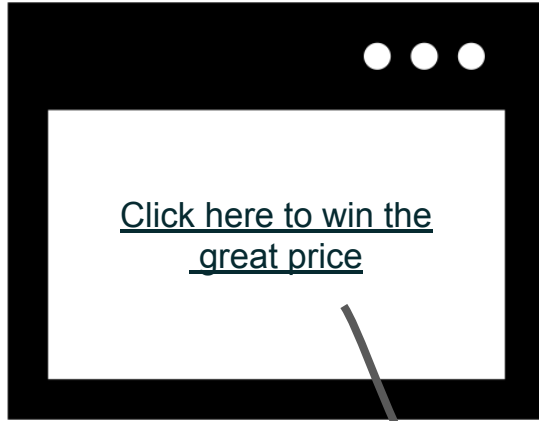
CSRF stands for Cross-Site Request Forgery. It's a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the website trusts. In a CSRF attack, an innocent end user is tricked by an attacker into submitting a web request that they did not intend.

```
https://vulnerable.site.com/email/change?email=pwned@evil-user.net
```

# What is CSRF

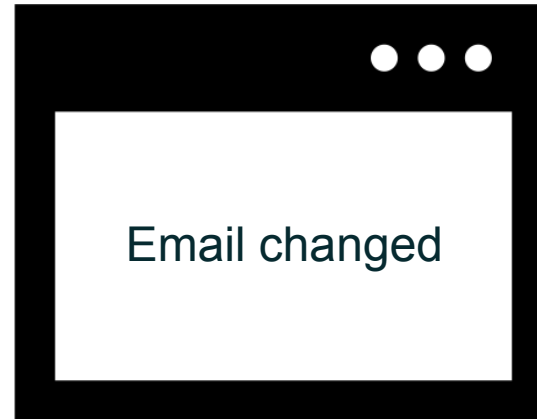
The user was previous logged in to the vulnerable site, so the session cookie is send with the request

<https://evil.hacker.com>



Session

[https://vulnerable.site.com/email/change?  
email=pwned@evil-user.ru](https://vulnerable.site.com/email/change?email=pwned@evil-user.ru)



# What is CSRF

For example, suppose an application contains a function that lets the user change the email address on their account. When a user performs this action, they make an HTTP request like the following:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=yvthwsztyeQkAPzeQ5gHgTvlyxHfsAfE

email=peter@mysite.com
```

# What is CSRF

With these conditions in place, the attacker can construct a web page containing the following HTML

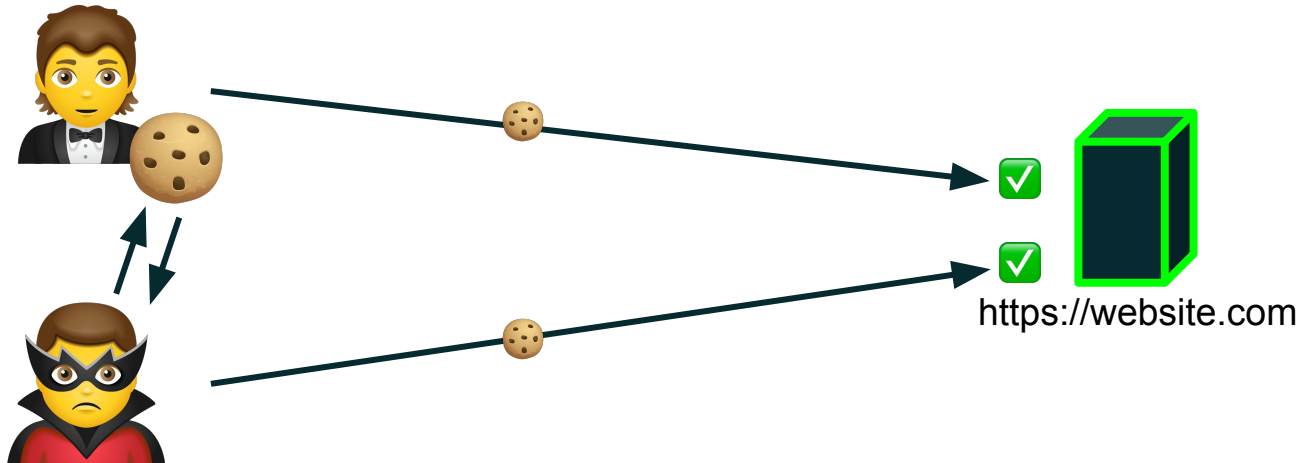
```
<html>
  <body>
    <form action="https://vulnerable-website.com/email/change" method="POST">
      <input type="hidden" name="email" value="pwned@evil-user.ru" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

# Cross-Site Request Forgery In Action



# Why does a CSRF attack work?

- Browsers automatically send cookies
  - Also session cookies
- When user is authenticated
  - Site cannot distinguish between legitimate requests and forged requests

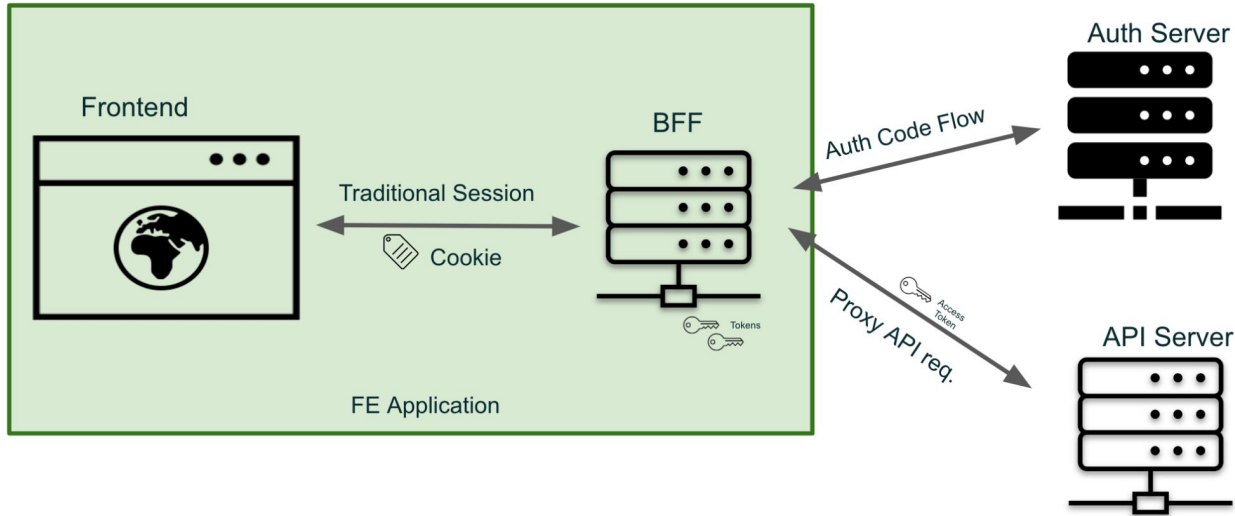




# Real life examples

- **ING direct (2008)**: allowed illicit money transfers
- **Paypal (2016)**: attacker can change a users profile without permission ([more](#))
- **Moodle (2020)**: A popular open-source Learning Management System, was found to have a CSRF vulnerability that could allow an attacker to change a user's password.
- **Fortinet FortiOS (2021)**: A CSRF vulnerability was discovered that could allow an attacker to perform administrative operations without the user's consent.
- **OkCupid (2021)**: Vulnerability in dating site OkCupid could be used to trick users into 'liking' or messaging other profiles ([more](#))
- **Grafana (2022)**: Vulnerability which allows anonymous attackers to elevate their privileges ([more](#))

# Remember our Backend for Frontend (BFF)



Cookies are used to handle the session management, so inherently a naive BFF is vulnerable to CSRF attacks.

# How can we stop CSRF attacks



# Defences for CSRF

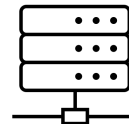
- Logging off web applications when not in use (logout after inactivity)
- Traditional Applications (php, webforms, mvc)
  - Synchronizer Token Pattern
  - Double-Submit Cookie Pattern
- SameSite Cookie
- SOP & Cross-Origin Resource Sharing (CORS)

# CSRF Defences



**Synchronizer Token**

# CSRF Defence - Synchronizer Token



Login

Cookie + CSRF Token

```
<form action="/email/change" method="post">  
  <input type="hidden" name="csrf_token" value="OWY4Nm...AS==">  
  <input type="text" name="email">
```

Send legitimate post request

```
POST /change/email HTTP/1.1  
Cookie: session=yvthwsztye....NJNAJ  
  
email=peter@mysite.com&csrf_token=OWY4Nm...AS==
```

# Preventing CSRF by Synchronizer token



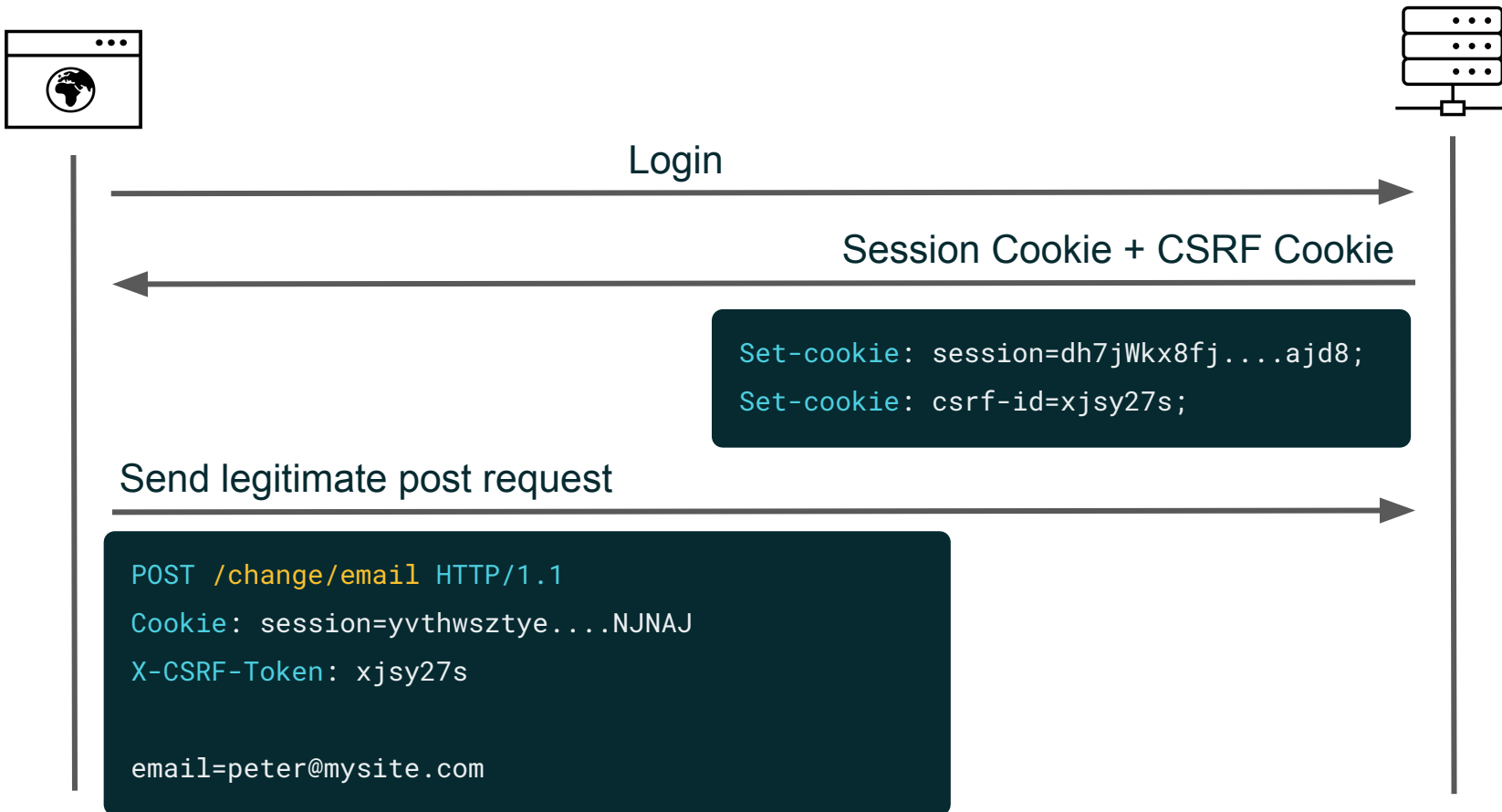
# CSRF Defences



**Double-Submit  
Cookie**



# CSRF Defence - Double-Submit Cookie



# Preventing CSRF by Double-Submit Cookie



# CSRF Defences

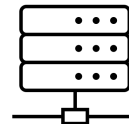


**SameSite  
Cookie**

# CSRF Defence - SameSite cookie



https://mysite.com



https://mysite.com

Login

Response + Cookie

```
Cookie: session=yvthwsztye....NJNAJ; SameSite=lax
```

Send legitimate post request

```
POST /change/email HTTP/1.1
```

```
Cookie: session=yvthwsztye....NJNAJ
```

```
email=peter@mysite.com
```

Cookie is marked SameSide, so it is only send back on https://mysite.com

# CSRF Defence - SameSite cookie

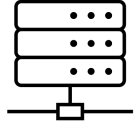


https://evil.hacker.com

A forged post request

```
POST /change/email HTTP/1.1
```

```
email=pwned@evil-user.ru
```



https://mysite.com



No cookie; Blocked by  
server

# SameSite cookie support

'SameSite' cookie attribute - OTHER

Usage  
Global

Same-site cookies ("First-Party-Only" or "First-Party") allow servers to mitigate the risk of CSRF and information leakage attacks by asserting that a particular cookie should only be sent with requests initiated from the same registrable domain.

Current aligned Usage relative Date relative Filtered All

Chrome	Edge *	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile *	UC Browser for Android	Android Browser *
	12-15	3.1-11.1										
4-50	16-17	12-13.1		10-38			3.2-11.4					
51-79	18-85	14-14.1	2-59	39-70			12-12.5	4				
80-121	86-121	15-17.3	60-122	71-105	6-10		13-17.3	5-22		12-12.1		2.1-4.4.4
122	122	17.4	123	106	11	122	17.4	23	all	73	15.5	122
123-125		TP	124-126									

Notes Test on a real browser Known issues (2) Resources (11) Feedback

This feature is backwards compatible. Browsers not supporting this feature will simply use the cookie as a regular cookie. There is no need

1 Not shipped with the initial release but later with the 2018 June security update (Patch Tuesday) to Windows 10 RS3 (2017 Fall Creators U

2 Partial support because only supported in IE 11 on Windows 10 RS3 (2017 Fall Creators Update) and newer, but not in IE 11 on other Win

3 Cookies without `SameSite` are treated as `Lax` by default, `SameSite=None` cookies without `Secure` are rejected.

4 Partial due to the lack of support in macOS before 10.14 Mojave.

5 Partial due to [the bug](#) that treats `SameSite=None` and invalid values as `Strict` in macOS before 10.15 Catalina and in iOS before 13.

# Preventing CSRF by SameSite Cookie



# CSRF Defences

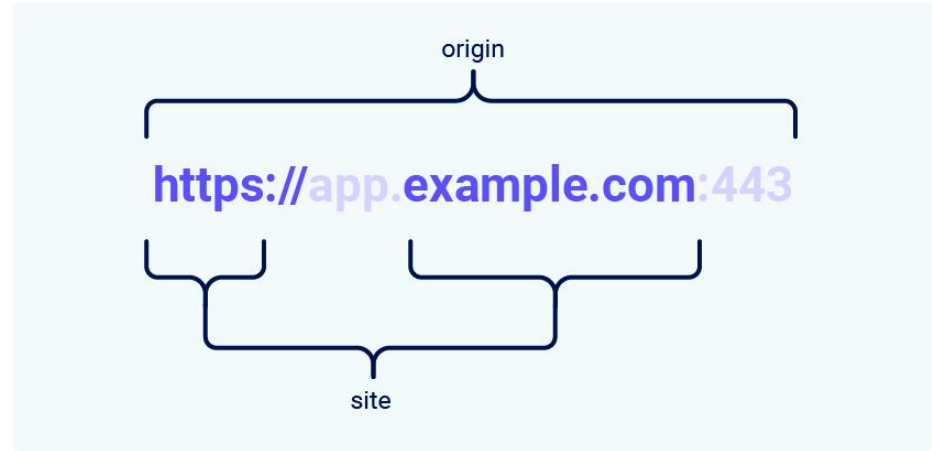


**CORS & Preflight**

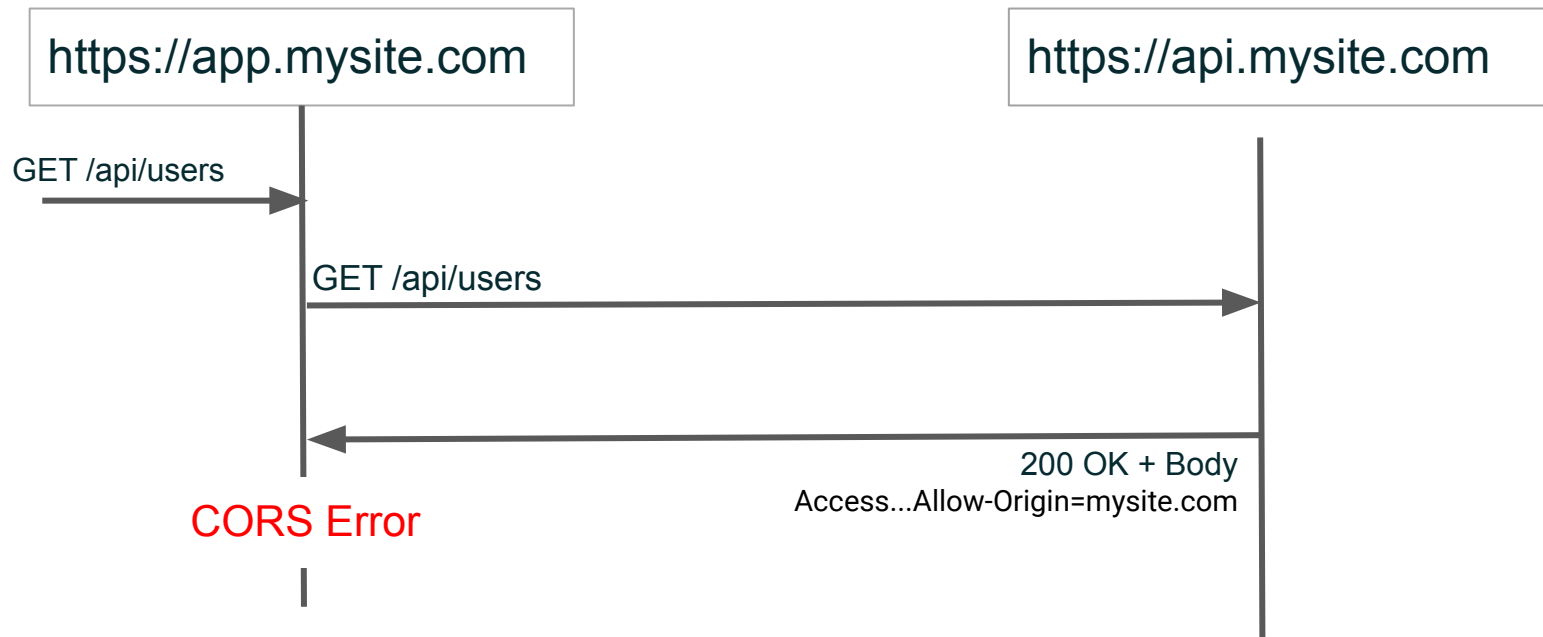


# Origin, SOP & CORS

- **Origin:** Web content's origin is defined by the scheme (protocol), hostname (domain), and port of the URL used to access it.
- **SOP:** Same Origin Policy : SOP restricts web resources from being accessed across different origins
- **CORS:** Cross Origin Resources Sharing : Allows web resources to be accessed across different origins with appropriate permissions

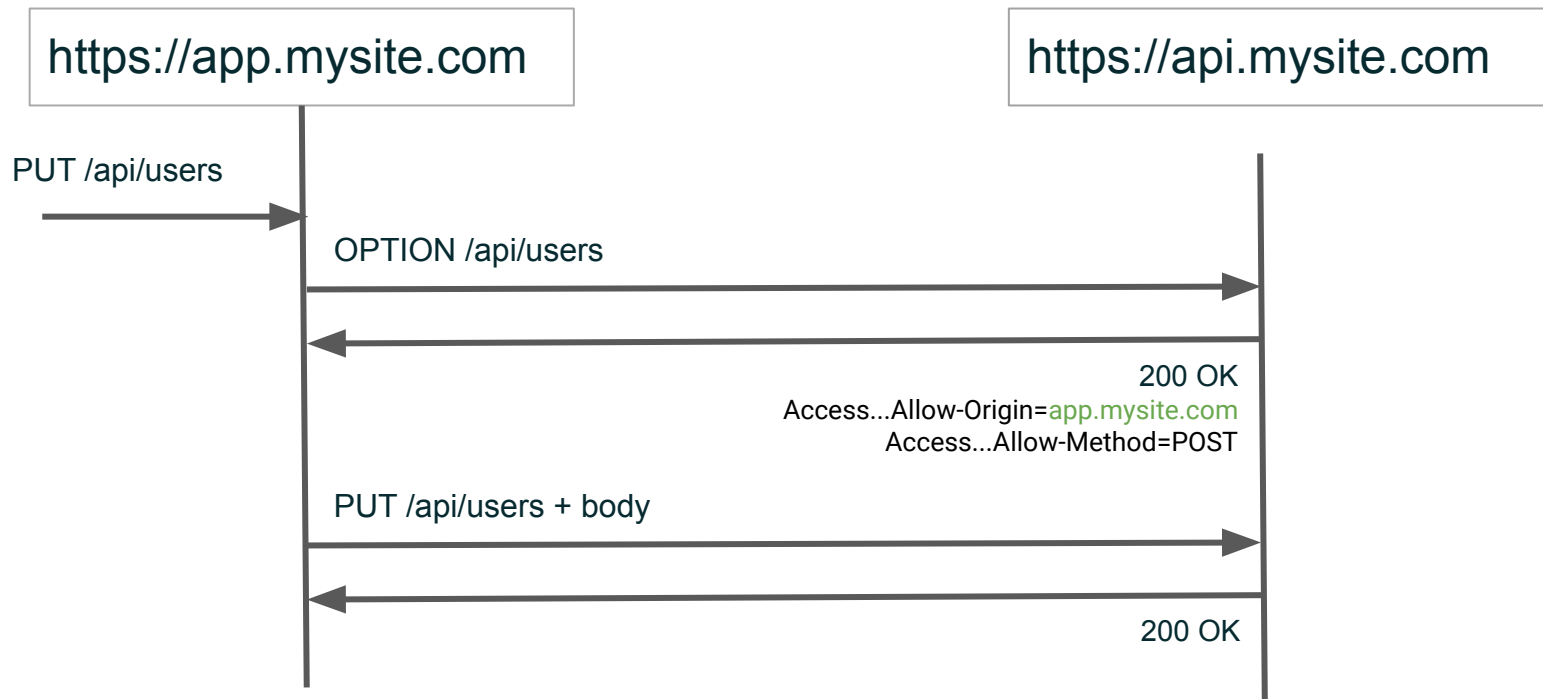


# Simple Request (GET, HEAD, or sometime POST)



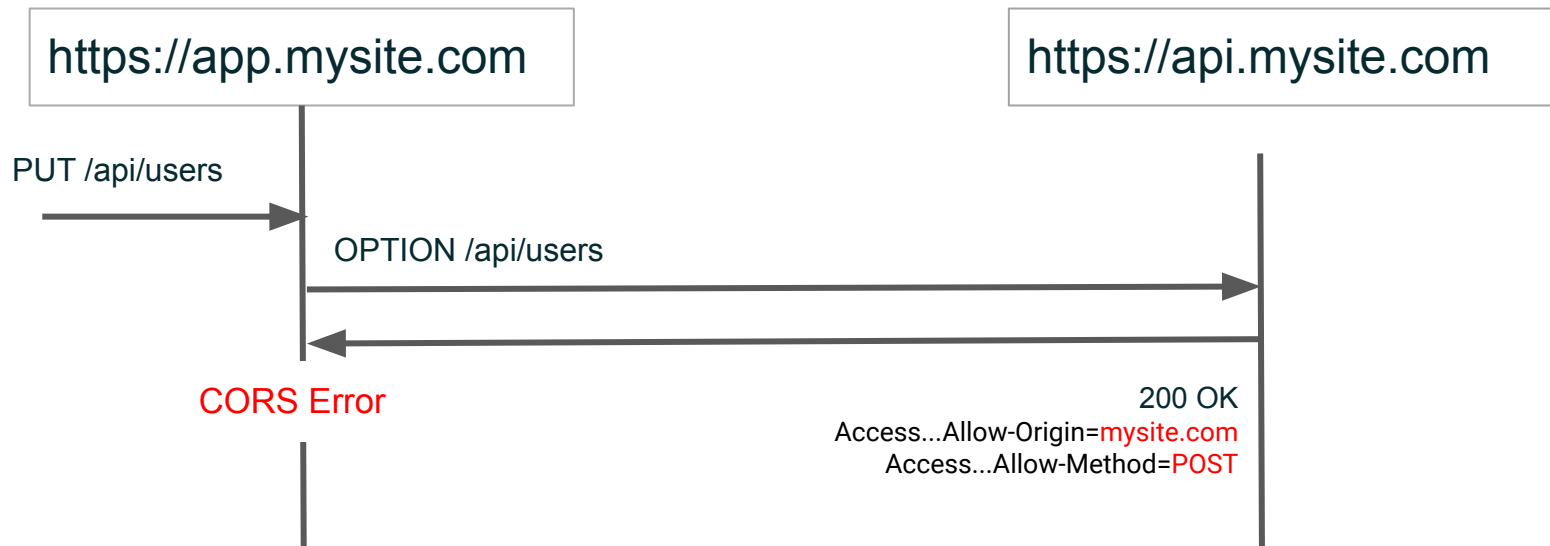
SOP doesn't block simple requests (GET, HEAD) but the browser reject reading of the response body.

# None Simple Request (PUT, DELETE, ...)



The browser sends a preflight request to understand what (non-simple/unsafe) requests the source allows. If the server allows it the browser continues with the actual request.

# None Simple Request (PUT, DELETE, ...)



Name	Status	Type
<input type="checkbox"/> content	CORS error	xhr
<input type="checkbox"/> content	200	preflight

The browser sends a preflight request to understand what (non-simple/unsafe) requests the source allows.

# Preventing CSRF by Same Origin Policy (SOP)



# Mitigation for CSRF

- **Synchronizer Tokens** and **Double Submit Cookie** are a good CSRF defence. By requiring the browser to submit a secret token along with the requested data the backend can identify and reject illegal requests.
- **Same-site** cookies neutralize CSRF. SiteSite cookies are not included in cross-site requests. But they do not protect against Cross Origin Request Forgery.
- If you set your **CORS** policy to only accept requests from trusted domains, it can prevent CSRF attacks originating from malicious websites.

So are we safe now?

Not yet, sorry



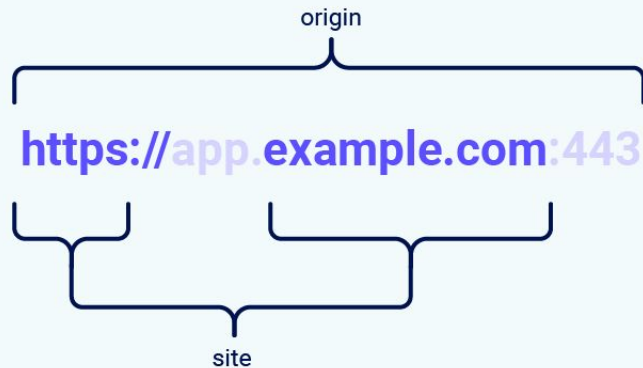
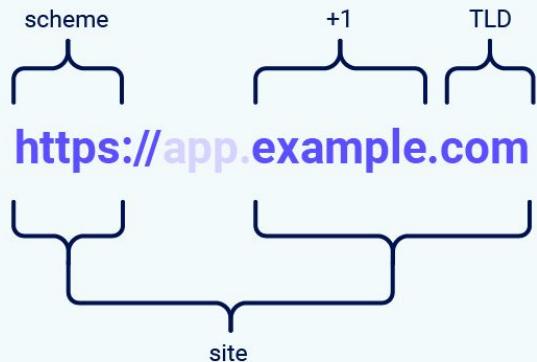
# CSRF Defences



**Block FORM  
posting  
&  
Strict  
Content-type**



# Same Site cs Same Origin



The **SameSite cookie** restrictions relates to the **site**. Multiple sub domains are considered as the same site.

Do we trust all subdomains? 🤔

# **Same Site != Same Origin**

## **Is SOP not handling this?**

Yes, Same Origin Policy (SOP) is blocking all Cross Origin resource requests.

But ...

# Submitting data FORM post

## Form POST Request

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=yvthwsztye

email=peter@mysite.com
```

## Empty POST Request

```
POST /passw/reset HTTP/1.1
Host: vulnerable-website.com
Cookie: session=yvthwsztye
```

SOP is not helping here 🤔

# Submitting data fetch - nocors

Fetch with mode="no-cors"

```
fetch('https://mysite.com/api/change', {  
  mode: 'no-cors'  
  method: 'POST',  
  headers: {  
    'Content-Type': 'application/json'  
  },  
  body: JSON.stringify({  
    email: 'peter@mysite.com'  
  })  
})
```

SOP is not helping here, either 🤔

# Improper Content-Type validation == a vulnerability

Accept Form URL Encoded values

```
const app = express();  
app.use(express.urlencoded());  
app.use(express.json());
```

Accept all content types

```
const app = express();  
app.use(express.json({ type: "*/*" }));
```

SOP is for scripts, so make sure you block form post 🤔

# Strict Content-Type verification



# CSRF Defences



**CORS  
&  
Custom Header**

# An empty POST request is a simple request!

Empty POST Request

```
POST /passwd/reset HTTP/1.1  
Host: vulnerable-website.com  
Cookie: session=yvthwsztye
```

An empty post is simple request, so no preflight.





# Block all simple requests



# Enforcing CORS (preflight) for all API routes

```
fetch('https://mysite.com/passw/reset', {  
  method: 'POST',  
  headers: {  
    'X-Cors': 1,  // <-- force preflight  
  },  
})
```

- Add a custom header (any header)
- This triggers a pre-flight for any cors origin request
- Enforce at the server for every endpoint

NextJS & the Duende BFF framework using this technique to protect local endpoints

So are we safe now?



Yes, thats it



# Key takeaways



# Key takeaways

CSRF Matters when you rely on cookies for authentication

SameSite cookie mitigate CSRF, but not Cross **Origin**

Request Forgery (CORF)

API can rely on CORS as a defence against Cross Origin

Request Forgery

# Further reading

- [Robust defenses for cross-site request forgery.](#)
- [Portswigger - Cross-site request forgery \(CSRF\)](#)
- [HackTricks - Cross Site Request Forgery](#)
- [NCC group - Common CSRF prevention misconceptions](#)
- [The Past, Present, and Future of Cross-Site/Cross-Origin Request Forgery by Dr Philippe De Ryck](#)
- [Comparing the BFF Security architecture with an SPA using a public API by damienbod.](#)
- [Common no-cors misconceptions](#)