Università
Ca’Foscari
Venezia

Master’s Degree programme — Second Cycle
(D.M. 270/2004)

in Informatica — Computer Science

Final Thesis

—

Ca’ Foscari
Dorsoduro 3246
30123 Venezia

Client-Side Security Using CORS

Supervisor
Prof. Focardi Riccardo

Candidate
Mohamed Abdelhamied
Matriculation number 849656

Academic Year
2014/201

Abstract:

Nowadays web security is an important issue, as everyone uses computers to access different websites such as news, social networks, bank accounts, etc. It becomes crucial to keep sensitive information, cookies and passwords, protected against any kind of web threat including malware and web-specific attack patterns such as Cross-Site Request Forgery (CSRF) or Cross-Site Scripting (XSS). There are different mechanisms for preventing or mitigating this type of attacks, among which the standard *same-origin policy* of browsers that limits accesses performed by scripts executed in a web page to other web pages that share the same origin. In practice, however, web application might need to access other origins through JavaScript. For this reason, W3C has recently introduced a new mechanisms called Cross-Origin Resource Sharing (CORS) that permits a controlled access to cross-origin web pages. In this thesis, we illustrate CORS and investigate its main advantages with respect to the standard same-origin policy. To support our study we have performed a number of experiments that point out the flexibility and potential of CORS when applied to typical web-application use cases.

Acknowledgments

First of all, I would like to thank and express my deepest appreciation to my supervisor Professor Focardi Riccardo, who is really rich of ideas and possesses a very good personality. I am sure that without his systematic supervision, positive approach and guidance this work would not have been possible and also I want to thank him for his significant help in curving my future career path in the field of Computer Security. I would also like to give my thanks and appreciation to the Informatics Department of Ca' Foscari for providing a stimulating environment full of passionate people.

I would also like to thank Stefano Calzavara, for his support during my work in my thesis and my best friend Phu Nguyen who has always patiently helped me during my Master degree study providing the best solutions to my problems. I would also like to extend my gratitude to Tinsu Gare, Kameran Lak, and Federico Pistone, for covering my living expenses during the early months of my study. I would also like to thank my friends Ismail Elezi, Alberto Meggiato, Phùng Kiên, and Andrea Lazzarotto, for helping me in the study during my master, in addition I'd like to thank Miriam Pagano and Mereb Habte for editing the thesis and language correction.

And last but not least, I would like to thank my family and friends: my father, Abdelhamied Hassan, my lovely mother Zahira Sadek, my brothers Khaled and Azat and sisters Rasha and Marwa, who are always happy for the good things happening to me (more than I am for myself) and always caring for me, supporting and having my back.

Thank you Allah for being with me in my life!!!

Table of Contents

	Abstract	ii
	Acknowledgments	iii
	List of Figures	vii
	List of Tables	x
1	Introduction	1
2	Background Concept	4
	2.1 Communication Mechanisms	6
	2.1.1 Same-Origin Policy	6
	2.1.2 Cross-Site-Request	11
	2.2 Security Attacks	13
	2.2.1 Cross-site scripting (XSS)	13
	2.2.1.1 Stored XSS	13

	2.2.1.2 Reflected XSS	14
	2.2.2 Cross-site request forgery (CSRF)	16
3	Cross-Origin-Resource-Sharing (CORS)	18
	3.1 Notion of CORS	19
	3.1.1 The Client	20
	3.1.2 The browser	20
	3.1.3 The Server	21
	3.2 Simple Request	21
	3.3 Preflight Request	23
	3.4 Advanced Request	25
4	Our Contribution	27
	4.1 Simple CORS Request	30
	4.1.1. Allowed Origin request	30

4.1.2 Wild-Card request 34

4.2 Preflight Request 36

4.3 Advanced Request 46

5 Conclusion and Future work 54

5.1 Review 54

5.2 Conclusion 55

Bibliography 57

List of Figures

2.1	Client-browser-server	5
2.2	Domain A and B	5
2.3	Origin	7
2.4	Same Origin	7
2.5	Different Origin	8
2.6	Cross Site Request	11
2.7	XHR request	12
2.8	Stored XSS Attack	14
2.9	Reflected XSS Attack	15
2.10	CSRF Attack	16
3.1	CORS request	19
3.2	CORS simple request based on allowed access origin	22
3.3	CORS simple request using wildcard	23
3.4	CORS preflight-request	24
3.5	CORS advanced request based using credentials header	26
4.1	Server Configuration based on allowed origin	30

4.2	Clients Configuration	31
4.3	CORS Response of Client in origin 1112 using Network tab	32
4.4	CORS Response of Client in origin 1112 using Console tab	33
4.5	CORS Response of Client in origin 1113 using Network tab	33
4.6	CORS Response of Client in origin 1113 using Console tab	34
4.7	Server Configuration using wildcard	35
4.8	CORS Response of Client in origin 1113 using Network tab	35
4.9	Server Configuration for Preflight response	37
4.10	Client Code in origin 1111	38
4.11	Client in origin 1111 preflight request	39
4.12	Actual request using PUT method	40
4.13	Cashed Preflight response	41
4.14	Client Code in origin 1113	42
4.15	Client in origin 1113 preflight request	43
4.16	Actual request using POST method and custom header X-CaFoscari DAIS	44
4.17	Rejecting Preflight-request using method TEST	45
4.18	Server Configuration for Advanced Request	47
4.19	Login Page code of origin 1111	48

4.20	CORS Request for origin 1111 without credentials	49
4.21	Login page of origin 1111	50
4.22	Response of unauthenticated user in origin 1111	50
4.23	Login Page code of origin 1112	51
4.24	CORS Request for origin 1112 include credentials	52
4.25	Login page of origin 1112	52
4.26	Response of unauthenticated user in origin 1112	53

List of Tables

2.1 Browser determine the request if it same origin or no 10

4.1 Technical Description of clients and server 29

CHAPTER 1

Introduction

Nowadays web security is an important issue, as everyone uses computers to access different websites such as news, social networks, bank accounts, etc. It becomes crucial to keep sensitive information, cookies and passwords, protected against any kind of web threat including malware and web-specific attack patterns such as Cross-Site Request Forgery (CSRF) or Cross-Site Scripting (XSS). There are different mechanisms for preventing or mitigating this type of attacks.

The Internet is a huge network that gives clients the possibility to share information or exchange data among themselves. Users would not like to give unauthorized people the permission to access their own private data and so to manipulate them. This is one of the main relevant issues to be solved due to the rapid increase of techniques implemented for communications. A lot of researchers and scientist are trying to tackle this problem through different approaches and, at the same time, they are trying to make the security policy simple, efficient and easy for the users. In fact, another point which is also crucial is the trade-off between security policy and the server structure since increasing the strength of security policies typically make applications more complex and hard to maintain. One of the solution to this problem is using a mechanism that can be responsible of the security policy and at the same time does not affect the structure of the server.

In this thesis we investigate the *Cross-Origin-Resource Sharing* (CORS) mechanism. The advantage of this mechanism is to make cross site requests possible between two different origins, unlike the standard *Same Origin Policy* (SOP) of browser, which is more restricted. In additions, it offers a security policy for accessing the resources based both on credentials and on access origin.

The thesis is organized as follows. In the next chapter we will present the basic concept of Same Origin Policy, Cross site request and Client-Security and Browser Security Policy. Chapter 3 is about the very notion of CORS (Cross-Origin Resource Sharing) and it shows the difference between a simple request and a more advanced request and the acceptance or denial of this request. Chapter 4 presents the mechanism with some experimental results based on the type of the request the mechanism offer it. Finally, Chapter 5 concludes our work, and gives a guideline for web developer.

CHAPTER 2

Background Concept

In this chapter we will discuss the difference between two mechanisms for communication between sites. First, we will start by illustrate Same Origin Policy (SOP) mechanism, by showing how two sites in same origin communicate, then we will show the second mechanism Cross-Site- Request, and we will illustrate, what will happen if two sites in different origin want to exchange information such as (web-fonts, stylesheet, etc.). Later in this chapter we will show two different kind of possible attacks and illustrate how they are work. But before we start to speak about that, I'd like first to speak about how two site exchange information. This process of exchanging information usually occurs in 3 parties: Client, Server, and Browser.

Every element of this process has its own role in making this process successful. Figure (2.1) shows how those elements interact during the process.

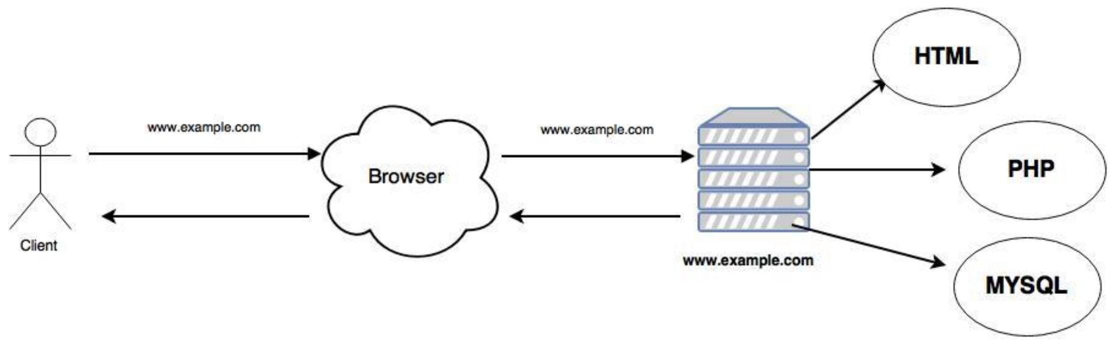


Figure 2.1: Client-server-browser

The client wants to access the website (www.example.com). In order to do that he sends a request to the browser and the browser checks the website and sends the response back to the client. This is what we do every day when accessing a website. The previous scenario is between Client and Server only, but what if two servers want to interact so to retrieve information? Figure (2.2) below shows what happens in reality when domain A wants to get information from domain B.

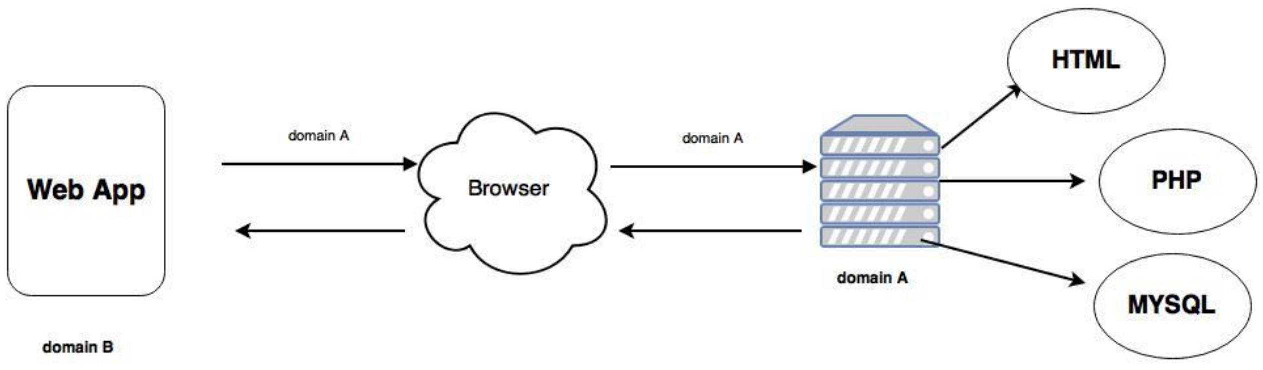


Figure 2.2: domain A and domain B

In this situation the previous scenario changes based on two conditions:

1- Same-origin Policy.

2- Cross-site request.

2.1 Communication Mechanism:

2.1.1 Same-Origin Policy:

The concept of Same-Origin Policy[1], is an important notion from the security point of view. Generally, it checks if the request between the two domains have the same origin or not; then, the origin is defined by W3C [1] as a combination between three parameters:

1- URL Scheme.

2- Host name.

3- Port number.

The figure (2.3) below shows the combination of the three parameters.

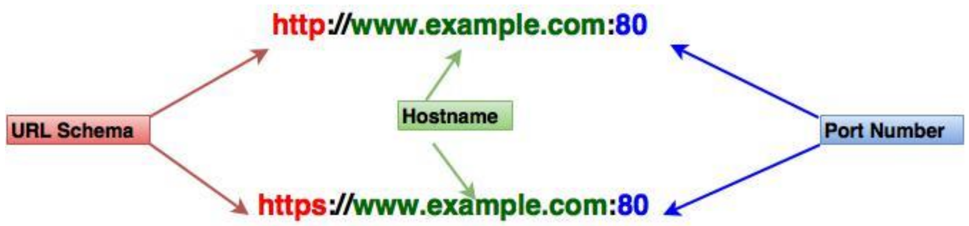


Figure 2.3: Origin

Each parameter has its own relevance. Let’s be back at the example of two domains, in order to understand in which way each parameter is decisive to make the browser consider the request as having the same origin or not.

In the previous example, we have a web app and the server in figure (2.4)

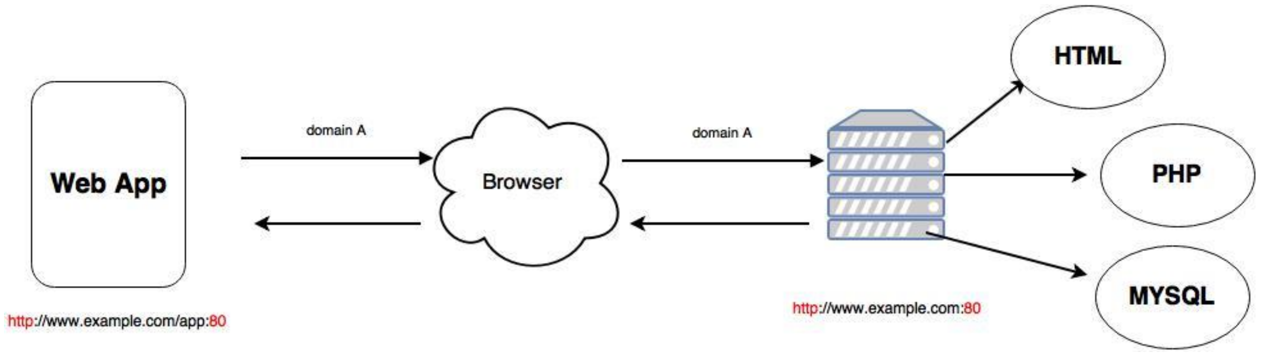


Figure 2.4: Same origin

We can see that both the web app and the server have the same origin, because, based on the previous definition [1], they have the same URL scheme, Hostname, and port number. As we can see in the figure (2.4), the URL scheme for both app and server is the same (**http**), plus they have the same hostname (**example.com**) and port number (**80**). In this situation the Browser, the second member of the process, can easily recognize that they share the same origin, as mentioned in the definition [1] of the origin above, and so it allows the request between the client (app) and the server. Now, let's examine what happens in case the origin of app is different from the origin of server.

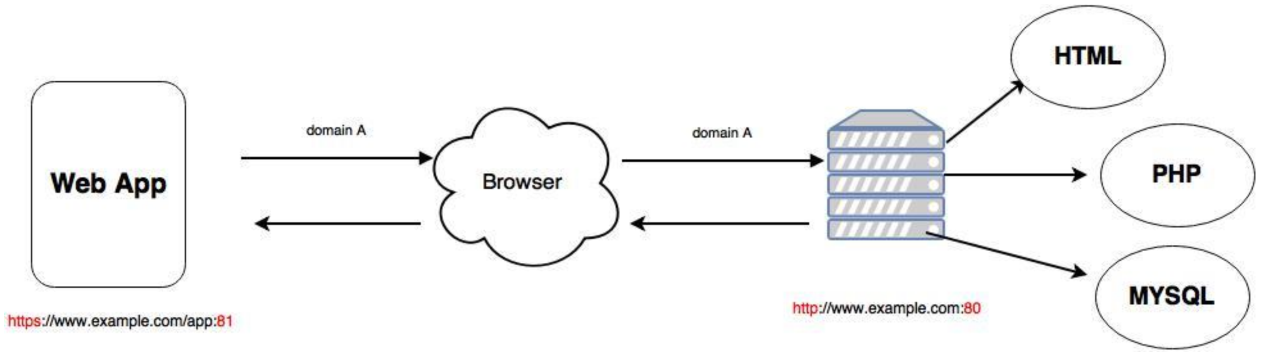


Figure 2.5: Different Origin

As shown in figure (2.5) above, the request is rejected from the browser, because the app and the server have different origins.

Even if they have the same hostname (**example.com**), yet the origin of the app has a different URL scheme (**https**) from the server (**http**) and also a different port number (**81**) in the app and (**80**) in the server, so the request is denied because the browser treats the app and the server not having the Same Origin. Thus, in brief, we can say that the role played by the browser in responding to the request consists mainly in checking the origin for both the app and the server. Anyway, at this point, we still need to go a step further and investigate how the browser can determine if the request is coming from the same origin or not and why all this happens.

To examine the first part of the previous consideration, Table (2.1) illustrates how the browser works out the same-origin request when the app makes a request to the server by sending it to http://www.example.com:80

Incoming request	**Result**	**Reason**
http://www.example.com/app1:80	**Success**	Same origin
http://www.example.com/app2:80	**Success**	Same origin
https://www.example.com/app3:80	**Failed**	Different URL Scheme
http://www.example.com/app4:81	**Failed**	Different Port Number
http://www.example.it/app:80	**Failed**	Different Host Name

Table 2.1: Browser determine the request if it same origin or no

With regard to the reason why the browser prevents incoming requests from different Origin [1], we can say that in doing so the browser only permits the access to apps and scripts, in order to prevent any kind of attacks, such as cross-site request forgery (CSRF) or Cross-site scripting (XSS).

2.1.2 Cross-Site-Request:

Considering again our example of domain A and domain B. Based on the previous scenario, if they want to communicate, they cannot, because each domain lives in different origin. This is the so called “Cross-site request”. The concept of “Cross-site request” has become an important issue for big companies and web developers, who are now advancing in coming up with new methods to be able to exchange and make requests from different origins, by using XMLHttpRequest (XHR) object. This method gives the developers the possibility to retrieve information from different origins, as shown in figure (2.6)

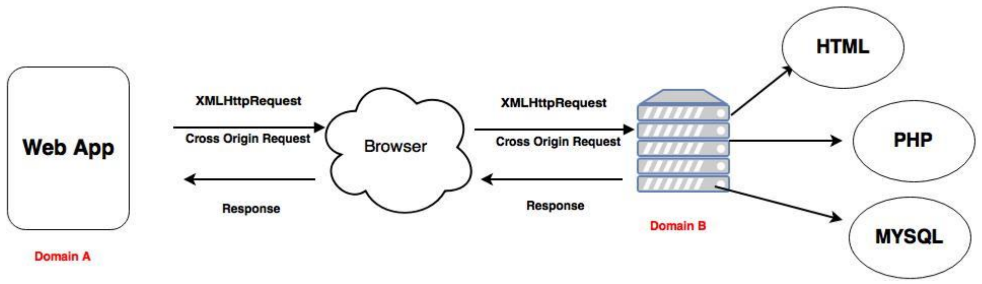
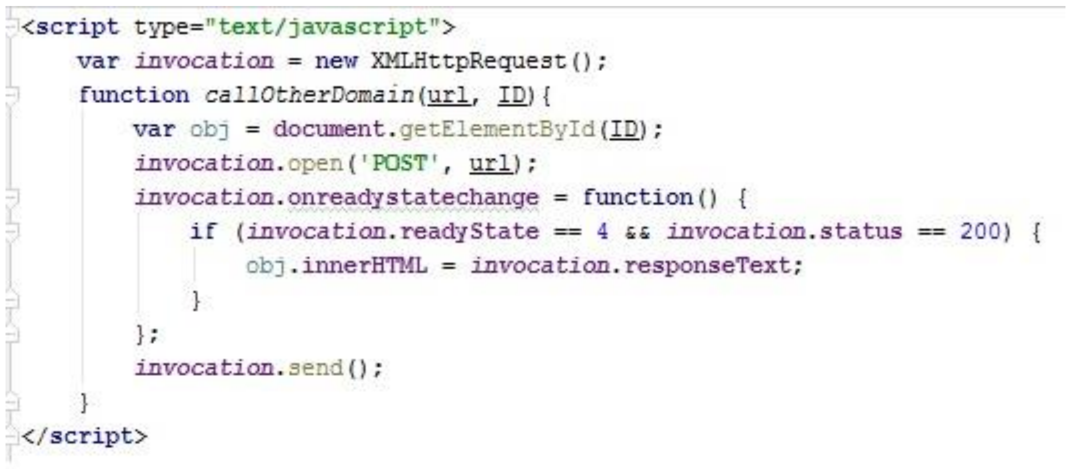


Figure 2.6: Cross-Site request

The figure (2.6) above shows what is called “cross-site request”. Web App in domain A wants to get some information or data from server in domain B such as (HTML page, web-font, stylesheet script, etc). Figure (2.7) shows a simple request using XMLHttpRequest (XHR)



```
<script type="text/javascript">
    var invocation = new XMLHttpRequest();
    function callOtherDomain(url, ID){
        var obj = document.getElementById(ID);
        invocation.open('POST', url);
        invocation.onreadystatechange = function() {
            if (invocation.readyState == 4 && invocation.status == 200) {
                obj.innerHTML = invocation.responseText;
            }
        };
        invocation.send();
    }
</script>
```

Figure 2.7: XHR Request

But from a security point of view it can lead to Cross-site scripting (XSS) or Cross-site request forgery (CSRF) vulnerabilities. This happens because we cannot trust any request from different domains, since it may be sending malicious script to leak important information or sensitive data, such as cookies or passwords.

2.2 Security Attacks:

2.2.1 Cross-site scripting (XSS):

According to the previous scenario, Cross-site scripting is considered to be one of the most important issues related to security. In our work, XSS vulnerability is a very dangerous attack [2] [3], as it gives the attacker the permission to inject malicious script (e.g., HTML, JavaScript) into web applications, and doing so to leak sensitive information from the user, such as cookies and passwords. Once the attacker retrieves sensitive data, he can use them at will. There are two well-known kinds of XXS vulnerability: Stored XSS and Reflected XSS.

2.2.1.1 Stored XSS:

From the name of this attack [2] [3], we can recognize that this kind of attacks happen when the attacker store (inject) his malicious code permanently on the server. Once the user visit an infected page, the malicious script is automatically executed, as shown in figure (2.8).

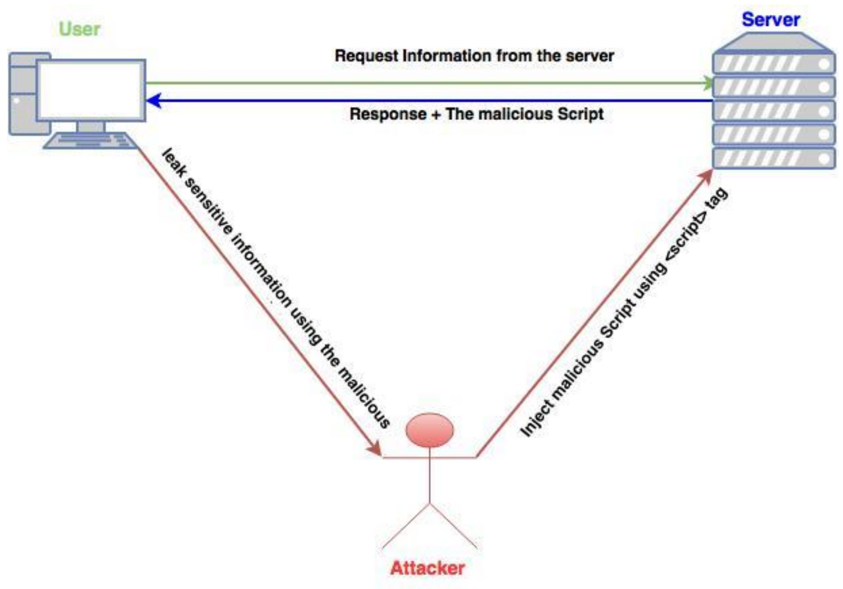

User
Server
Request Information from the server
Response + The malicious Script
leak sensitive information using the malicious
Inject malicious Script using <script> tag
Attacker

Figure 2.8: Stored XSS attack

In the figure we can see that the attacker inject his malicious code into the server, and that once the user request data from it and visit the infected page, not only the server will respond with the requested data, but also the malicious script will be executed. This explains how the attacker gains access to sensitive data.

2.2.1.2 Reflected XSS:

This kind of attack [2] [3] is different from the previous one. Here, instead of storing the malicious code into the server, the attacker deals with the client by sending an emergency e-mail or by creating an URL containing a malicious script that will steal sensitive information, such as bank or social network account, once the client clicks on it. As the user ties to access the link given by the attacker, which is a vulnerable website, the attacker gets the victim data and can have complete access to his account, as shown in figure (2.9).

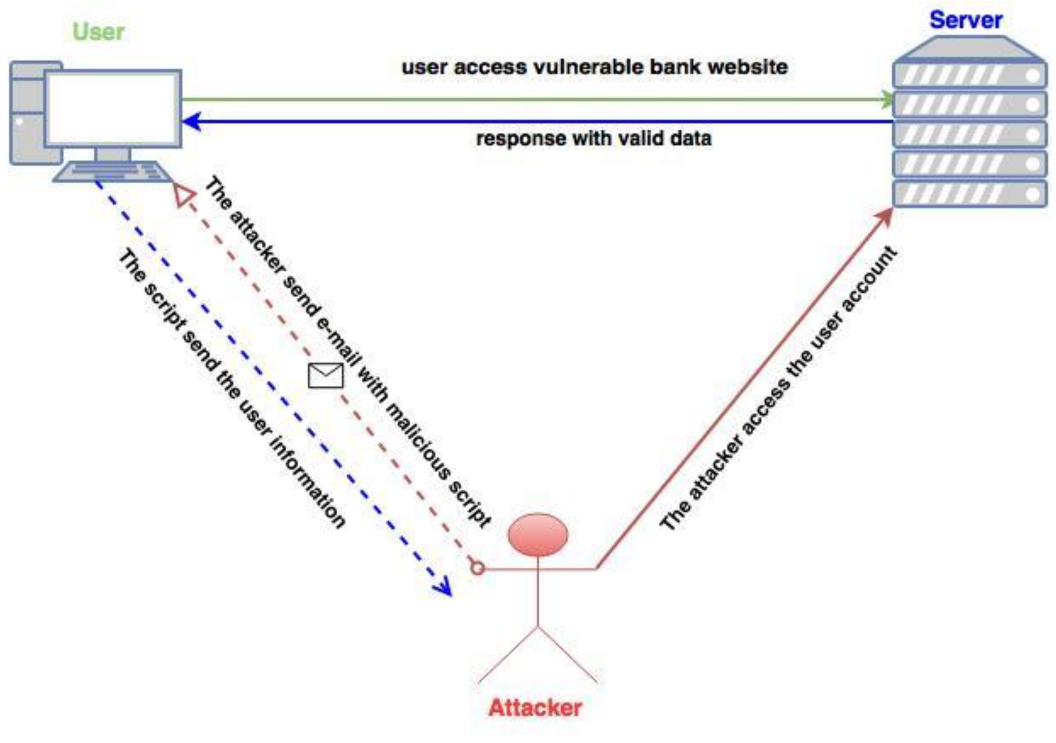

User
Server
user access vulnerable bank website
response with valid data
The attacker send e-mail with malicious script
The script send the user information
The attacker access the user account
Attacker

Figure 2.9: Reflected XSS attack

2.2.2 Cross-Site Request Forgery (CSRF):

Different from the previous attack, the CSRF attack [4] [5] occurs when a user accesses a vulnerable web app and creates a session. In doing so he opens a malicious site in another browser tab. The malicious site first steals the user's session cookies and then it makes a request to the vulnerable app by using the user's cookies within the session window. In this way the web app is not able to distinguish if the request is coming from an authenticated user or from the attacker. Figure (2.10) shows an example of this kind of attack.

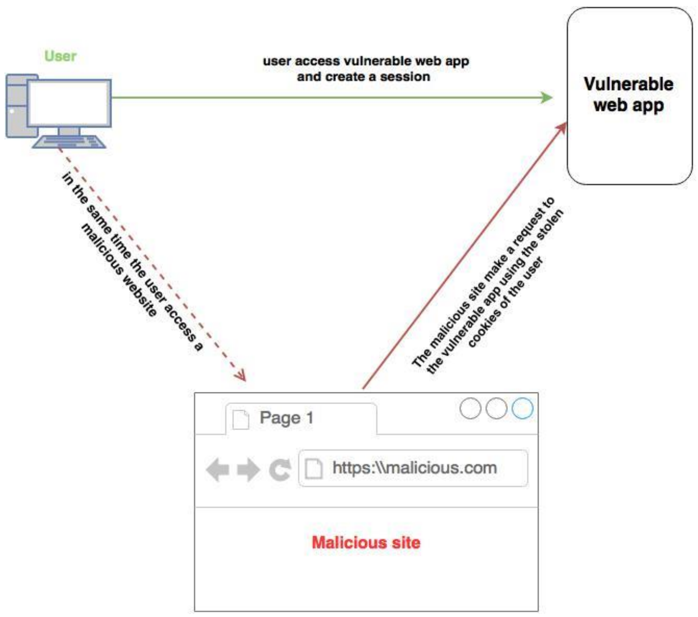

User
user access vulnerable web app and create a session
Vulnerable web app
in the same time the user access a malicious website
The malicious site make a request to the vulnerable app using the stolen cookies of the user
Page 1
https:\\malicious.com
Malicious site

Figure 2.10: CSRF attack

We can see how the attacker accesses the vulnerable app through the use of cookies stolen from the user.

CHAPTER 3

Cross-origin resource sharing (CORS)

In this chapter we will discuss the standard mechanism established by W3C [6] that is Cross-Origin resource sharing (CORS), and how it works.

We have seen so far that, it might be hard for two sites hosted in different origins to communicate or retrieve information, because of the same origin policy. Furthermore, to prevent (attacker) from stealing sensitive data through a malicious script, W3C are working on a new mechanism which gives the opportunity to make requests from different origins and at the same time intensifies security policies. We will start this chapter discussing the notions of the mechanism. After that, we will show the different kinds of requests used by it.

3.1 Notion of CORS:

CORS [6] [7] is a mechanism that allows resources that are hosted in different origins to make successful requests. The mechanism is based on XMLHttpRequest. A simple example can be seen in figure (3.1).

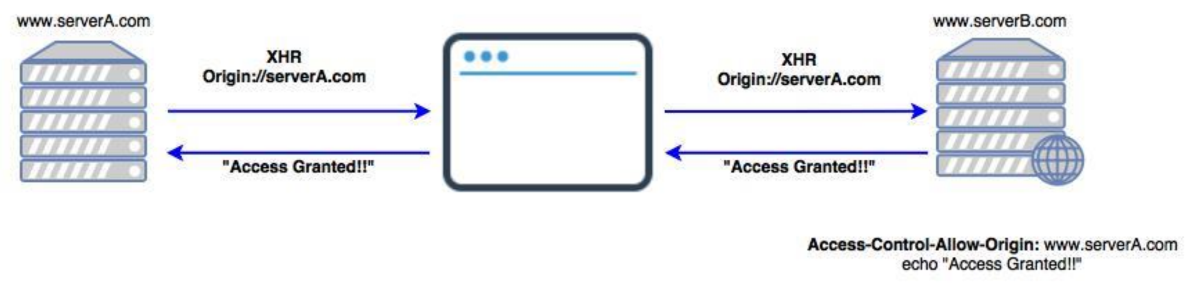


Figure 3.1: CORS request

Figure (3.1) presents how server A make a CORS request using XHR to retrieve information from server B. Still including its origin (serverA.com), server B checks if the header origin matches with the allowed origin or not. In our case, it does match and so it gives server A the permission to access.

In order to understand better how CORS works, we first need to know the role of each member of the CORS request. In figure (3.1), we can see that there are three members participating in the request: the Server, the Browser and the Client.

3.1.1 The client:

The client role is to send the XHR, including its origin and the method of the request. Later in this chapter we will understand the importance of mentioning the method in the request.

3.1.2 The Browser:

The browser is responsible for transmitting the request from the client to the server and then send back the response. In the meanwhile, the browser checks if the origin of the client has the permission to access the data in the server. In case there is no match, the browser does not send back the response to the client and it will show an error message. We will address the different types of error messages in the following chapters.

3.1.3 The server:

The server is where the CORS mechanism is established by setting up the Allow origin access list and the allowed method. If the request matches the allow origin and the request method, the server responds to the request with the right data; if this is not the case, the server responds with an error message. Now, we will deepen the different kinds of CORS requests.

3.2 Simple Request:

The simple request [6] [7] follows two methods: The first one is using allowed access origin defined by the server, the second one based on wild card which give the permission to any origin to access. We will see what happens in case of accepting or rejecting the request.

First, let’s start by the analysis of a simple request, using the allowed access origin. Figure (3.2) illustrates how this type of request, based on the allowed access origin, works.

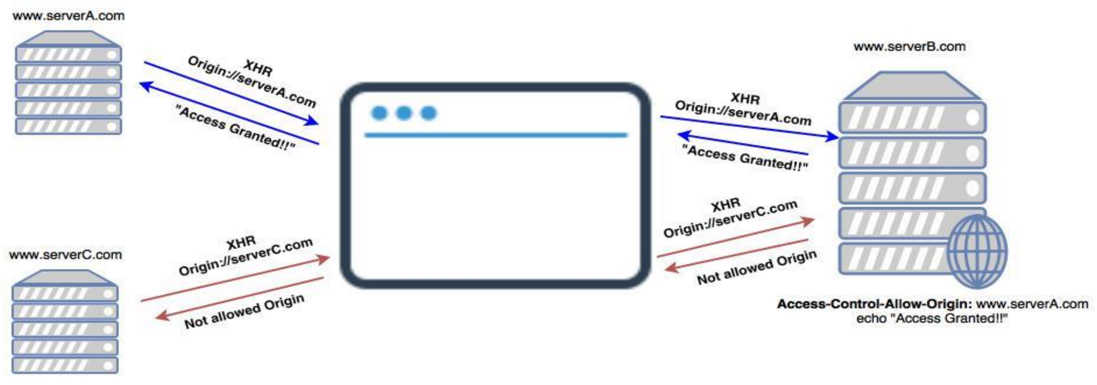


Figure 3.2: CORS simple request based on allowed access origin

The figure (3.2) shows two different servers making a request: server A and server C. On the other side, we have server B, which only gives the permission to server A to make the request by using the header **Access-Control-Allow-Origin: www.serverA.com**. Under the previous conditions, the browser sends the response only to server A and gives an error message to server C. Since the origin is not defined in server B, the request fail.

When the simple request is based on the wild card [6] [7], it means that the server allows the access to any origin by using a header (**Access-Control-Allow-Origin: ***). Thus, any request from any origin will be allowed, so it shouldn't be used, unless there are no sensitive data that can be leaked. We can see how this scenario works in figure (3.3).

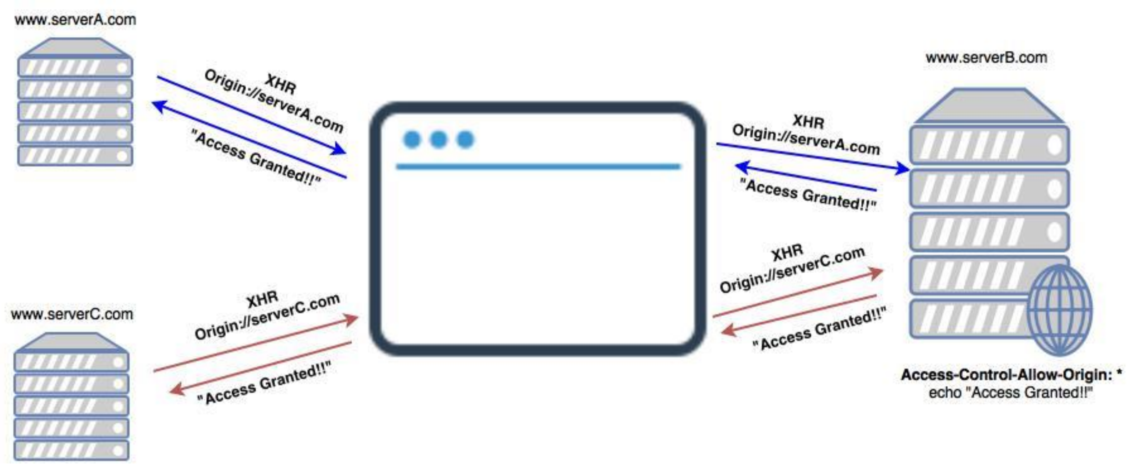


Figure 3.3: CORS simple request using wildcard

Server A and server C make a request to server B where Access-Control-Allow-Origin uses the wild card *. As W3C [6] mentions, using a wild card is not secure at all, since anyone can use it to their advantage and perform malicious attacks, like CSRF.

3.3 Preflight Request:

The main aim of the preflight request [6] [7] is to determine which are the possible headers, content types, and request methods allowed by the server. After that, if the preflight matches all the parameters defined by the server, then the client will send the actual request. Usually, The

HTTP standards request Methods is (GET, POST, HEAD). If the request method is different from the standard one (e.g, PUT or DELETE) , the browser recognizes it as OPTIONS requests that because the browser checks if those method is defined in the server or no if it is defined in the server then, it will send the actual request. In addition, in case we use the POST [7] method to make the request using different content type from the such as (XML payload to the server using application/xml, text/xml, or using custom headers), In this case the request is preflight. We can see different types of request in figure (3.4).

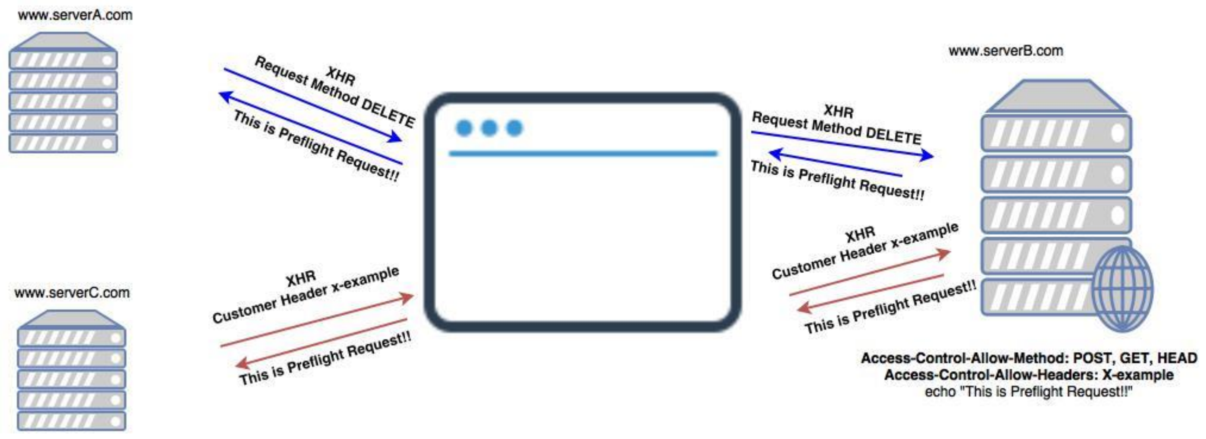

www.serverA.com
XHR
Request Method DELETE
This is Preflight Request!!
www.serverC.com
XHR
Customer Header x-example
This is Preflight Request!!
www.serverB.com
XHR
Request Method DELETE
This is Preflight Request!!
XHR
Customer Header x-example
This is Preflight Request!!
Access-Control-Allow-Method: POST, GET, HEAD
Access-Control-Allow-Headers: X-example
echo "This is Preflight Request!!"

Figure 3.4: CORS preflight-request

As we can see, we have two servers: A and C. As the allowed method defined by server B is POST, GET, HEAD and server A is trying to make a request using a different request Method (DELETE), it will lead to a Preflight request [6] [7]. For the same reason, server C will be considered a preflight-request too [6][7], as it is using a custom header x-example.

3.4 Advanced Request:

In this section we will discuss the Advanced CORS request [6] [7] which, differently from the **Access-Control-Allow-Origin** and the **Access-Control-Allow-Methods,** contains more parameters such as: including credential (cookies) for authenticate the client and making a request or Max-Age header [6] [7], which is used to indicate how long the result can be cashed before making a new preflight-request. We can see an example of it in figure (3.5).

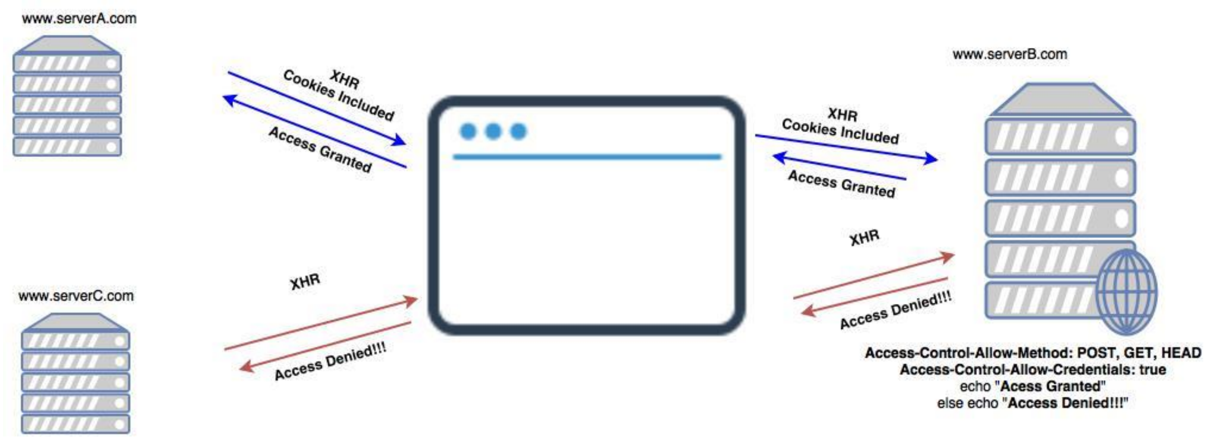


Figure 3.5: CORS advanced request based using credentials header

In this example we can see server A and server C making a request to server B; the server requires the Credentials to include in the request otherwise it will fail. Server A makes a request including the Credentials (cookies) and the response is “Access Granted”; on the contrary, server C makes the request without including the Credentials and the response received is “Access Denied”, as it doesn’t meet the requirements for the request to server B.

CHAPTER 4

Our Contribution

In this thesis we investigate the use of CORS for providing secure, cross-site interaction in web applications, Furthermore, in this chapter we will show different experiements using CORS mechanism in practical way, starting from the simple request, then the preflight request, and finally the advanced request. More specifically:

1. We devise a set of experiments to illustrate in details the various mechanisms implemented by CORS;
2. We consider typical web application use cases and show how CORS can be applied to provide secure cross-site interaction;
3. Based on our experiments, we discuss guidelines for web developers that intend to adopt CORS in their web applications and we draw some concluding remarks about the level of security achieved by the adoption of CORS.

Let’s answer now our question about how two different domains that live in different origin can interact and communicate in a safe way. Let’s recall our example of domain A and domain B mentioned in chapter 2, figure (2.6). In this example, we have made a request from domain A to domain B and to achieve our objective we have used CORS mechanism.

Now, we start by showing how we can make a simple request and how the server can respond accepting the request or rejecting it. Then, we explain the difference between using allowed origins and using the wild card as discussed in chapter 3.

Before we start speaking about our experiment, we first need to know how to establish the right environment to do our experiment. Let’s recall the main players of CORS request discussed in chapter 3:

1. The client: a simple page to retrieve information from the server page.
2. The browser: Google Chrome Version 48.0.2564.97 m
3. The server: we have created a localhost using Apache and then we have created a page from which the client can retrieve the data.

In order to show the difference between the response of the server in case of acceptance or denial of the request, we create two different clients. The Technical description of the clients and the server is shown in table (4.1).

URL	**Description**	**Origin**
Localhost:1111/test.php	Server	**1111**
Localhost:1112/test.html	Client	**1112**
Localhost:1113/test.html	Client	**1113**

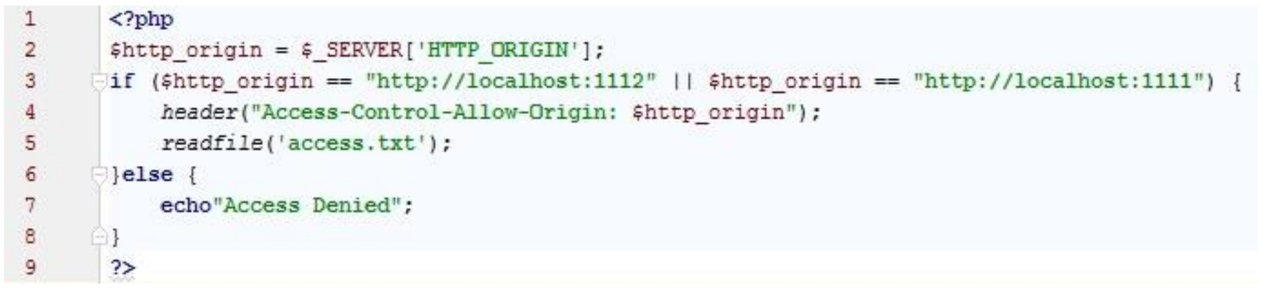
Table 4.1: Technical Description of clients and server

Considering that each site lives in different origins, based on the specified port as shown in the previous table.

4.1 Simple Request:

4.1.1 Allowed Origin request:

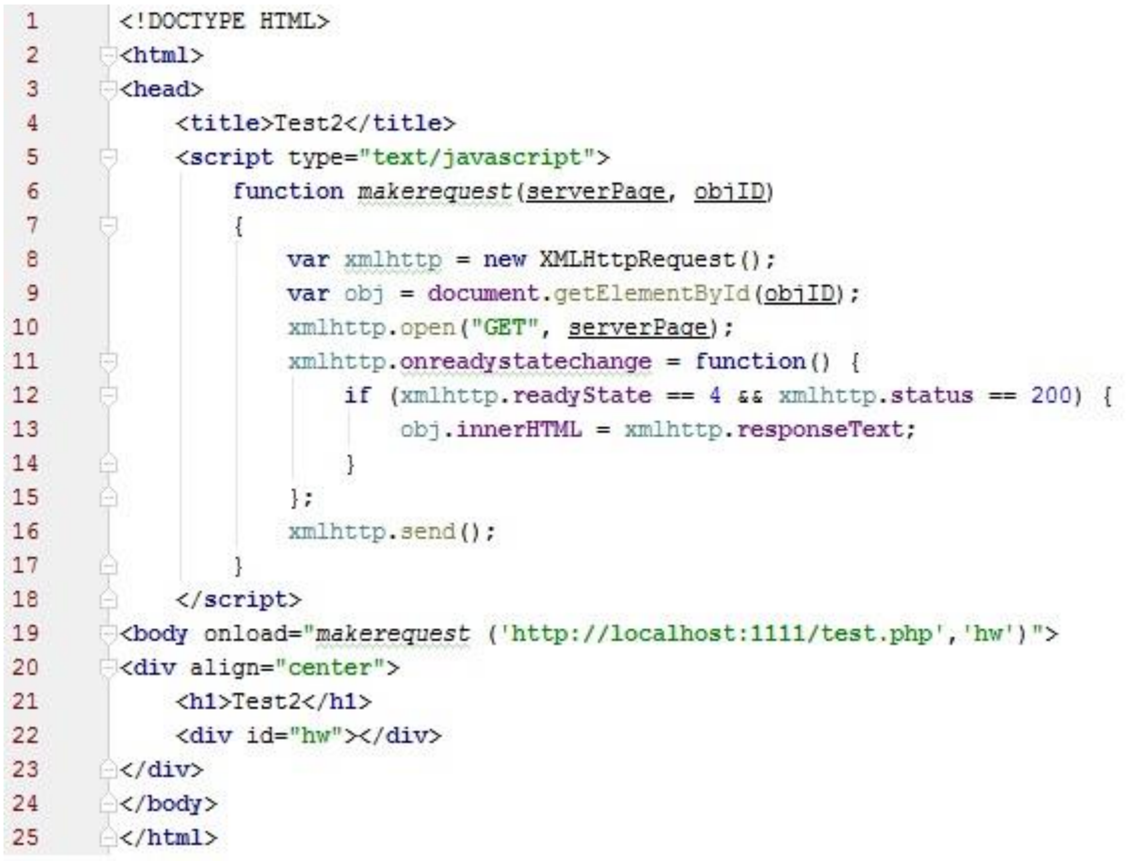
The request we make is based on Allowed origin. In this request, the server accepts requests from clients living only in origin 1112. The following PHP snippet shows the configuration of the server.



```php
<?php
$http_origin = $_SERVER['HTTP_ORIGIN'];
if ($http_origin == "http://localhost:1112" || $http_origin == "http://localhost:1111") {
    header("Access-Control-Allow-Origin: $http_origin");
    readfile('access.txt');
}else {
    echo"Access Denied";
}
?>
```

Figure 4.1: Server Configuration based on allowed origin

The configuration of CORS is based on the definition of the allowed origin through the variable $http_origin. Then, it sets the Access-Control-Allow-Origin header equal to $http_origin to give access to the client living in the 1112 origin. The following html snippet shows the configuration for both clients living in 1112 and 1113.

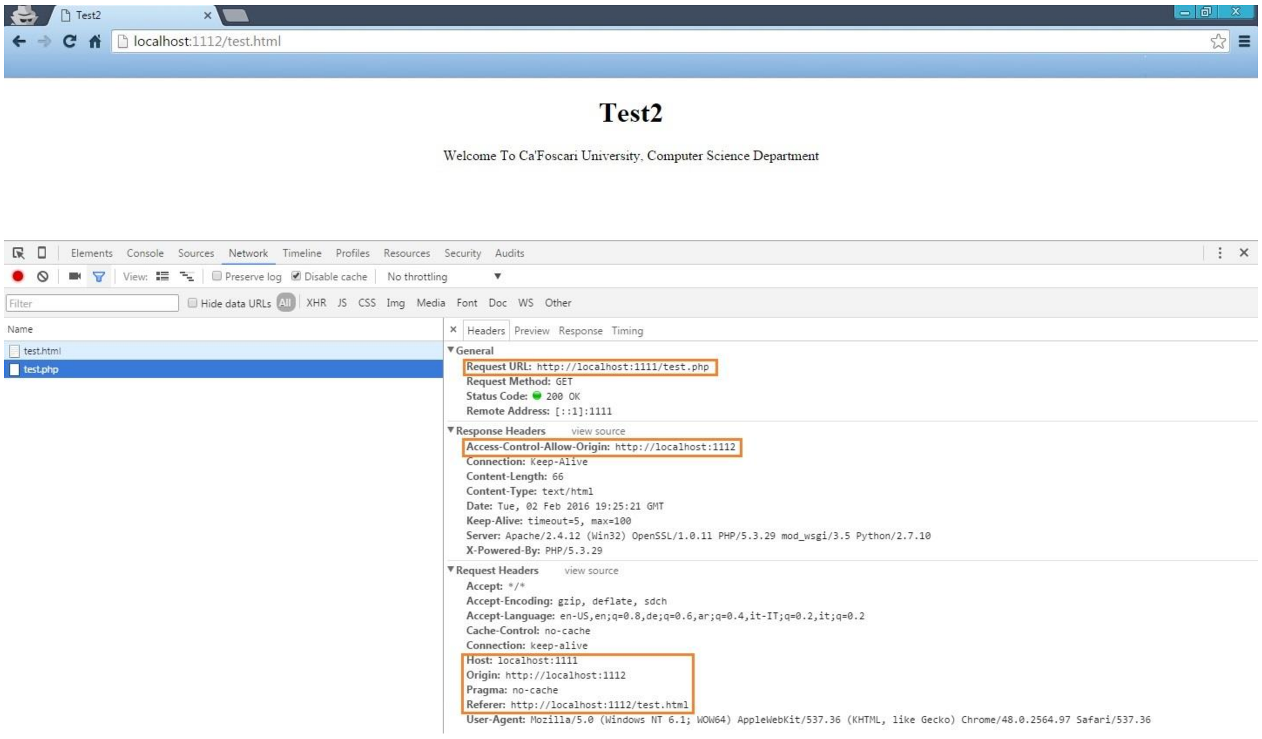


```
1   <!DOCTYPE HTML>
2   <html>
3   <head>
4       <title>Test2</title>
5       <script type="text/javascript">
6           function makerequest(serverPage, objID)
7           {
8               var xmlhttp = new XMLHttpRequest();
9               var obj = document.getElementById(objID);
10              xmlhttp.open("GET", serverPage);
11              xmlhttp.onreadystatechange = function() {
12                  if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
13                      obj.innerHTML = xmlhttp.responseText;
14                  }
15              };
16              xmlhttp.send();
17          }
18      </script>
19  <body onload="makerequest ('http://localhost:1111/test.php','hw')">
20  <div align="center">
21      <h1>Test2</h1>
22      <div id="hw"></div>
23  </div>
24  </body>
25  </html>
```

Figure 4.2: clients Configuration

In the client code, the CORS request starts at line 5 and ends at line 18. In line 6 we create a function that includes two parameters: the server page and objID, which is where the response will be shown. After that, we create the XHR request and assign it to the variable xmlhttp. At line 10 we set the request method to GET. Line 12 is a condition if the request is successful append the response to the objID. Line 16 is meant to send the request.

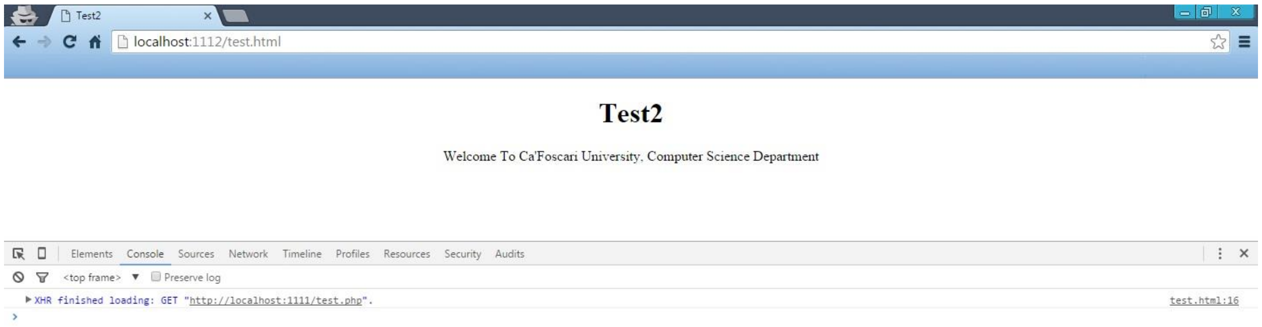
From now on, we will explain how the server responds to requests from two different clients.


Test2
localhost:1112/test.html
Test2
Welcome To Ca'Foscari University, Computer Science Department
Elements Console Sources Network Timeline Profiles Resources Security Audits
View: Preserve log Disable cache No throttling
Filter Hide data URLs All XHR JS CSS Img Media Font Doc WS Other
Name
test.html
test.php
Headers Preview Response Timing

```
▼General
   Request URL: http://localhost:1111/test.php
   Request Method: GET
   Status Code: ● 200 OK
   Remote Address: [::1]:1111
▼Response Headers     view source
   Access-Control-Allow-Origin: http://localhost:1112
   Connection: Keep-Alive
   Content-Length: 66
   Content-Type: text/html
   Date: Tue, 02 Feb 2016 19:25:21 GMT
   Keep-Alive: timeout=5, max=100
   Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1l PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
   X-Powered-By: PHP/5.3.29
▼Request Headers     view source
   Accept: */*
   Accept-Encoding: gzip, deflate, sdch
   Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
   Cache-Control: no-cache
   Connection: keep-alive
   Host: localhost:1111
   Origin: http://localhost:1112
   Pragma: no-cache
   Referer: http://localhost:1112/test.html
   User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.97 Safari/537.36
```

Figure 4.3: CORS Response of Client in origin 1112 using Network tab

In figure (4.3) above, we can see that the URL request http://localhost:1111/test.php and the origin allowed by the server is http://localhost:1112. Since the request is successful, origin 1112 gets a positive response from the server. Furthermore, by using the console tab we can see that the XHR request is successfully finished, as shown in figure (4.4).



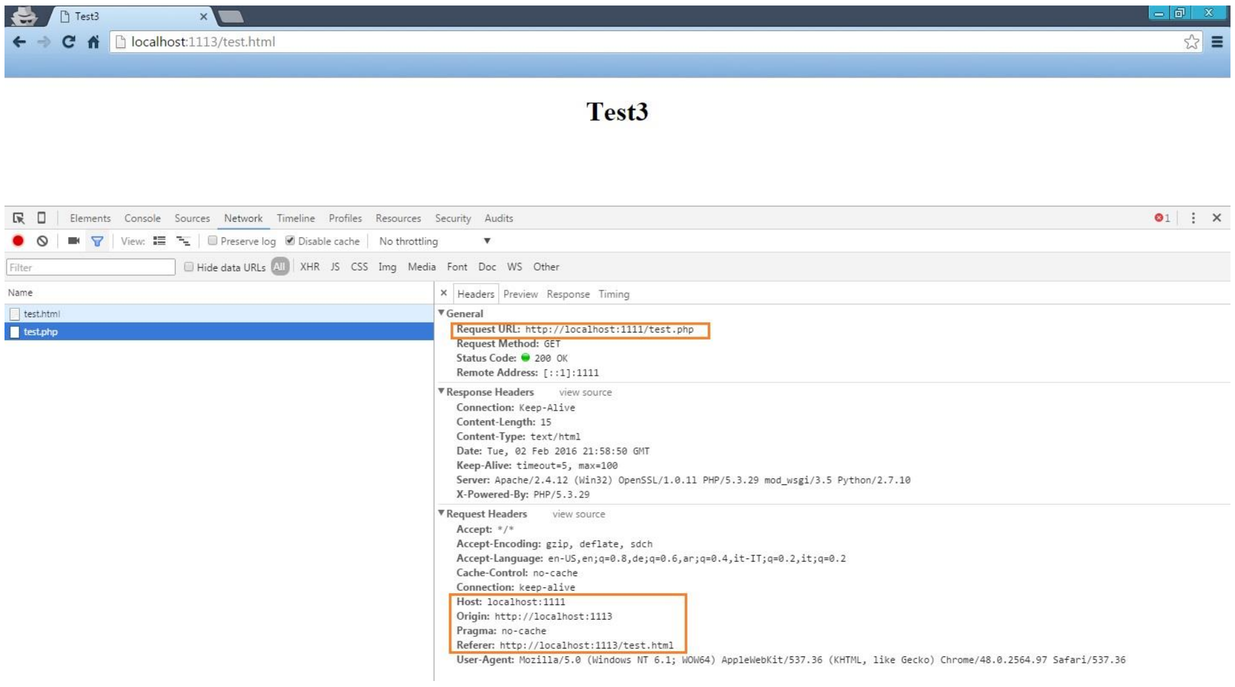
Test2

Welcome To Ca'Foscari University, Computer Science Department

```
▶XHR finished loading: GET "http://localhost:1111/test.php".                    test.html:16
```

Figure 4.4: CORS Response of Client in origin 1112 using Console tab

On the other hand, when origin 1113 sends a request to the server, it fails due to the allowed origin defined in the server. As we can see in figure (4.5), there is no data shown in the page.

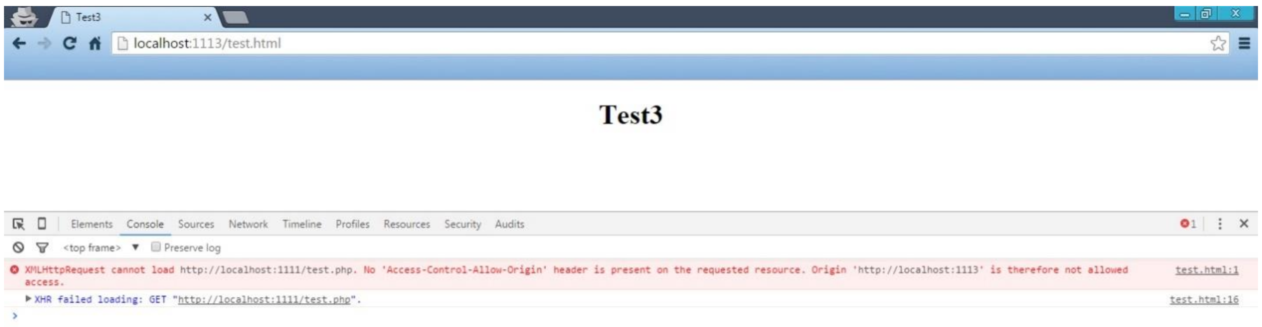


Test3

```
▼General
    Request URL: http://localhost:1111/test.php
    Request Method: GET
    Status Code: ● 200 OK
    Remote Address: [::1]:1111
▼Response Headers
    Connection: Keep-Alive
    Content-Length: 15
    Content-Type: text/html
    Date: Tue, 02 Feb 2016 21:58:50 GMT
    Keep-Alive: timeout=5, max=100
    Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1l PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
    X-Powered-By: PHP/5.3.29
▼Request Headers
    Accept: */*
    Accept-Encoding: gzip, deflate, sdch
    Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
    Cache-Control: no-cache
    Connection: keep-alive
    Host: localhost:1111
    Origin: http://localhost:1113
    Pragma: no-cache
    Referer: http://localhost:1113/test.html
    User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.97 Safari/537.36
```

Figure 4.5: CORS Response of Client in origin 1113 using Network tab

Even if we check in the console tab, we can still see the error message shown by the browser. The reason behind this message is that the browser sends the request to the server to check if this origin matches the allowed one or not. Then, the server rejects the request, hence the browser generates the message shown in figure (4.6).



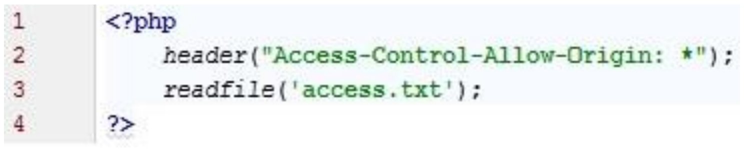
```
XMLHttpRequest cannot load http://localhost:1111/test.php. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:1113' is therefore not allowed access.        test.html:1
▶ XHR failed loading: GET "http://localhost:1111/test.php".        test.html:16
```

Figure 4.6: CORS Response of Client in origin 1113 using Console tab

We can see that the XHR request fails and the browser shows an error message (Origin 'http://localhost:1113' is therefore not allowed access.)

4.1.2 Wild-Card request:

The wild card request is the second kind of CORS simple request. This time the server accepts all the incoming requests from different origins, as shown in figure (4.7).

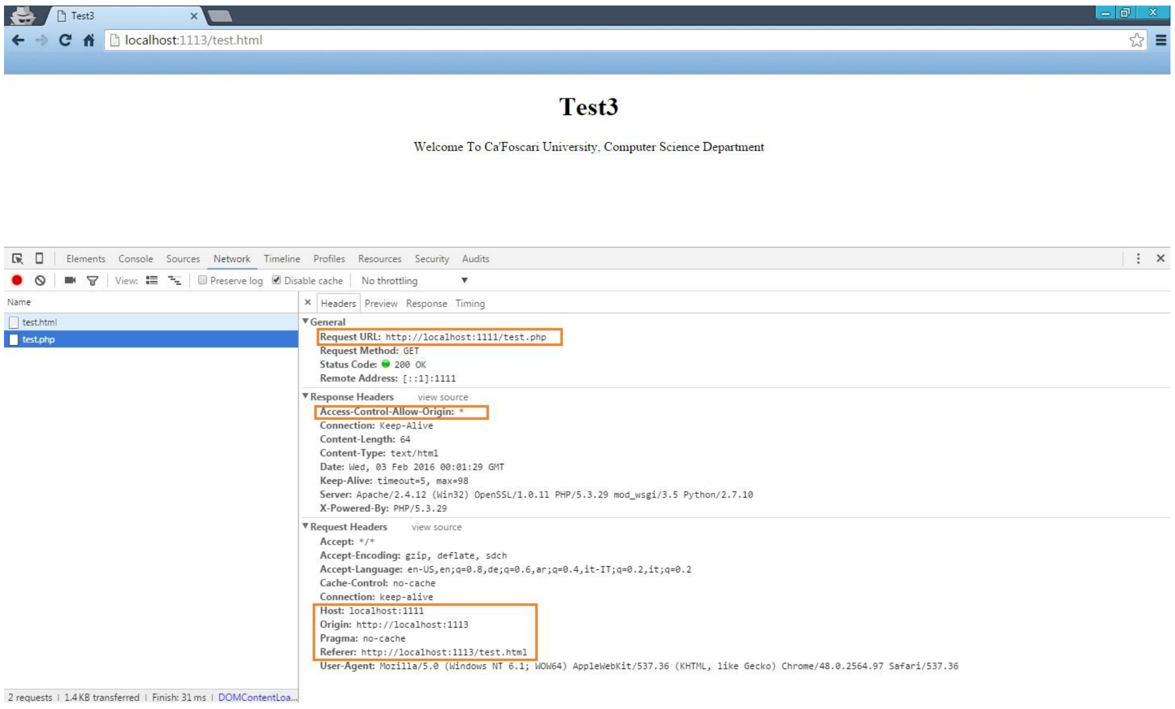


```php
<?php
    header("Access-Control-Allow-Origin: *");
    readfile('access.txt');
?>
```

Figure 4.7: Server Configuration using wildcard

By using the `header("Access-Control-Allow-Origin: *");` in the configuration of the server, we give the permission to all incoming requests to access the data stored in the server.

Therefore, this time the request coming from origin 1113 is allowed, as shown in figure (4.8).



Test3
localhost:1113/test.html
Elements Console Sources Network Timeline Profiles Resources Security Audits
View: Preserve log Disable cache No throttling
Name
test.html
test.php
Headers Preview Response Timing
2 requests | 1.4 KB transferred | Finish: 31 ms | DOMContentLoa...

Test3

Welcome To Ca'Foscari University, Computer Science Department

```
▼General
    Request URL: http://localhost:1111/test.php
    Request Method: GET
    Status Code: 200 OK
    Remote Address: [::1]:1111
▼Response Headers      view source
    Access-Control-Allow-Origin: *
    Connection: Keep-Alive
    Content-Length: 64
    Content-Type: text/html
    Date: Wed, 03 Feb 2016 00:01:29 GMT
    Keep-Alive: timeout=5, max=98
    Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1l PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
    X-Powered-By: PHP/5.3.29
▼Request Headers      view source
    Accept: */*
    Accept-Encoding: gzip, deflate, sdch
    Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
    Cache-Control: no-cache
    Connection: keep-alive
    Host: localhost:1111
    Origin: http://localhost:1113
    Pragma: no-cache
    Referer: http://localhost:1113/test.html
    User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.97 Safari/537.36
```

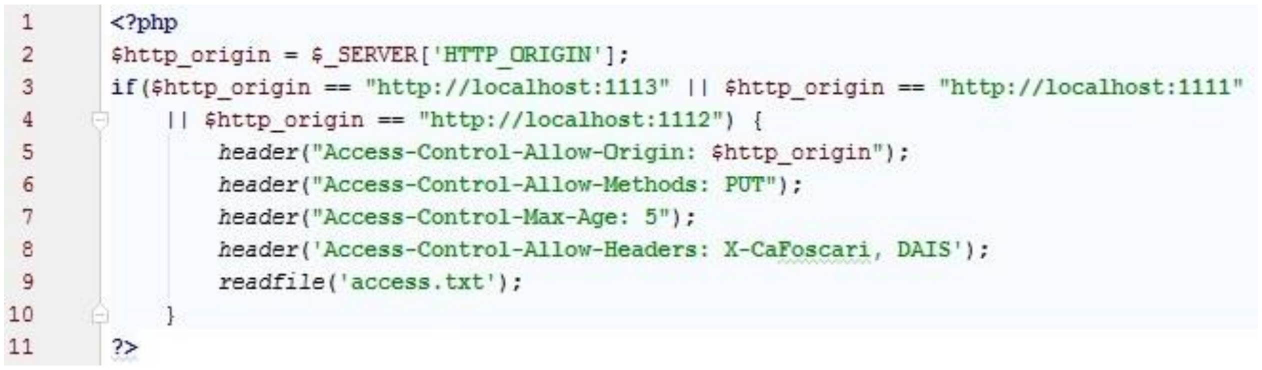
Figure 4.8: CORS Response of Client in origin 1113 using Network tab

According to W3C, the use of wild card should happen only when we want to give the permission to anyone to access the resource (e.g. web-fonts). On the contrary, in case the server includes sensitive information or personal data, wild card should be avoided in CORS requests.

4.2 Preflight-Request:

In this part we investigate the aim of preflight-request. In our example, we make two experiments for Preflight request: in the first one we use the method PUT and in the second one we use custom header (X-Ca'foscari,DAIS). For the server we use a new header (`header("Access-Control-Max-Age: 5")`), in order to improve the performance of the request. In this way, every time the browser makes two requests: the pre-flight one and the actual request. Thus, this header allows the response of the preflight within a given time that to be cashed, for instance, in our case lasts only 5 seconds.

The following PHP snippet shows the configuration of the server



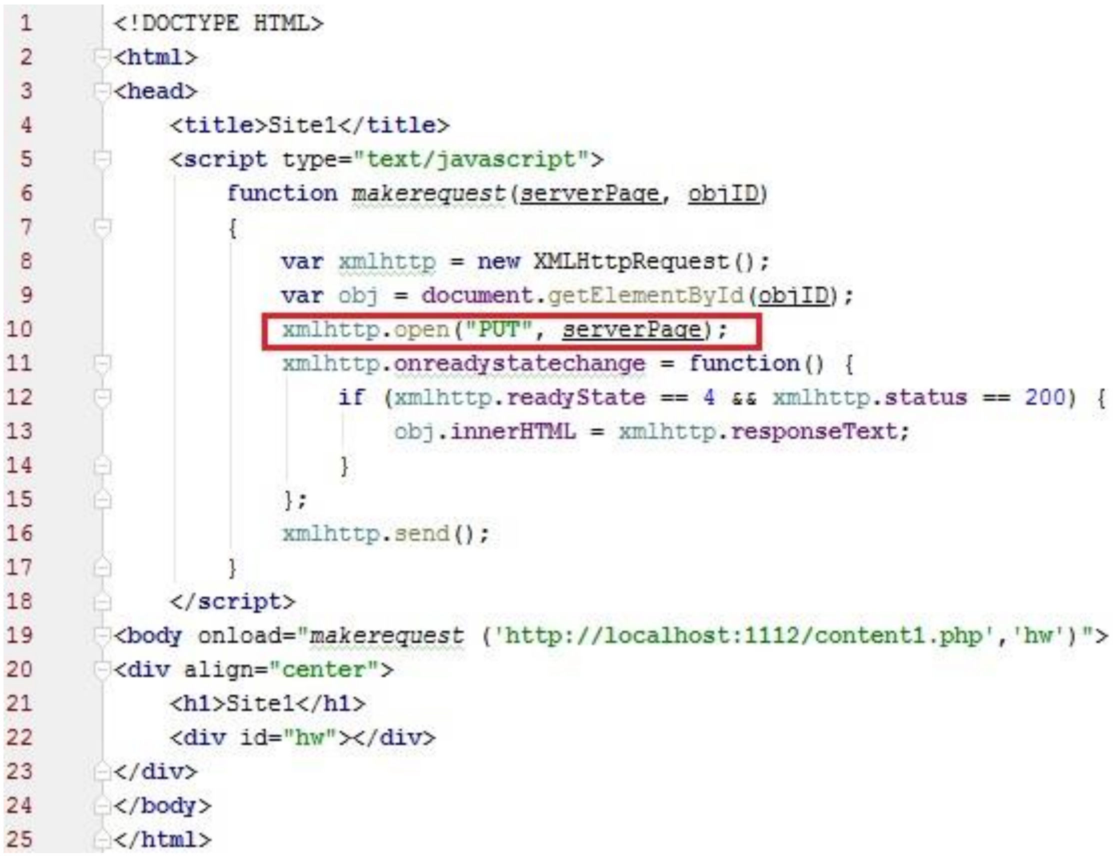
```php
<?php
$http_origin = $_SERVER['HTTP_ORIGIN'];
if($http_origin == "http://localhost:1113" || $http_origin == "http://localhost:1111"
    || $http_origin == "http://localhost:1112") {
        header("Access-Control-Allow-Origin: $http_origin");
        header("Access-Control-Allow-Methods: PUT");
        header("Access-Control-Max-Age: 5");
        header('Access-Control-Allow-Headers: X-CaFoscari, DAIS');
        readfile('access.txt');
    }
?>
```

Figure 4.9: Server Configuration for Preflight response

As we can see, there are new headers in the configuration of the server.

The first one (`header("Access-Control-Allow-Methods: PUT");`) defines the allowed method; the second one (`header('Access-Control-Allow-Headers: X-CaFoscari, DAIS');`) defines the allowed Headers; the last one is (`header("Access-Control-Max-Age: 5");`) responsible for caching the response of preflight request.

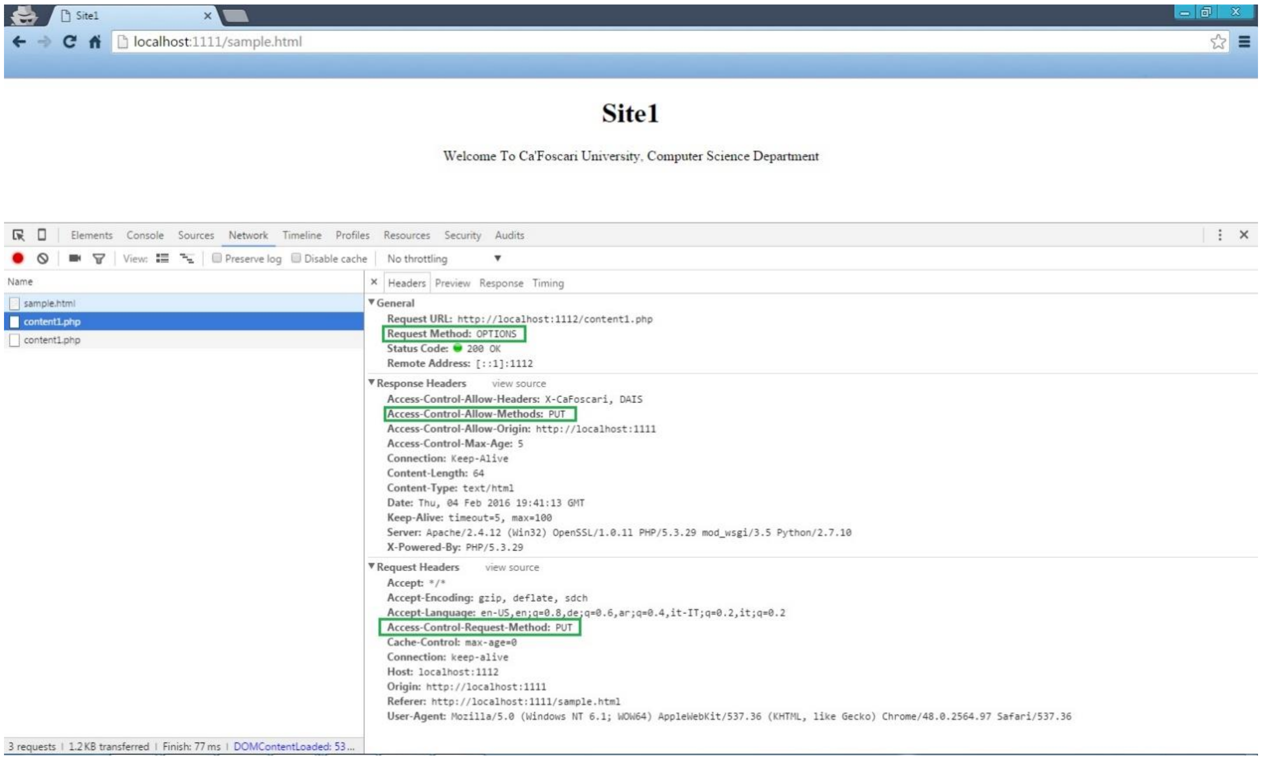
About the client we have two pages. Let's start by examining the first one. This first page has the origin Localhost:1111/sample.html and makes a request using PUT method. The following snippet code shows the code of the page.



```
1   <!DOCTYPE HTML>
2   <html>
3   <head>
4       <title>Site1</title>
5       <script type="text/javascript">
6           function makerequest(serverPage, objID)
7           {
8               var xmlhttp = new XMLHttpRequest();
9               var obj = document.getElementById(objID);
10              xmlhttp.open("PUT", serverPage);
11              xmlhttp.onreadystatechange = function() {
12                  if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
13                      obj.innerHTML = xmlhttp.responseText;
14                  }
15              };
16              xmlhttp.send();
17          }
18      </script>
19  <body onload="makerequest ('http://localhost:1112/content1.php','hw')">
20  <div align="center">
21      <h1>Site1</h1>
22      <div id="hw"></div>
23  </div>
24  </body>
25  </html>
```

Figure 4.10: Client Code in origin 1111

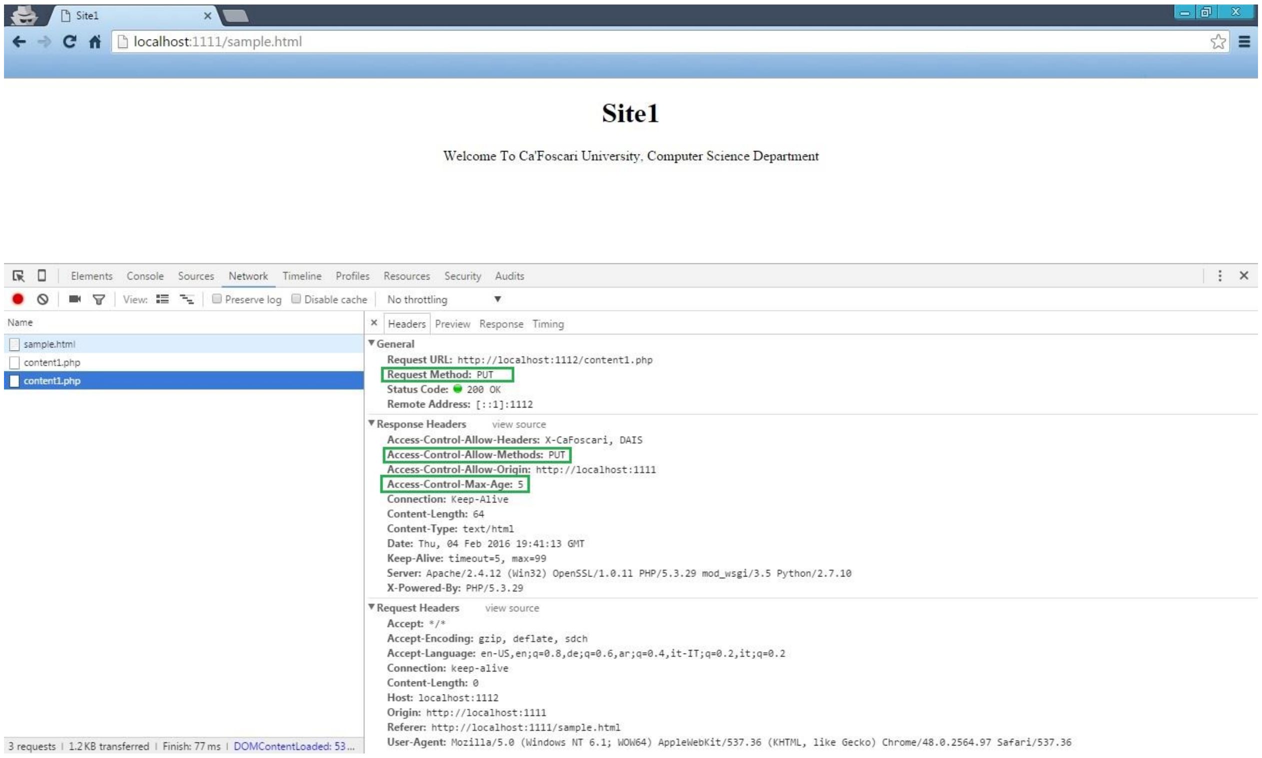
As we can see in line 10, the method of the request is PUT, when site1 makes a request through this method, the server considers it as a preflight request, as we shown in figure (4.11).


Site1
localhost:1111/sample.html
Site1
Welcome To Ca'Foscari University, Computer Science Department
Elements Console Sources Network Timeline Profiles Resources Security Audits
Preserve log Disable cache No throttling
Name
sample.html
content1.php
content1.php
Headers Preview Response Timing
3 requests | 1.2 KB transferred | Finish: 77 ms | DOMContentLoaded: 53...

```
▼General
   Request URL: http://localhost:1112/content1.php
   Request Method: OPTIONS
   Status Code: 200 OK
   Remote Address: [::1]:1112
▼Response Headers     view source
   Access-Control-Allow-Headers: X-CaFoscari, DAIS
   Access-Control-Allow-Methods: PUT
   Access-Control-Allow-Origin: http://localhost:1111
   Access-Control-Max-Age: 5
   Connection: Keep-Alive
   Content-Length: 64
   Content-Type: text/html
   Date: Thu, 04 Feb 2016 19:41:13 GMT
   Keep-Alive: timeout=5, max=100
   Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1l PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
   X-Powered-By: PHP/5.3.29
▼Request Headers     view source
   Accept: */*
   Accept-Encoding: gzip, deflate, sdch
   Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
   Access-Control-Request-Method: PUT
   Cache-Control: max-age=0
   Connection: keep-alive
   Host: localhost:1112
   Origin: http://localhost:1111
   Referer: http://localhost:1111/sample.html
   User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.97 Safari/537.36
```

Figure 4.11: Client in origin 1111 preflight request

The figure shows the OPTION request (preflight request), with the request method PUT. As we said before, the browser sends the preflight request to determine the allowed method, that we can find in the Response Headers section as (`("Access-Control-Allow-Methods: PUT");`). Thus, the allowed method results to be PUT. After that, the browser sends the actual request to retrieve information, as we can see in figure (4.12).


Site1
localhost:1111/sample.html
Elements Console Sources Network Timeline Profiles Resources Security Audits
Preserve log Disable cache No throttling
Name
sample.html
content1.php
content1.php
Headers Preview Response Timing
3 requests | 1.2 KB transferred | Finish: 77 ms | DOMContentLoaded: 53...

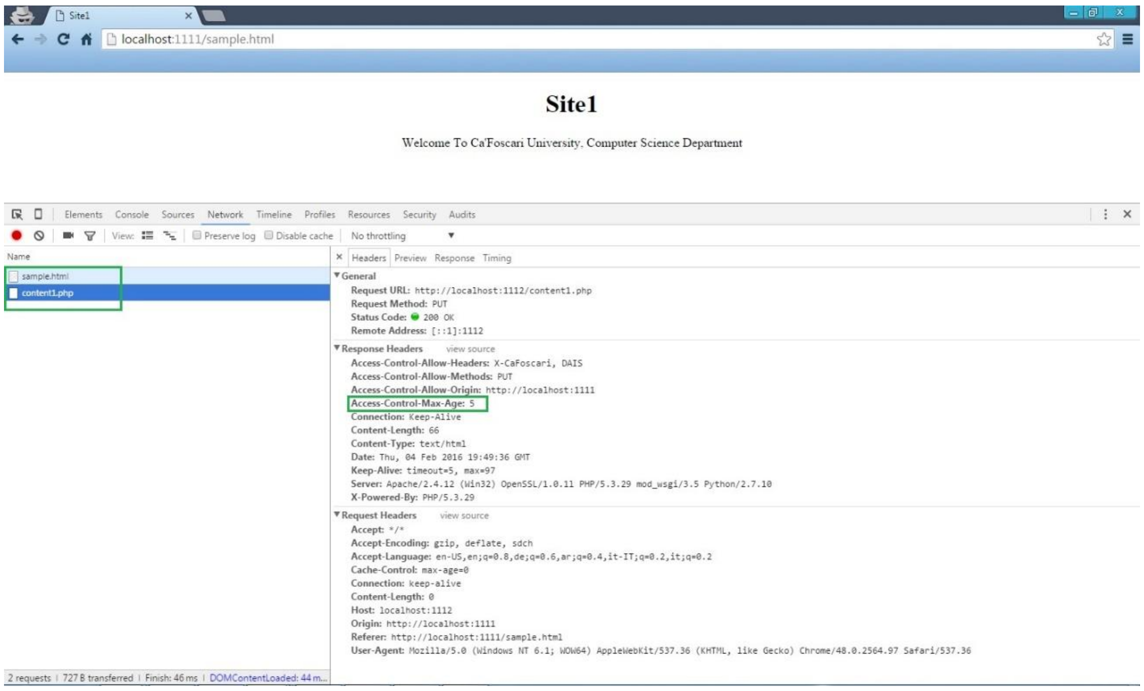
Site1

Welcome To Ca'Foscari University, Computer Science Department

```
▼General
  Request URL: http://localhost:1112/content1.php
  Request Method: PUT
  Status Code: 200 OK
  Remote Address: [::1]:1112
▼Response Headers    view source
  Access-Control-Allow-Headers: X-CaFoscari, DAIS
  Access-Control-Allow-Methods: PUT
  Access-Control-Allow-Origin: http://localhost:1111
  Access-Control-Max-Age: 5
  Connection: Keep-Alive
  Content-Length: 64
  Content-Type: text/html
  Date: Thu, 04 Feb 2016 19:41:13 GMT
  Keep-Alive: timeout=5, max=99
  Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1l PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
  X-Powered-By: PHP/5.3.29
▼Request Headers    view source
  Accept: */*
  Accept-Encoding: gzip, deflate, sdch
  Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
  Connection: keep-alive
  Content-Length: 0
  Host: localhost:1112
  Origin: http://localhost:1111
  Referer: http://localhost:1111/sample.html
  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.97 Safari/537.36
```

Figure 4.12: Actual request using PUT method

The figure above represents the actual request with PUT method. Furthermore, we can see also the (`("Access-Control-Max-Age: 5");`) header, which indicates a 5 seconds time response for the preflight request. In doing so, instead of making the browser send the preflight request and the actual one every time the client tries to retrieve the information, we capture it within the time of cached response in figure (4.13). Shows that the preflight request does not exist.



Site1

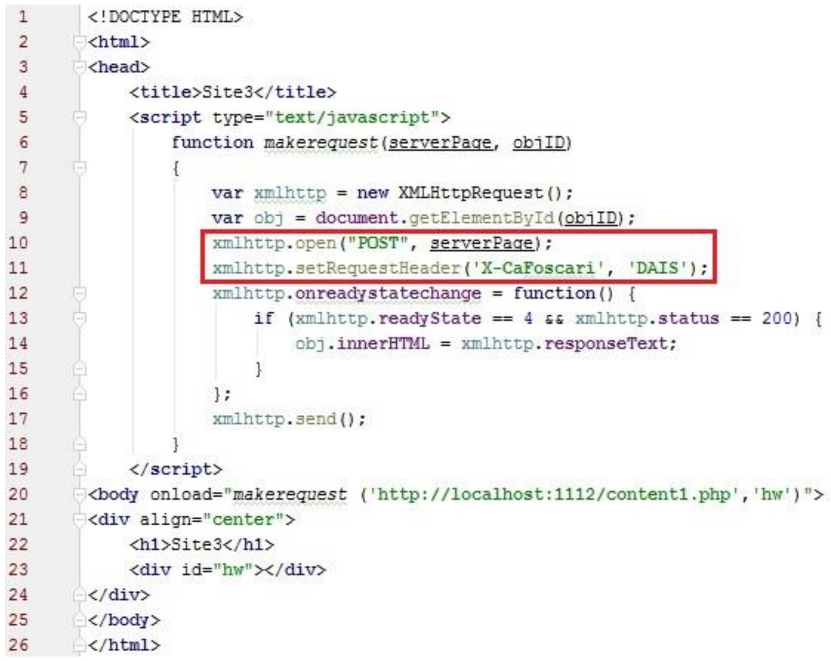
Welcome To Ca'Foscari University, Computer Science Department

```
General
    Request URL: http://localhost:1112/content1.php
    Request Method: PUT
    Status Code: 200 OK
    Remote Address: [::1]:1112
Response Headers    view source
    Access-Control-Allow-Headers: X-CaFoscari, DAIS
    Access-Control-Allow-Methods: PUT
    Access-Control-Allow-Origin: http://localhost:1111
    Access-Control-Max-Age: 5
    Connection: Keep-Alive
    Content-Length: 66
    Content-Type: text/html
    Date: Thu, 04 Feb 2016 19:49:36 GMT
    Keep-Alive: timeout=5, max=97
    Server: Apache/2.4.12 (Win32) OpenSSL/1.0.11 PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
    X-Powered-By: PHP/5.3.29
Request Headers    view source
    Accept: */*
    Accept-Encoding: gzip, deflate, sdch
    Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
    Cache-Control: max-age=0
    Connection: keep-alive
    Content-Length: 0
    Host: localhost:1112
    Origin: http://localhost:1111
    Referer: http://localhost:1111/sample.html
    User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.97 Safari/537.36
```

2 requests | 727 B transferred | Finish: 46 ms | DOMContentLoaded: 44 m...

Figure 4.13: Cashed Preflight response

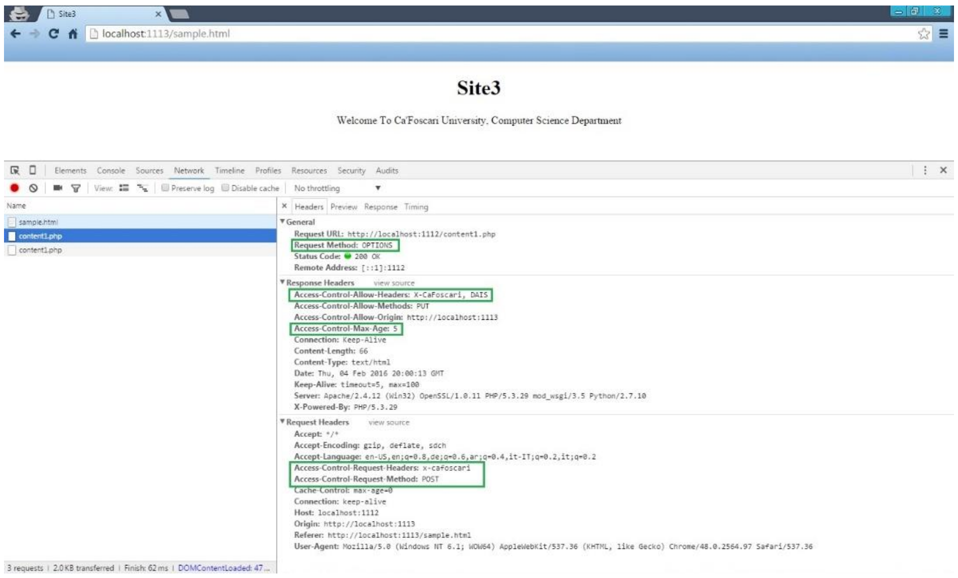
Now we move to the second page experiment, where we use the custom header (X-Ca’foscari,DAIS). Here the second client has the origin Localhost:1113/sample.html and it makes a request by using POST method with custom header (X-Ca’foscari,DAIS). The following snippet code shows the code of the page.



```
1   <!DOCTYPE HTML>
2   <html>
3   <head>
4       <title>Site3</title>
5       <script type="text/javascript">
6           function makerequest(serverPage, objID)
7           {
8               var xmlhttp = new XMLHttpRequest();
9               var obj = document.getElementById(objID);
10              xmlhttp.open("POST", serverPage);
11              xmlhttp.setRequestHeader('X-CaFoscari', 'DAIS');
12              xmlhttp.onreadystatechange = function() {
13                  if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
14                      obj.innerHTML = xmlhttp.responseText;
15                  }
16              };
17              xmlhttp.send();
18          }
19      </script>
20  <body onload="makerequest ('http://localhost:1112/content1.php','hw')">
21  <div align="center">
22      <h1>Site3</h1>
23      <div id="hw"></div>
24  </div>
25  </body>
26  </html>
```

Figure 4.14: Client Code in origin 1113

Line 10 shows the method of request, while line 11 shows the custom header (X-Ca'foscari,DAIS). When site 2 makes a request by using this method of the custom header, the server considers it as a preflight request, as shown in figure (4.15).



Site3

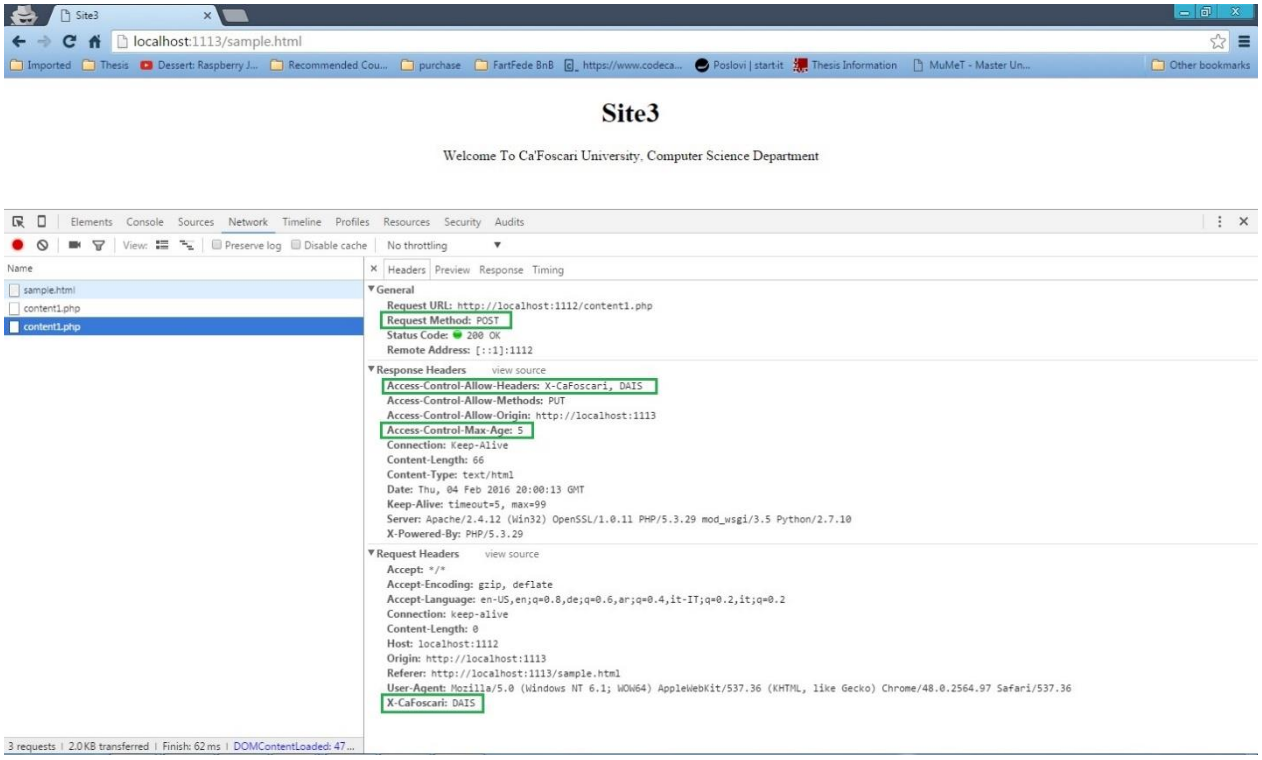
Welcome To Ca'Foscari University, Computer Science Department

```
▼General
   Request URL: http://localhost:1112/content1.php
   Request Method: OPTIONS
   Status Code: 200 OK
   Remote Address: [::1]:1112
▼Response Headers     view source
   Access-Control-Allow-Headers: X-CaFoscari, DAIS
   Access-Control-Allow-Methods: PUT
   Access-Control-Allow-Origin: http://localhost:1113
   Access-Control-Max-Age: 5
   Connection: Keep-Alive
   Content-Length: 66
   Content-Type: text/html
   Date: Thu, 04 Feb 2016 20:00:13 GMT
   Keep-Alive: timeout=5, max=100
   Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1l PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
   X-Powered-By: PHP/5.3.29
▼Request Headers     view source
   Accept: */*
   Accept-Encoding: gzip, deflate, sdch
   Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
   Access-Control-Request-Headers: x-cafoscari
   Access-Control-Request-Method: POST
   Cache-Control: max-age=0
   Connection: keep-alive
   Host: localhost:1112
   Origin: http://localhost:1113
   Referer: http://localhost:1113/sample.html
   User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.97 Safari/537.36
```

Site3
localhost:1113/sample.html
Elements Console Sources Network Timeline Profiles Resources Security Audits
View: Preserve log Disable cache No throttling
Name
sample.html
content1.php
content1.php
Headers Preview Response Timing
3 requests | 2.0 KB transferred | Finish: 62 ms | DOMContentLoaded: 47...

Figure 4.15: Client in origin 1113 preflight request

The figure shows the OPTIONS request (preflight request), with custom header (X-Ca'foscari,DAIS) with the method POST, we see in the Response Headers section, header (`('Access-Control-Allow-Headers: X-CaFoscari, DAIS');`), then the browser sends the actual request to retrieve the information, as we see in figure (4.16).

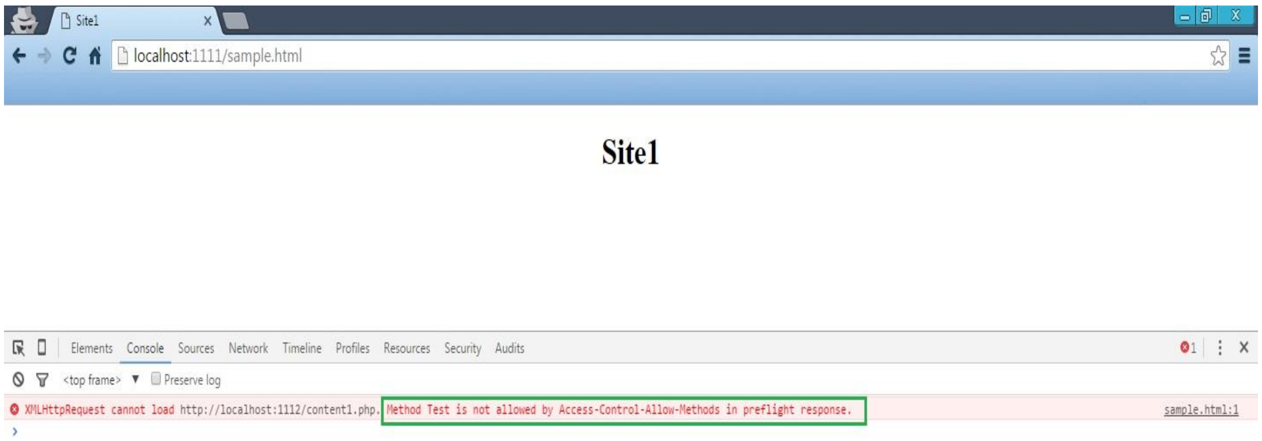


```
▼General
    Request URL: http://localhost:1112/content1.php
    Request Method: POST
    Status Code: 200 OK
    Remote Address: [::1]:1112
▼Response Headers    view source
    Access-Control-Allow-Headers: X-CaFoscari, DAIS
    Access-Control-Allow-Methods: PUT
    Access-Control-Allow-Origin: http://localhost:1113
    Access-Control-Max-Age: 5
    Connection: Keep-Alive
    Content-Length: 66
    Content-Type: text/html
    Date: Thu, 04 Feb 2016 20:00:13 GMT
    Keep-Alive: timeout=5, max=99
    Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1l PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
    X-Powered-By: PHP/5.3.29
▼Request Headers    view source
    Accept: */*
    Accept-Encoding: gzip, deflate
    Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
    Connection: keep-alive
    Content-Length: 0
    Host: localhost:1112
    Origin: http://localhost:1113
    Referer: http://localhost:1113/sample.html
    User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.97 Safari/537.36
    X-CaFoscari: DAIS
```

Figure 4.16: Actual request using POST method and custom header X-CaFoscari DAIS

We can notice that the actual request uses the custom header X-CaFoscari DAIS, and the same as site one we see the Access-Control-Max-Age header equal to 5 seconds and within this time the browser will not send the preflight request again.

The previous two experiments show the server response when accept the preflight request, the last experiment showing the response in case of rejecting the preflight request, as we see in figure (4.17)



```
XMLHttpRequest cannot load http://localhost:1112/content1.php. Method Test is not allowed by Access-Control-Allow-Methods in preflight response.    sample.html:1
```

Figure 4.17: Rejecting Preflight-request using method TEST

As we see, the browser response in this case is (Method TEST is not allowed in preflight response), because we didn’t define this method in the server, for this reason the request rejected.

4.2 Advanced Request:

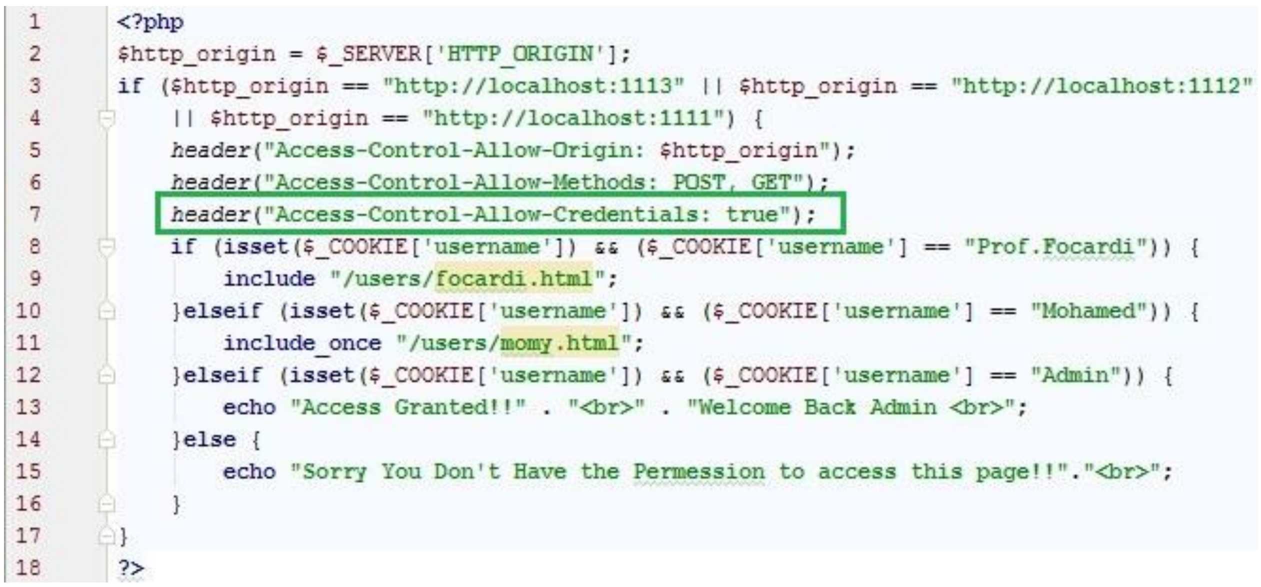
In this experiment, we create a toy site that includes 3 pages. The first one is the login page, which contains a form of username and password. The function of this form is to create a cookies based on the username input and then to send the cookies to the second page. Here the CORS request can be made including the cookies (credentials) or even without them. In order to do so, we need to add the following line of code in the request:

```
XMLHttpRequest.withCredentials = true;
```

With regard to the third page, which is stored in the server, it checks the cookies and, based on them, retrieves information. To make the server check whether the request includes the cookies (credentials), we use the following line of code:

```
header("Access-Control-Allow-Credentials: true");
```

Now, we will show the difference between including the credentials (cookies) in the request and not including them. To achieve our aim, we create two pages responsible for making the CORS request. The first one includes the credentials in the request; the other one does not include them. The following snippet code shows the configuration of the server page.

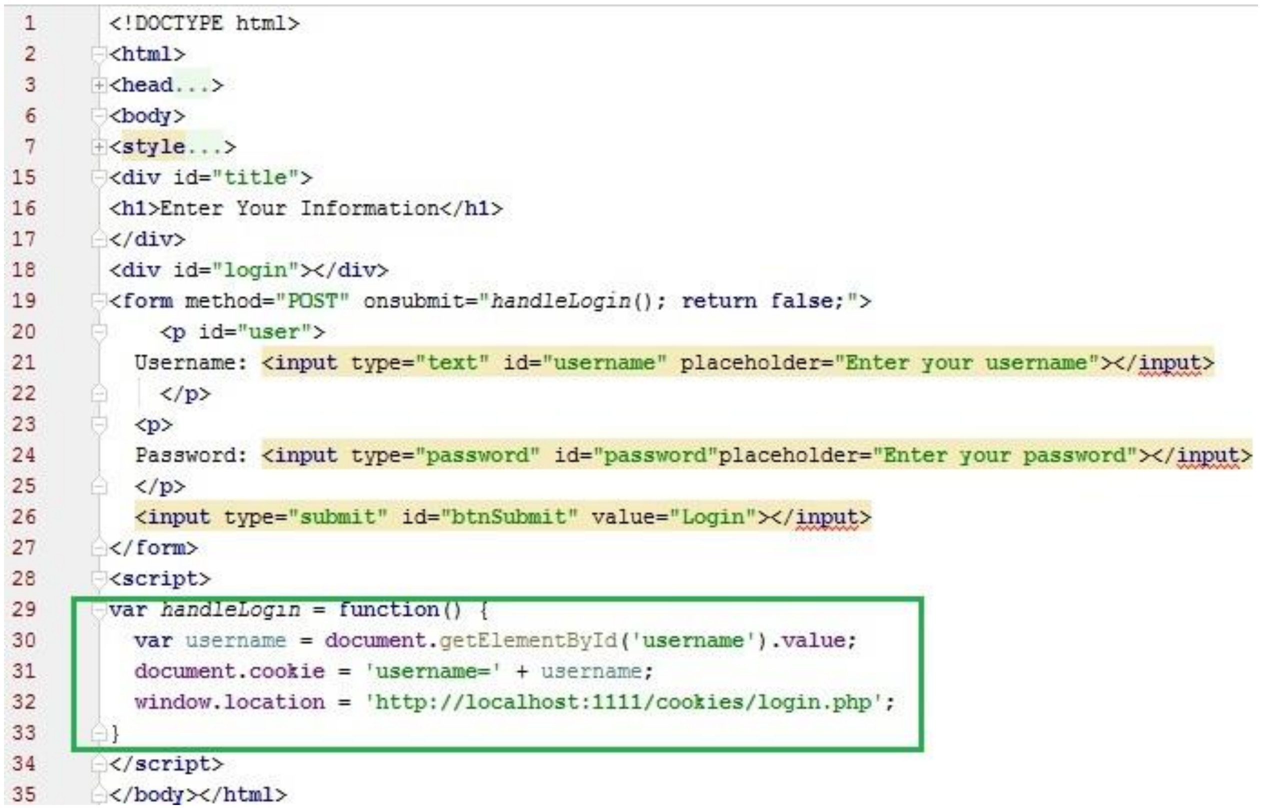


```php
<?php
$http_origin = $_SERVER['HTTP_ORIGIN'];
if ($http_origin == "http://localhost:1113" || $http_origin == "http://localhost:1112"
    || $http_origin == "http://localhost:1111") {
    header("Access-Control-Allow-Origin: $http_origin");
    header("Access-Control-Allow-Methods: POST, GET");
    header("Access-Control-Allow-Credentials: true");
    if (isset($_COOKIE['username']) && ($_COOKIE['username'] == "Prof.Focardi")) {
        include "/users/focardi.html";
    }elseif (isset($_COOKIE['username']) && ($_COOKIE['username'] == "Mohamed")) {
        include_once "/users/momy.html";
    }elseif (isset($_COOKIE['username']) && ($_COOKIE['username'] == "Admin")) {
        echo "Access Granted!!" . "<br>" . "Welcome Back Admin <br>";
    }else {
        echo "Sorry You Don't Have the Permession to access this page!!"."<br>";
    }
}
?>
```

Figure 4.18: Server Configuration for Advanced Request

We can see that the server includes the header `header("Access-Control-Allow-Credentials: true");` to check whether the cookies are included in the request. In case the cookies match with the defined one, it leads us to the profile page of the user.

Let's examine now a first case that does not include the cookies in the CORS request. The following snippet code shows the code of the first page which lives in origin localhost:1111

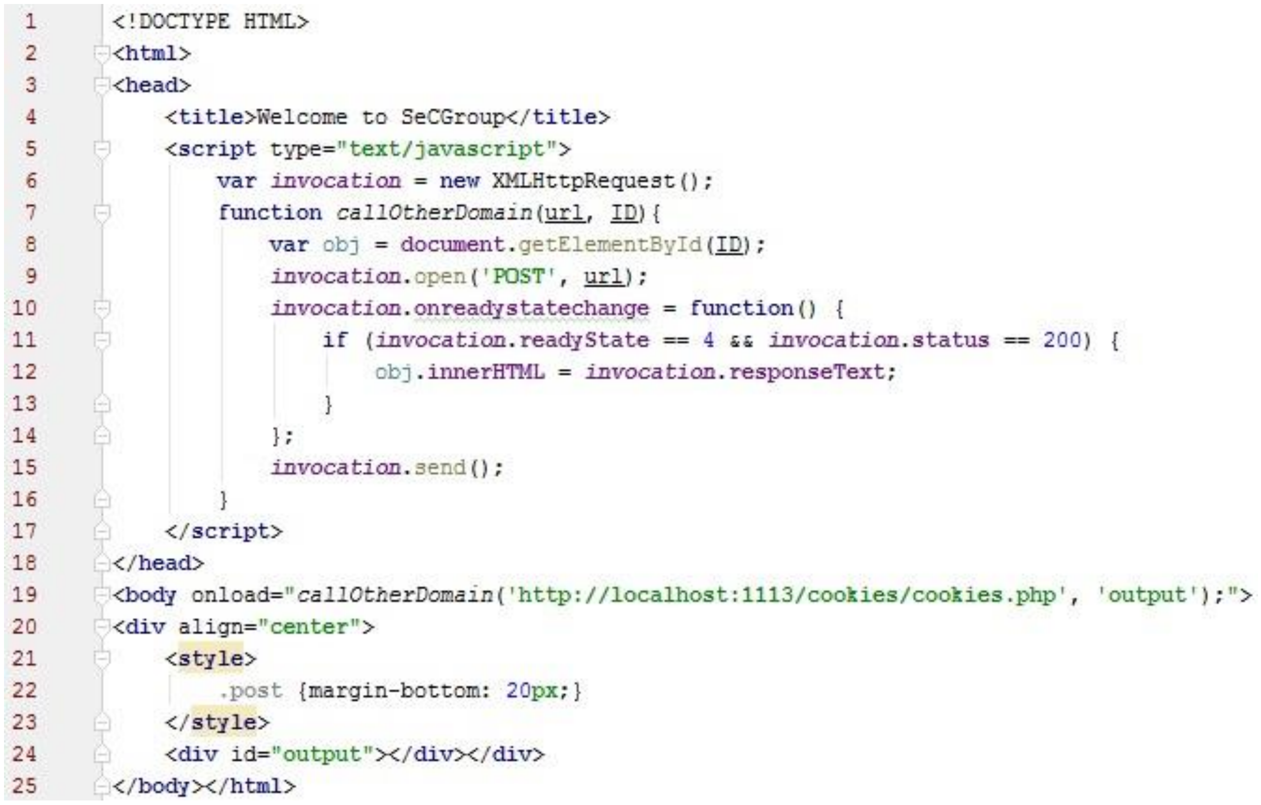


```
1   <!DOCTYPE html>
2   <html>
3   <head...>
6   <body>
7   <style...>
15  <div id="title">
16  <h1>Enter Your Information</h1>
17  </div>
18  <div id="login"></div>
19  <form method="POST" onsubmit="handleLogin(); return false;">
20      <p id="user">
21    Username: <input type="text" id="username" placeholder="Enter your username"></input>
22      </p>
23    <p>
24    Password: <input type="password" id="password"placeholder="Enter your password"></input>
25    </p>
26    <input type="submit" id="btnSubmit" value="Login"></input>
27  </form>
28  <script>
29  var handleLogin = function() {
30    var username = document.getElementById('username').value;
31    document.cookie = 'username=' + username;
32    window.location = 'http://localhost:1111/cookies/login.php';
33  }
34  </script>
35  </body></html>
```

Figure 4.19: Login Page code of origin 1111

We can notice that the function in line 31 assigns the cookies to the username input, while in line 32 send it assigns the cookies to another page, that is where the request of CORS was made.

With regard to the page responsible for the CORS request, the following snippet code shows that the request does not include the credentials.



```
<!DOCTYPE HTML>
<html>
<head>
    <title>Welcome to SeCGroup</title>
    <script type="text/javascript">
        var invocation = new XMLHttpRequest();
        function callOtherDomain(url, ID){
            var obj = document.getElementById(ID);
            invocation.open('POST', url);
            invocation.onreadystatechange = function() {
                if (invocation.readyState == 4 && invocation.status == 200) {
                    obj.innerHTML = invocation.responseText;
                }
            };
            invocation.send();
        }
    </script>
</head>
<body onload="callOtherDomain('http://localhost:1113/cookies/cookies.php', 'output');">
<div align="center">
    <style>
        .post {margin-bottom: 20px;}
    </style>
    <div id="output"></div></div>
</body></html>
```

Figure 4.20: CORS Request for origin 1111 without credentials

As we can see, the page does not include the line of code which is responsible for the cookies (credentials), hence the CORS request will not include it. Consequently, even when the client tries to submit the form by using one of the defined cookies in the server, as shown in figure (4.21)

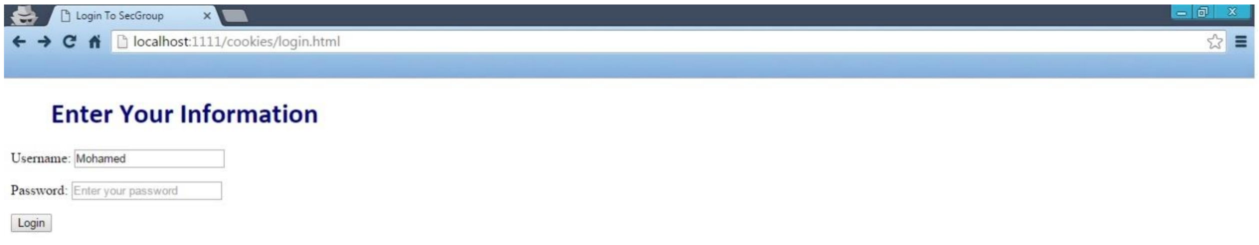
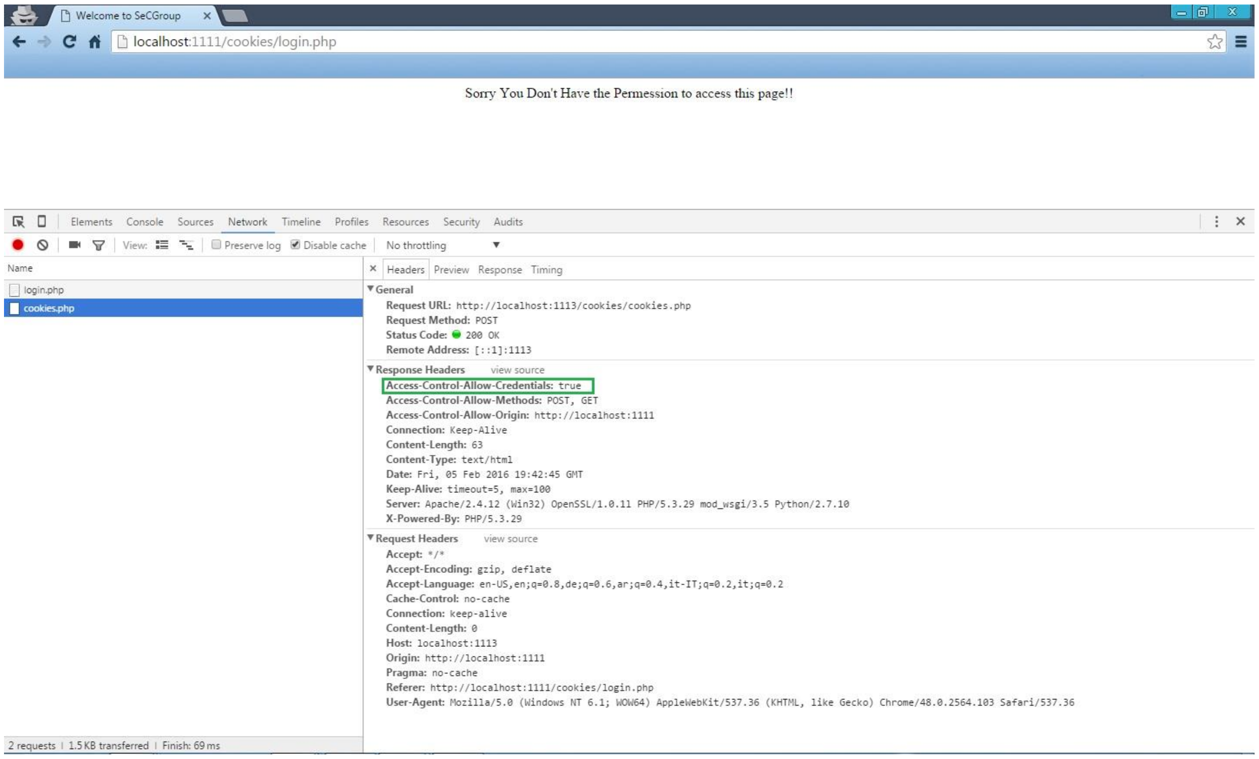

Login To SecGroup
localhost:1111/cookies/login.html
Enter Your Information
Username: Mohamed
Password: Enter your password
Login

Figure 4.21: Login page of origin 1111

Then make the CORS request, as shown in figure (4.22)


Welcome to SeCGroup
localhost:1111/cookies/login.php
Elements Console Sources Network Timeline Profiles Resources Security Audits
Preserve log Disable cache No throttling
Name
login.php
cookies.php
Headers Preview Response Timing
2 requests | 1.5 KB transferred | Finish: 69 ms

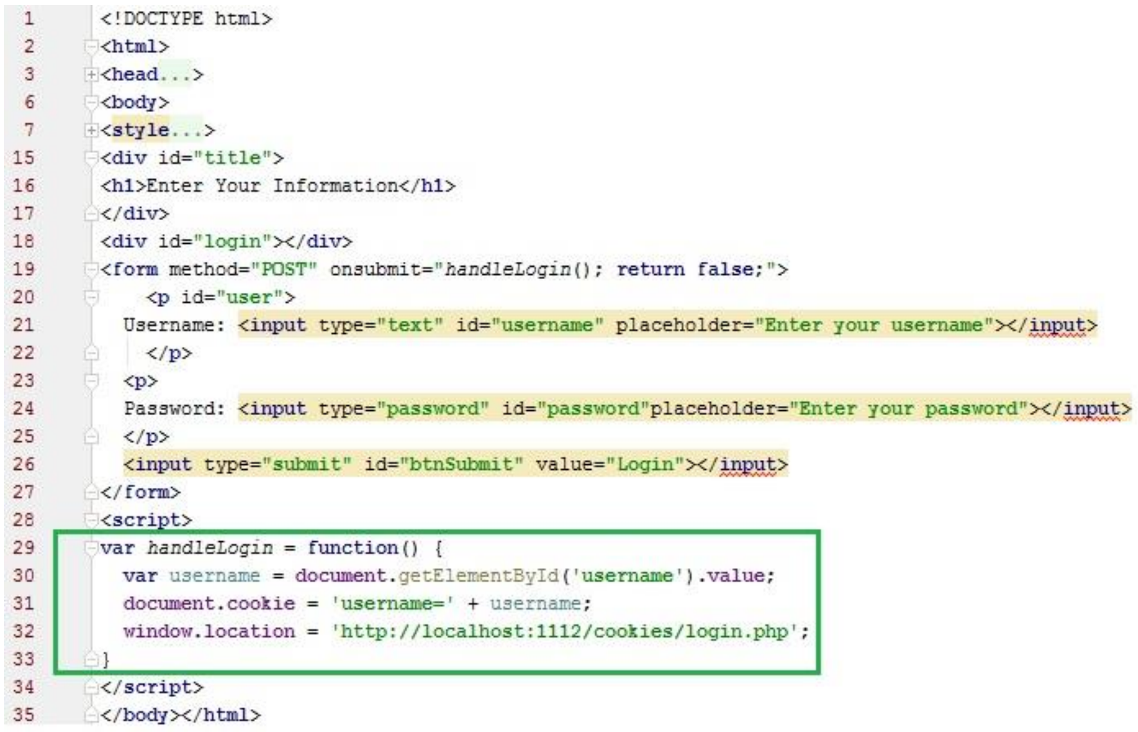
Sorry You Don't Have the Permession to access this page!!

```
▼General
    Request URL: http://localhost:1113/cookies/cookies.php
    Request Method: POST
    Status Code: 200 OK
    Remote Address: [::1]:1113
▼Response Headers    view source
    Access-Control-Allow-Credentials: true
    Access-Control-Allow-Methods: POST, GET
    Access-Control-Allow-Origin: http://localhost:1111
    Connection: Keep-Alive
    Content-Length: 63
    Content-Type: text/html
    Date: Fri, 05 Feb 2016 19:42:45 GMT
    Keep-Alive: timeout=5, max=100
    Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1l PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
    X-Powered-By: PHP/5.3.29
▼Request Headers    view source
    Accept: */*
    Accept-Encoding: gzip, deflate
    Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
    Cache-Control: no-cache
    Connection: keep-alive
    Content-Length: 0
    Host: localhost:1113
    Origin: http://localhost:1111
    Pragma: no-cache
    Referer: http://localhost:1111/cookies/login.php
    User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.103 Safari/537.36
```

Figure 4.22: Response of unauthenticated user in origin 1111

An error message is displayed to communicate to the user that he cannot access the page.

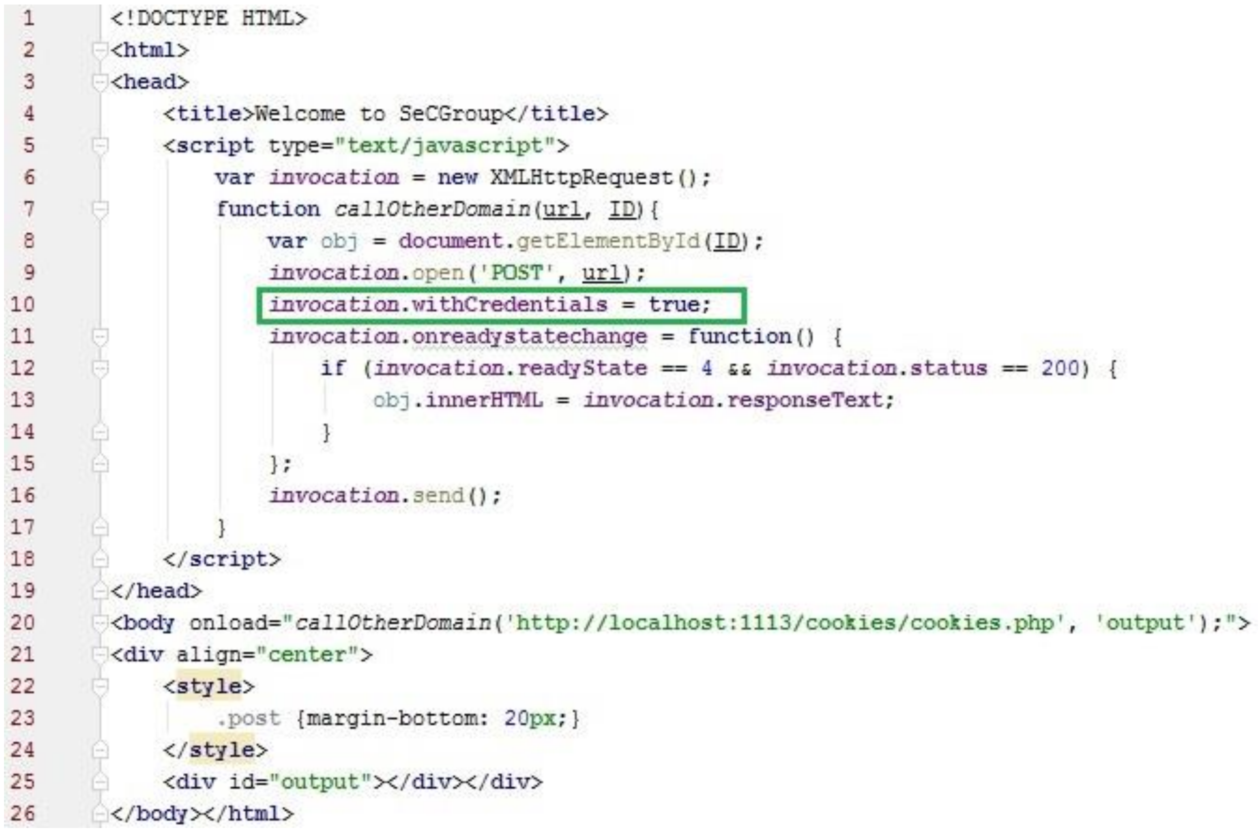
Let’s examine now a second case where we include the cookies in the CORS request. Figure (4.23) shows the code of the login page which lives in origin localhost:1112



```
<!DOCTYPE html>
<html>
<head...>
<body>
<style...>
<div id="title">
<h1>Enter Your Information</h1>
</div>
<div id="login"></div>
<form method="POST" onsubmit="handleLogin(); return false;">
    <p id="user">
  Username: <input type="text" id="username" placeholder="Enter your username"></input>
    </p>
  <p>
  Password: <input type="password" id="password"placeholder="Enter your password"></input>
  </p>
  <input type="submit" id="btnSubmit" value="Login"></input>
</form>
<script>
var handleLogin = function() {
  var username = document.getElementById('username').value;
  document.cookie = 'username=' + username;
  window.location = 'http://localhost:1112/cookies/login.php';
}
</script>
</body></html>
```

Figure 4.23: Login Page code of origin 1112

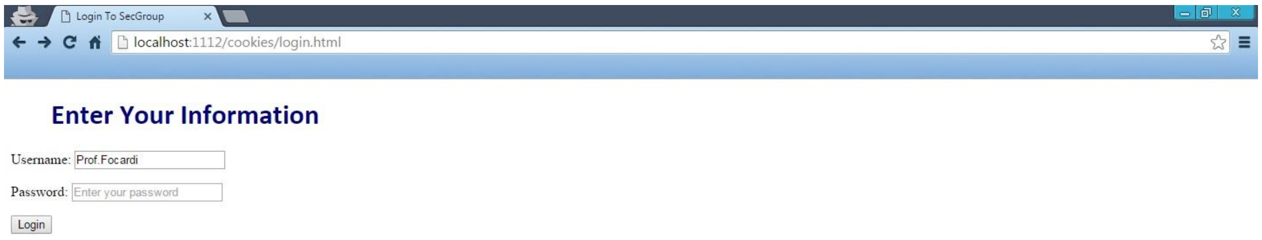
The following snippet code shows a CORS request that includes the credentials.



```
<!DOCTYPE HTML>
<html>
<head>
    <title>Welcome to SeCGroup</title>
    <script type="text/javascript">
        var invocation = new XMLHttpRequest();
        function callOtherDomain(url, ID){
            var obj = document.getElementById(ID);
            invocation.open('POST', url);
            invocation.withCredentials = true;
            invocation.onreadystatechange = function() {
                if (invocation.readyState == 4 && invocation.status == 200) {
                    obj.innerHTML = invocation.responseText;
                }
            };
            invocation.send();
        }
    </script>
</head>
<body onload="callOtherDomain('http://localhost:1113/cookies/cookies.php', 'output');">
<div align="center">
    <style>
        .post {margin-bottom: 20px;}
    </style>
    <div id="output"></div></div>
</body></html>
```

Figure 4.24 CORS Request for origin 1112 include credentials

Thus, when the client tries to submit the form using one of the cookies defined in the server, as shown in figure (4.25)



Enter Your Information

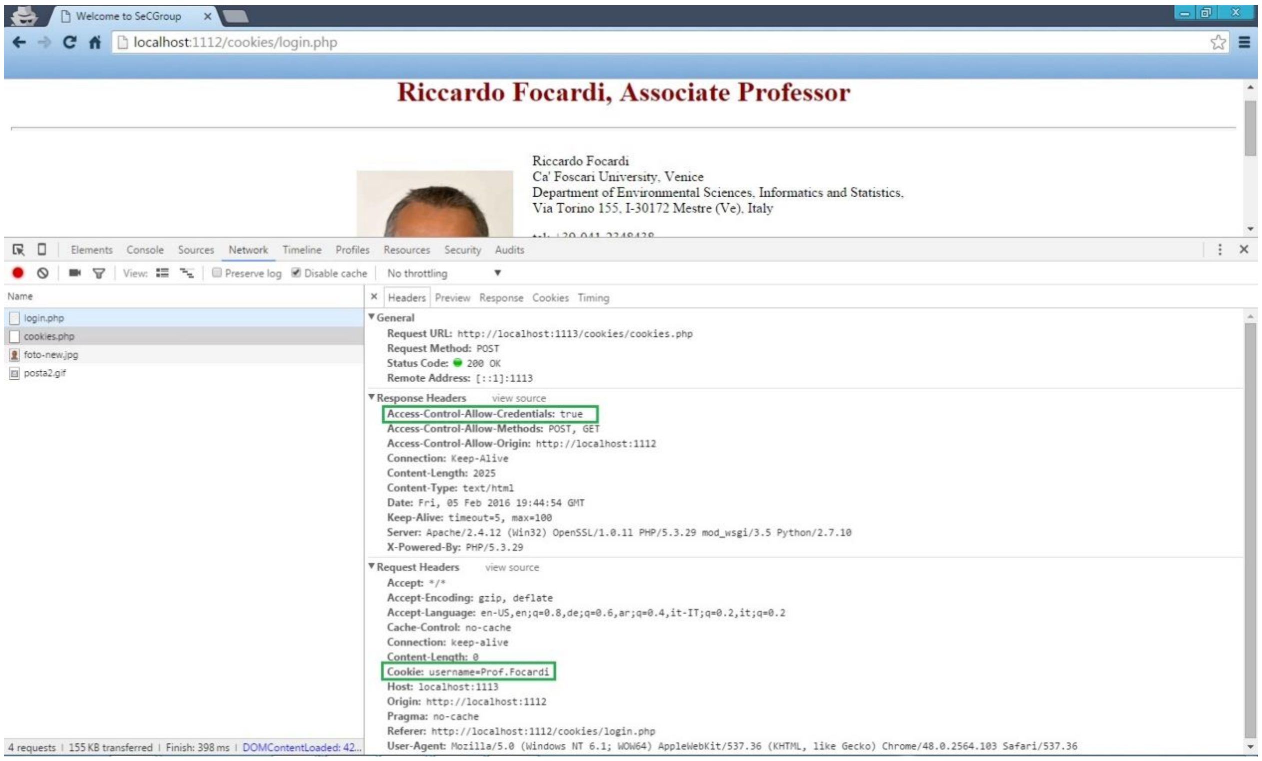
Username: Prof.Focardi

Password: Enter your password

Login

Figure 4.25 Login page of origin 1112

The function defined in the login page redirects the client to the page where the CORS request was made, as shown in figure(4.26).


Welcome to SeCGroup
localhost:1112/cookies/login.php
Riccardo Focardi, Associate Professor
Elements Console Sources Network Timeline Profiles Resources Security Audits
View: Preserve log Disable cache No throttling
Name
login.php
cookies.php
foto-new.jpg
posta2.gif
Headers Preview Response Cookies Timing
4 requests | 155 KB transferred | Finish: 398 ms | DOMContentLoaded: 42...

Riccardo Focardi
Ca' Foscari University, Venice
Department of Environmental Sciences, Informatics and Statistics,
Via Torino 155, I-30172 Mestre (Ve), Italy

```
▼General
   Request URL: http://localhost:1113/cookies/cookies.php
   Request Method: POST
   Status Code: ● 200 OK
   Remote Address: [::1]:1113
▼Response Headers     view source
   Access-Control-Allow-Credentials: true
   Access-Control-Allow-Methods: POST, GET
   Access-Control-Allow-Origin: http://localhost:1112
   Connection: Keep-Alive
   Content-Length: 2025
   Content-Type: text/html
   Date: Fri, 05 Feb 2016 19:44:54 GMT
   Keep-Alive: timeout=5, max=100
   Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1l PHP/5.3.29 mod_wsgi/3.5 Python/2.7.10
   X-Powered-By: PHP/5.3.29
▼Request Headers     view source
   Accept: */*
   Accept-Encoding: gzip, deflate
   Accept-Language: en-US,en;q=0.8,de;q=0.6,ar;q=0.4,it-IT;q=0.2,it;q=0.2
   Cache-Control: no-cache
   Connection: keep-alive
   Content-Length: 0
   Cookie: username=Prof.Focardi
   Host: localhost:1113
   Origin: http://localhost:1112
   Pragma: no-cache
   Referer: http://localhost:1112/cookies/login.php
   User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.103 Safari/537.36
```

Figure 4.26 Response of unauthenticated user in origin 1112

Finally, we can notice that the CORS request is successful and includes the cookies username, that is Prof. Focardi, so to retrieve information from his profile page.

CHAPTER 5

Conclusion

5.1 Review

In this thesis we have illustrated the main concepts behind websites interaction and we have revised typical attack scenarios. We have illustrated a new mechanism developed by W3C named Cross-Origin-Resource Sharing (CORS), and discussed the life cycle of this mechanism, by showing how different kind of requests are dealt with. We have experimented CORS mechanism by creating a toy site representing typical web application use cases.

5.2 Conclusions

We implemented our test cases using Apache server as a local host, and using PHP and JavaScript. One of the aims of our work was to give guidelines for web developers that intend to adopt CORS mechanism in their web applications, and to draw some remarks about the level of security achieved by the adoption of CORS.

In order to adopt CORS in web applications web developers should keep in mind that:

1. If the application will be accessible from the same origin, there is no need to implement CORS. By default, the browser allows requests based on Same origin policy (SOP) without any restrictions. Otherwise, if it will be accessible from different, trusted origins, developers can use the first method of CORS mechanism (i.e., simple request using allowed access origin), moreover, if the server does not include any sensitive data we can use the wildcard method which is the second method of CORS simple request.

2. In case of using a custom header or method different from the standard HTTP request, the preflight request is the solution in this case to prevent any kind of manipulation or leakage of sensitive information by providing what are the available method requests and the acceptable headers; in order to improve the performance of the server instead of responding twice for the preflight request and the actual request we can use the header Max-Age which makes the browser cash the preflight response in case it matches the defined parameters.
3. Finally, in order to make the application more secure CORS offers the credentials header that requires cookies to be included in the request, in order to check whether or not the request comes from a trusted user. If the request does not contain the credentials header, the request will be rejected.

Bibliography

[1] W3C, Same Origin Policy, https://www.w3.org/Security/wiki/Same_Origin_Policy

[2] OWASP, Cross-site Scripting, https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[3] Security Group, Ca'Foscari University, Cross-Site Scripting,

http://secgroup.dais.unive.it/teaching/security-course/cross-site-scripting-xss/

[4] OWASP, Cross-site Request Forgery,

https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

[5] Security Group, Ca'Foscari University, Cross-site Request Forgery

http://secgroup.dais.unive.it/teaching/security-course/cross-site-request-forgery-csrf/

[6] W3C, Cross-Origin Resource Sharing, https://www.w3.org/TR/cors/

[7] Mozilla Developer Network, HTTP access control (CORS) https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS