Lebanese American University

School of Arts and Sciences

Department of Computer Science and Mathematics

CSC435 (Computer Security)

Lab #03 (Web Attacks)

A Report Done By

Ahmad Hussein ahmad.hussein03@lau.edu

Mahmoud Joumaa mahmoud.joumaa@lau.edu

And presented to Dr. Ayman Tajeddine

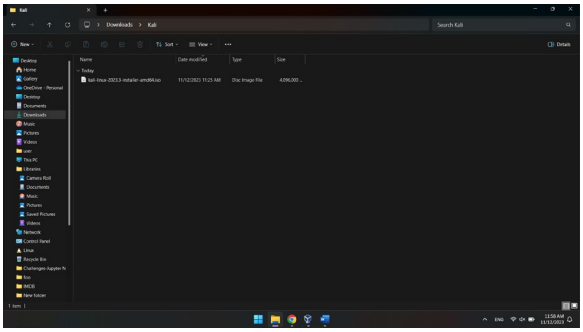
November 2023

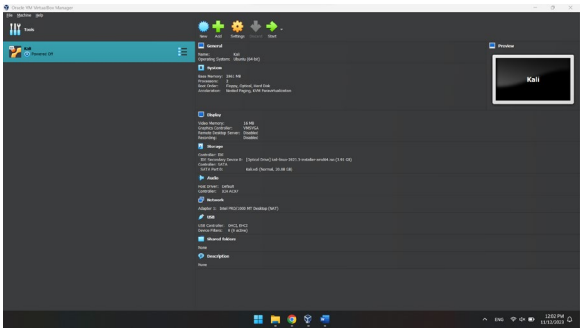
Table of Contents

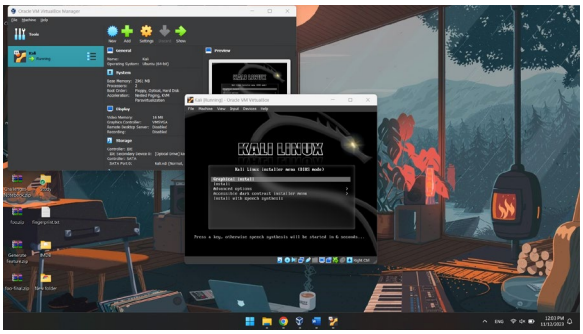
I. Prelab – Setting Up The Environment

1. Downloading and Setting Up a Virtual Machine

Before implementing the web attacks, we downloaded the virtual machine that we used for this lab. We opted for *Kali Linux* as it may be a good choice for setting up Damn Vulnerable Web Application (DVWA) as discussed in the next section.



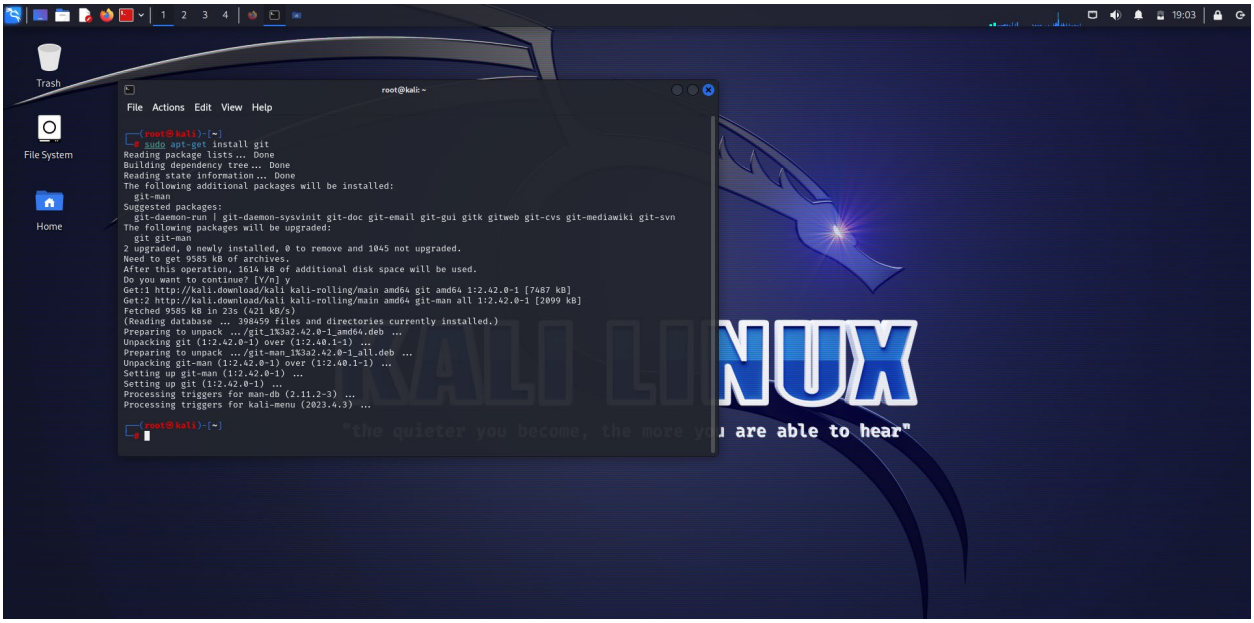




2. Downloading and Setting Up DVWA

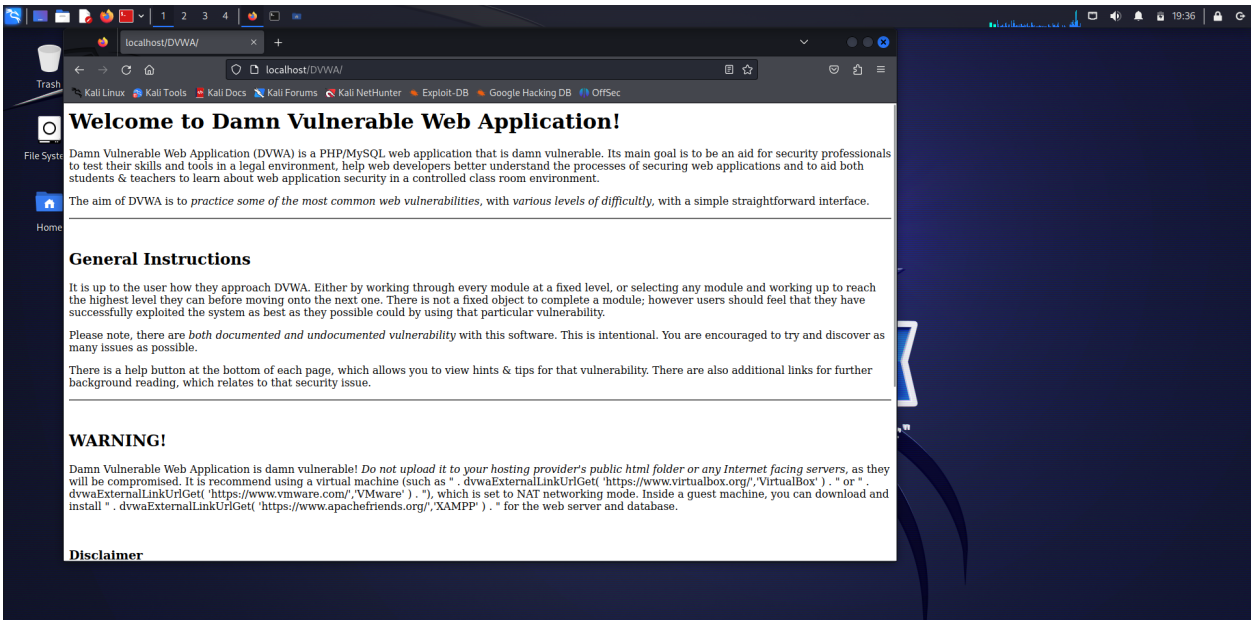
We first installed *apache*, which is the PHP web server to be utilized by DVWA.

Next, we installed *git* to be able to clone DVWA from GitHub.

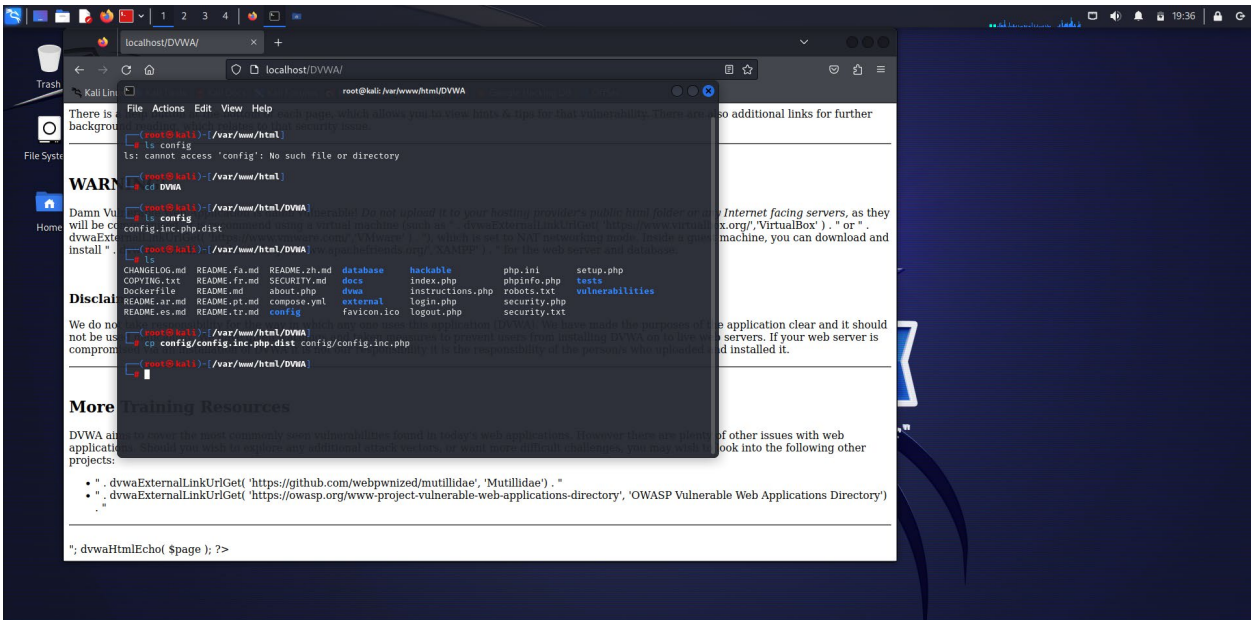


We then started the *apache2* service and installed DVWA using the *git clone* command.

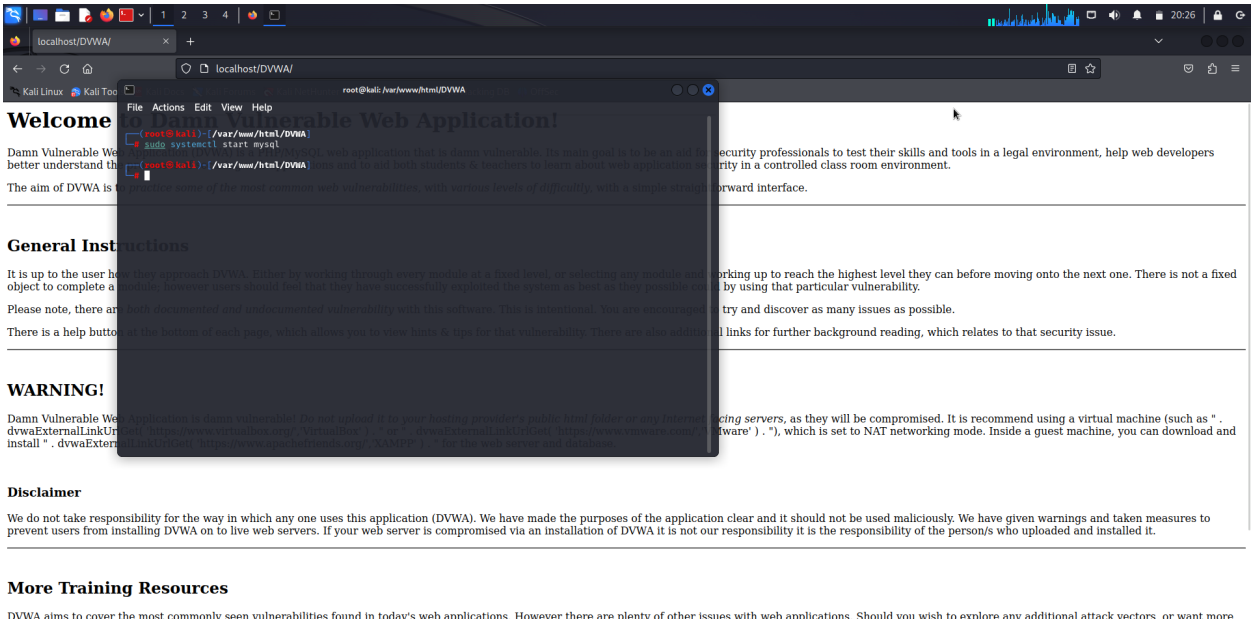
After those two commands completed executed, we navigated to *localhost/DVWA/* to make sure the executions were successful.



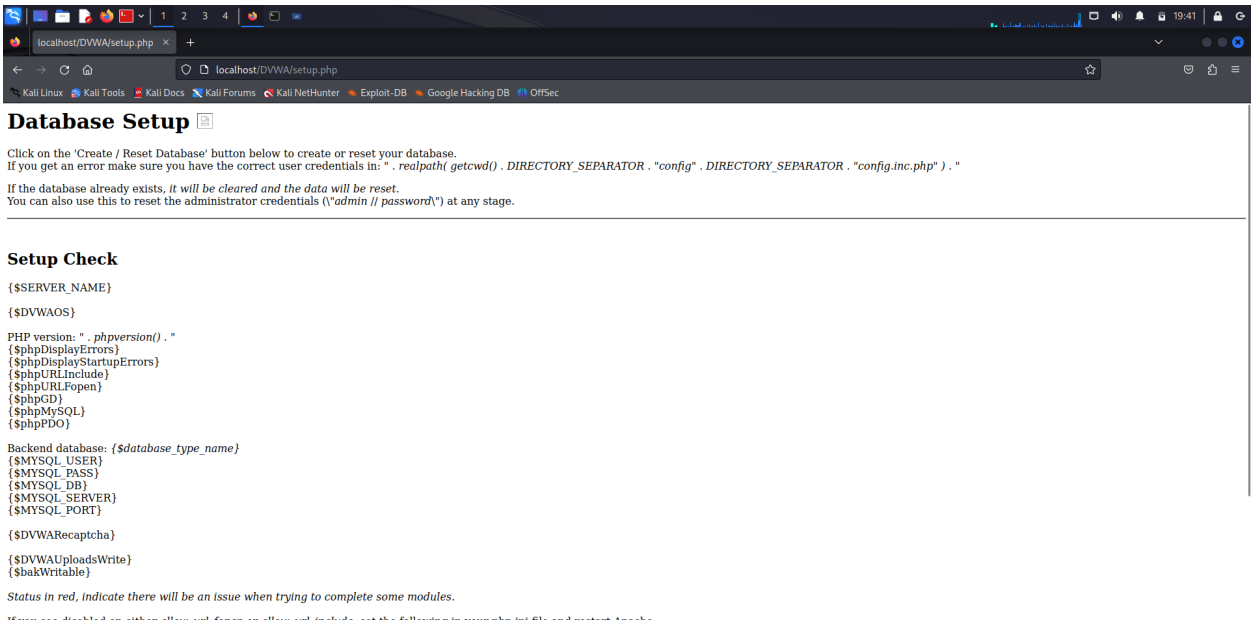
Following the provided tutorial (check Section VI: References), we copied the DVWA configuration file "*config.inc.php.dist*" as "*config.inc.php*".

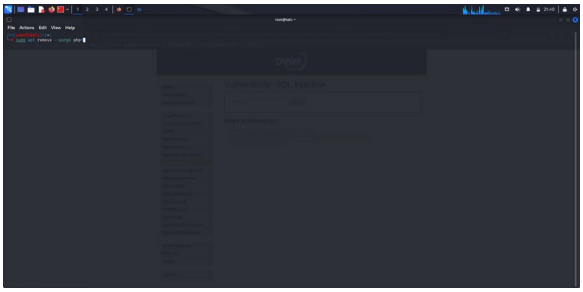


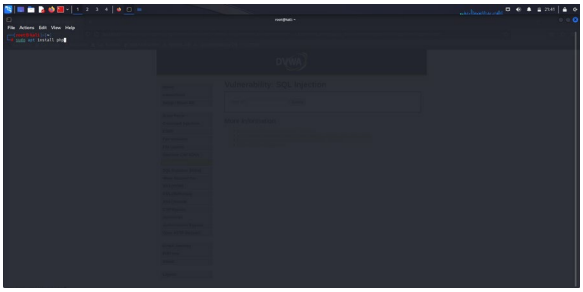
The next step was to start the *MySQL* service as shown in the next screenshot.



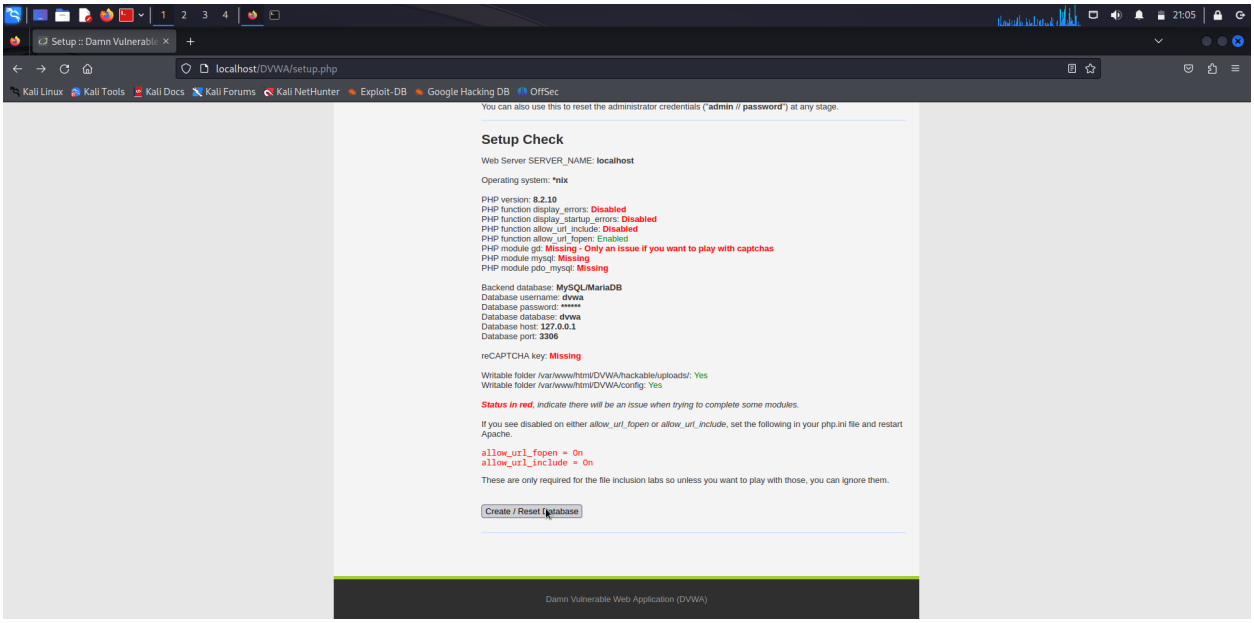
However, the pages seemed to be failing to link to the *CSS* styles, so we opted to uninstall *PHP* (using the *purge* command) and reinstall it (using the *install* command). This fixed the issue as shown in upcoming screenshots.





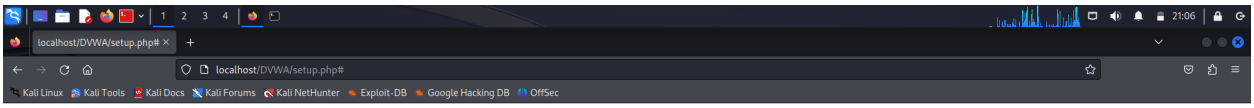


After fixing the styling issue, we navigated to *localhost/DVWA/setup.php* for a setup check.

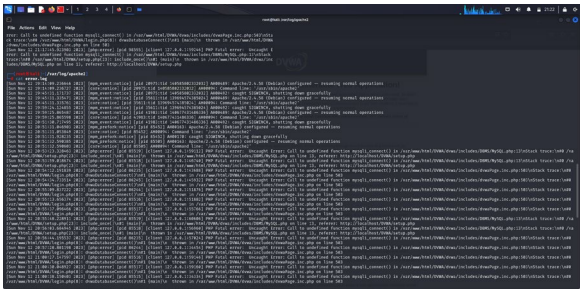


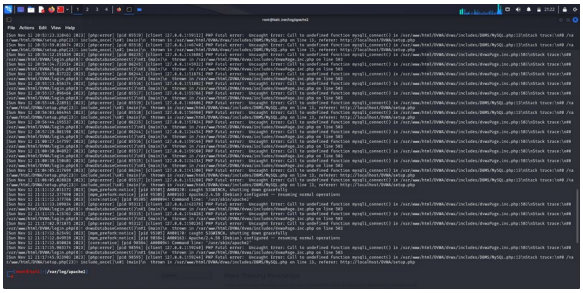
Following the provided tutorial, we connected to the database and executed the *flush privileges* command after creating the *DVWA* database.

Upon redirecting from the *setup.php* page and onto the login page, a white screen is shown, indicating an error.



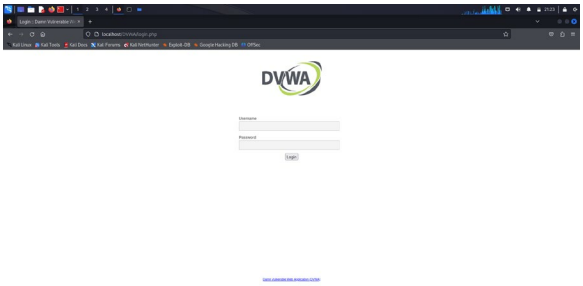
That is why we navigate to, and analyzed the contents of, the *apache2 log file* in an attempt to catch the error causing the white screen.

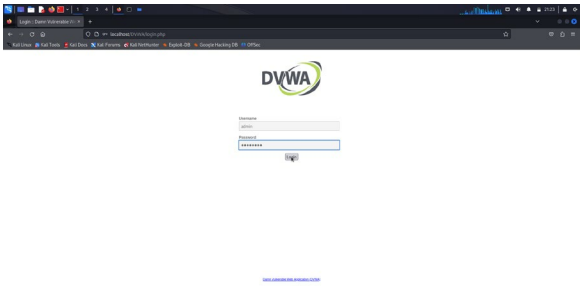


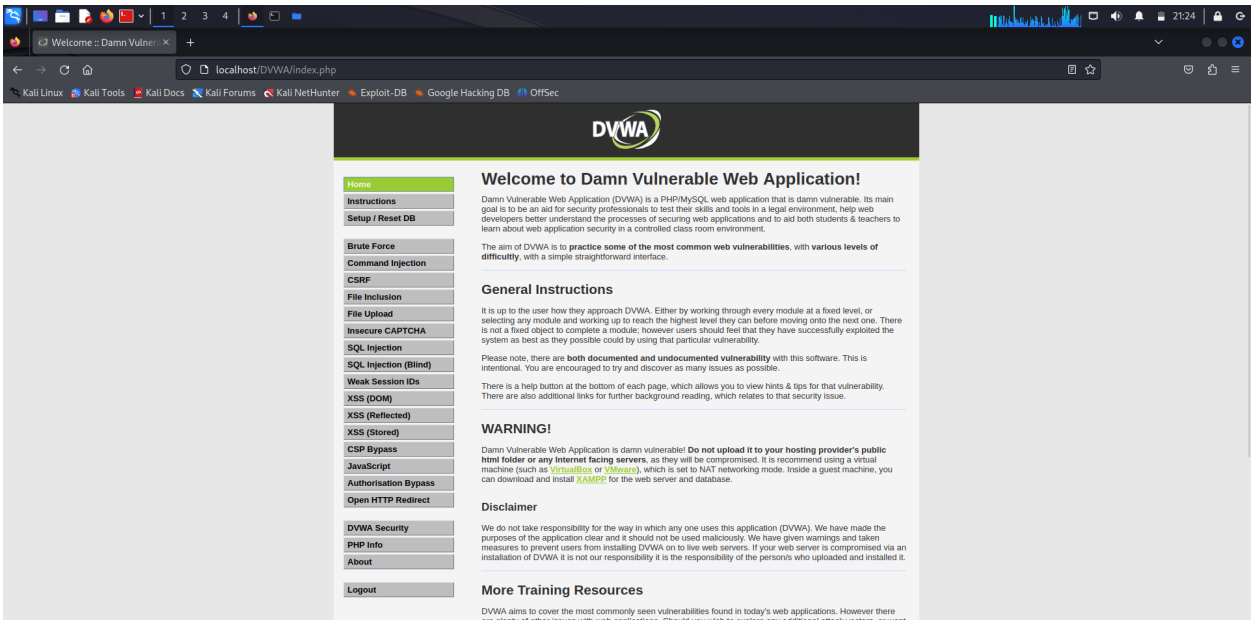


Upon inspection, we realized that the *php-mysql* extension was missing and consequently installed it using the *install php-mysql* command. This solved the issue as no error was then flagged.

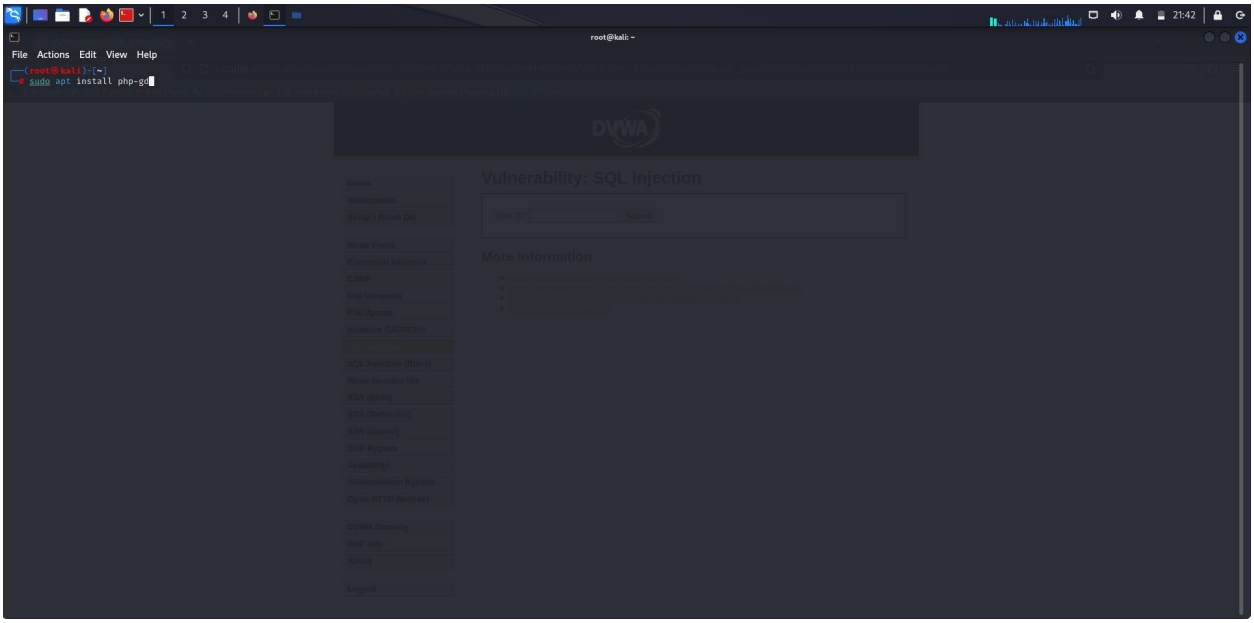

root@kali: /var/log/apache2
File Actions Edit View Help
[Sun Nov 12 21:17:32.849939 2023] [mpm_prefork:notice] [pid 98584] AH00163: Apache/2.4.58 (Debian) configured -- resuming normal operations
[Sun Nov 12 21:17:32.850026 2023] [core:notice] [pid 98584] AH00094: Command line: '/usr/sbin/apache2'
[Sun Nov 12 21:17:35.903376 2023] [php:error] [pid 98594] [client 127.0.0.1:59246] PHP Fatal error: Uncaught Error: Call to undefined function mysqli_connect() in /var/www/html/DVWA/dvwa/includes/dvwaPage.inc.php:503\nStack trace:\n#0 /var/www/html/DVWA/login.php(8): dvwaDatabaseConnect()\n#1 {main}\n thrown in /var/www/html/DVWA/dvwa/includes/dvwaPage.inc.php on line 503
[Sun Nov 12 21:17:45.923903 2023] [php:error] [pid 98595] [client 127.0.0.1:59246] PHP Fatal error: Uncaught Error: Call to undefined function mysqli_connect() in /var/www/html/DVWA/dvwa/includes/DBMS/MySQL.php:13\nStack trace:\n#0 /var/www/html/DVWA/setup.php(23): include_once()\n#1 {main}\n thrown in /var/www/html/DVWA/dvwa/includes/DBMS/MySQL.php on line 13, referer: http://localhost/DVWA/setup.php
(root㉿kali)-[/var/log/apache2]
#
(root㉿kali)-[/var/log/apache2]
sudo apt-get install php-mysql
Reading package lists ... Done
Building dependency tree ... Done
Reading state information ... Done
The following packages were automatically installed and are no longer required:
ettercap-common ettercap-graphical libluajit-5.1-2 libluajit-5.1-common python3-qrcode
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
php8.2-mysql
The following NEW packages will be installed:
php-mysql php8.2-mysql
0 upgraded, 2 newly installed, 0 to remove and 1036 not upgraded.
Need to get 121 kB of archives.
After this operation, 472 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://kali.download/kali kali-rolling/main amd64 php8.2-mysql amd64 8.2.10-2 [117 kB]
Get:2 http://http.kali.org/kali kali-rolling/main amd64 php-mysql all 2:8.2+93 [3660 B]
Fetched 121 kB in 1s (82.2 kB/s)
Selecting previously unselected package php8.2-mysql.
(Reading database ... 398132 files and directories currently installed.)
Preparing to unpack .../php8.2-mysql_8.2.10-2_amd64.deb ...
Unpacking php8.2-mysql (8.2.10-2) ...
Selecting previously unselected package php-mysql.
Preparing to unpack .../php-mysql_2%3a8.2+93_all.deb ...
Unpacking php-mysql (2:8.2+93) ...
Setting up php8.2-mysql (8.2.10-2) ...
Creating config file /etc/php/8.2/mods-available/mysqlnd.ini with new version
Creating config file /etc/php/8.2/mods-available/mysqli.ini with new version
Creating config file /etc/php/8.2/mods-available/pdo_mysql.ini with new version
Setting up php-mysql (2:8.2+93) ...
Processing triggers for libapache2-mod-php8.2 (8.2.10-2) ...
Processing triggers for php8.2-cli (8.2.10-2) ...
(root㉿kali)-[/var/log/apache2]
sudo systemctl restart apache2
(root㉿kali)-[/var/log/apache2]
#

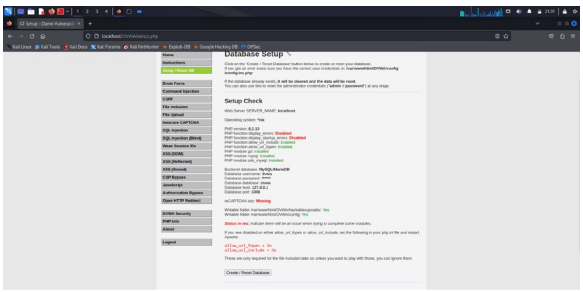

DVWA

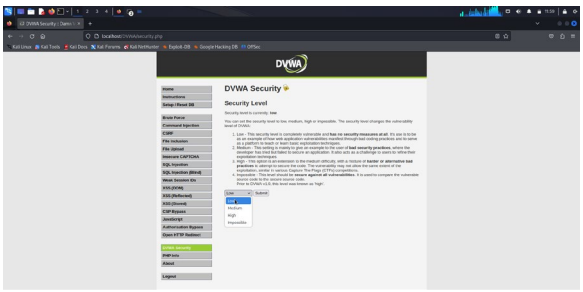

DVWA


Welcome :: Damn Vulner
localhost/DVWA/index.php
Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB OffSec
DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass
JavaScript
Authorisation Bypass
Open HTTP Redirect
DVWA Security
PHP Info
About
Logout
Welcome to Damn Vulnerable Web Application!
Damn Vulnerable Web Application (DVWA) is a PHP/MySQL web application that is damn vulnerable. Its main goal is to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and to aid both students & teachers to learn about web application security in a controlled class room environment.
The aim of DVWA is to practice some of the most common web vulnerabilities, with various levels of difficultly, with a simple straightforward interface.
General Instructions
It is up to the user how they approach DVWA. Either by working through every module at a fixed level, or selecting any module and working up to reach the highest level they can before moving onto the next one. There is not a fixed object to complete a module; however users should feel that they have successfully exploited the system as best as they possible could by using that particular vulnerability.
Please note, there are both documented and undocumented vulnerability with this software. This is intentional. You are encouraged to try and discover as many issues as possible.
There is a help button at the bottom of each page, which allows you to view hints & tips for that vulnerability. There are also additional links for further background reading, which relates to that security issue.
WARNING!
Damn Vulnerable Web Application is damn vulnerable! Do not upload it to your hosting provider's public html folder or any Internet facing servers, as they will be compromised. It is recommend using a virtual machine (such as VirtualBox or VMware), which is set to NAT networking mode. Inside a guest machine, you can download and install XAMPP for the web server and database.
Disclaimer
We do not take responsibility for the way in which any one uses this application (DVWA). We have made the purposes of the application clear and it should not be used maliciously. We have given warnings and taken measures to prevent users from installing DVWA on to live web servers. If your web server is compromised via an installation of DVWA it is not our responsibility it is the responsibility of the person/s who uploaded and installed it.
More Training Resources
DVWA aims to cover the most commonly seen vulnerabilities found in today's web applications. However there

As a final step for setting up DVWA, we installed *php-gd* and, after running a final setup check, we set the security level to *low* in the *DVWA Security* tab, and are now ready to implement the different web attacks listed for this lab.



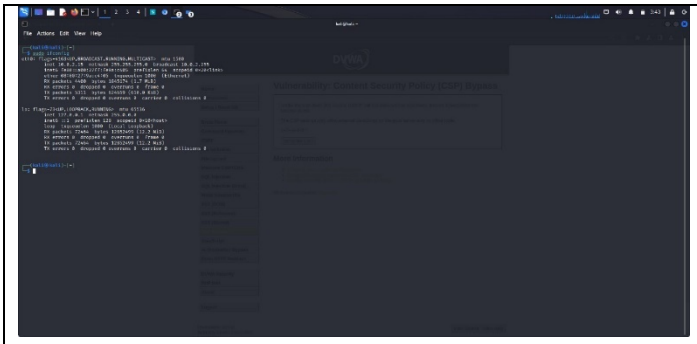
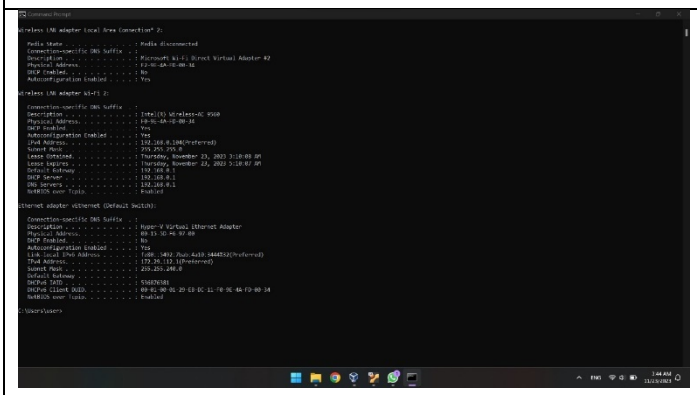




3. Downloading and Setting Up Other Software

Note that the following software discussed in this section were downloaded, during working on the lab, when we had reached their respective sections. The installation process is included here for ease of navigation and context flow.

Machine MAC and IP Addresses

	Virtual Machine MAC Address: 08:00:27:9a:c4:05 IP Address: 10.0.2.15
	Host Machine MAC Address: F0-9E-4A-FD-00-34 IP Address: 192.168.0.104

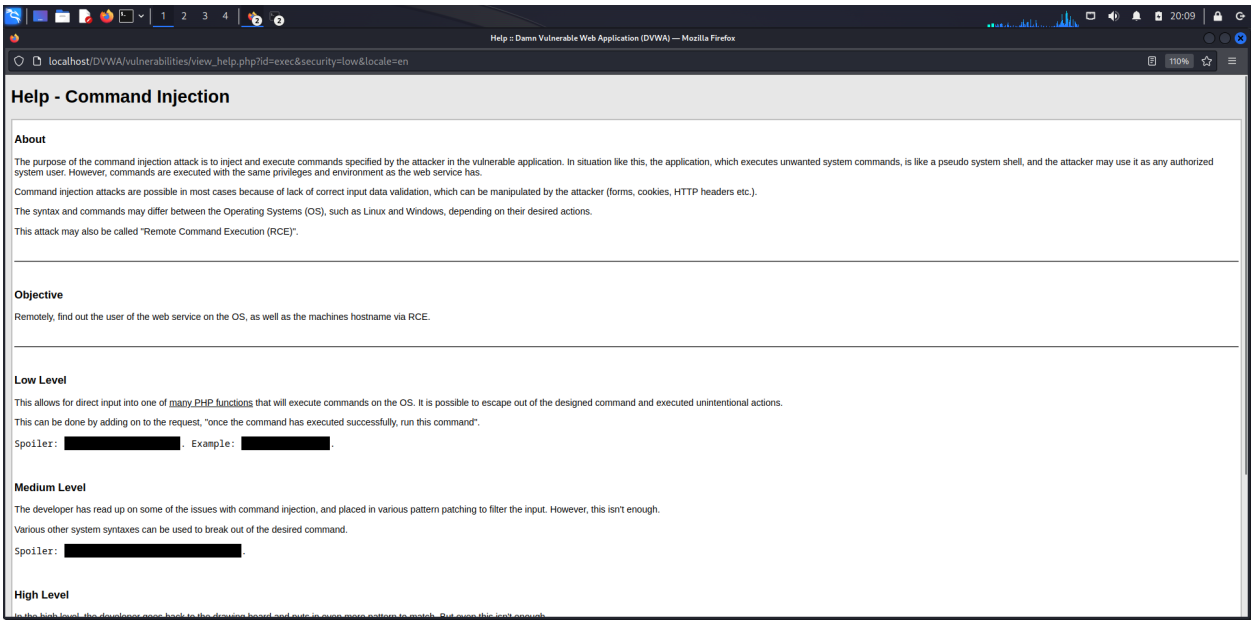
For the following sections (II through IV), it is important to note that the impossible level serves as a reference point for best practices to protect against such attacks, thus rendering it *impossible* to crack.

II. Command Execution and SQL Injection Challenges

1. Command Injection

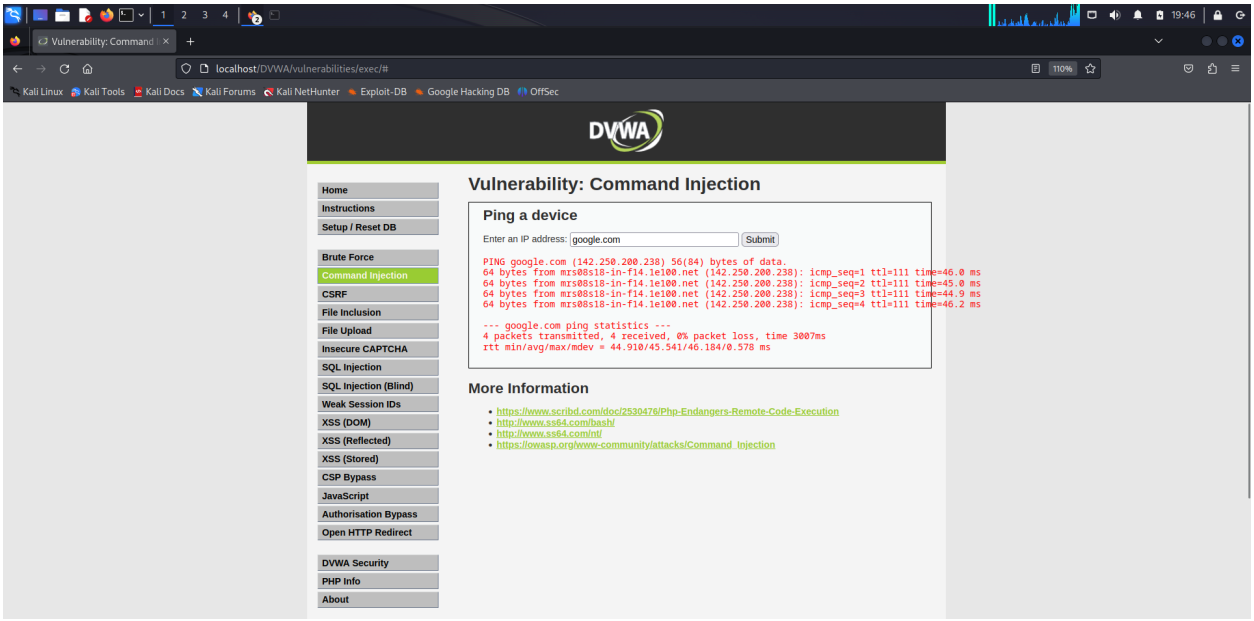
Referring to DVWA's *View Help* option, our objective is to find the *user* of the web service as well as the *hostname* of the machine via Remote Code Execution (RCE).

Since the commands execute with the same privilege as the web service, the victim would be vulnerable to any command that could be executed within the provided privilege, including listing files, manipulating directories, or installing and deleting applications or services.

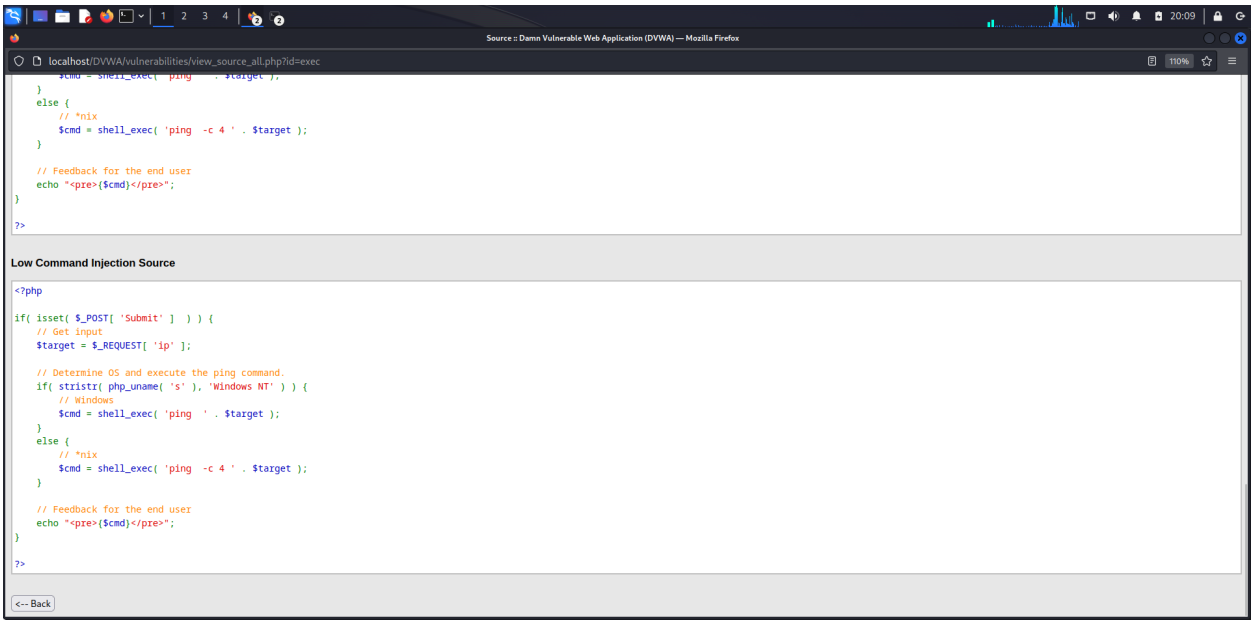


a. Low

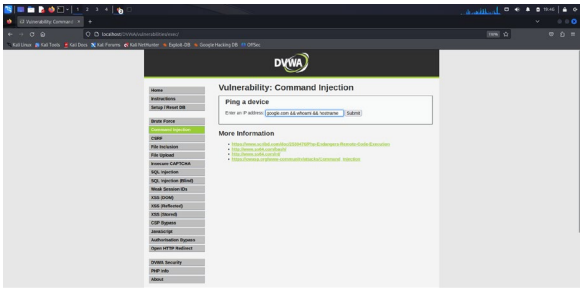
Before attempting the attack, we checked the page's behavior against expected input.

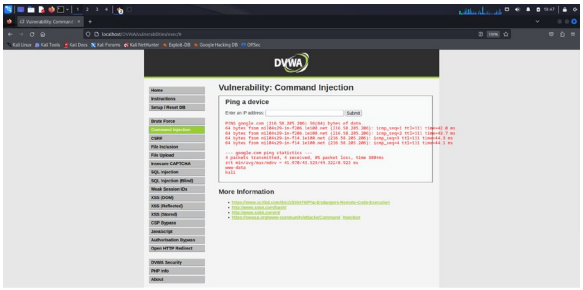


We then inspected the following provided source code:



It shows that no checks are being performed on the recorded input, so we can simply concatenate the *whoami* command (which shows the user) and *hostname* command (which shows the machine's hostname) to the original command of pinging a target address.

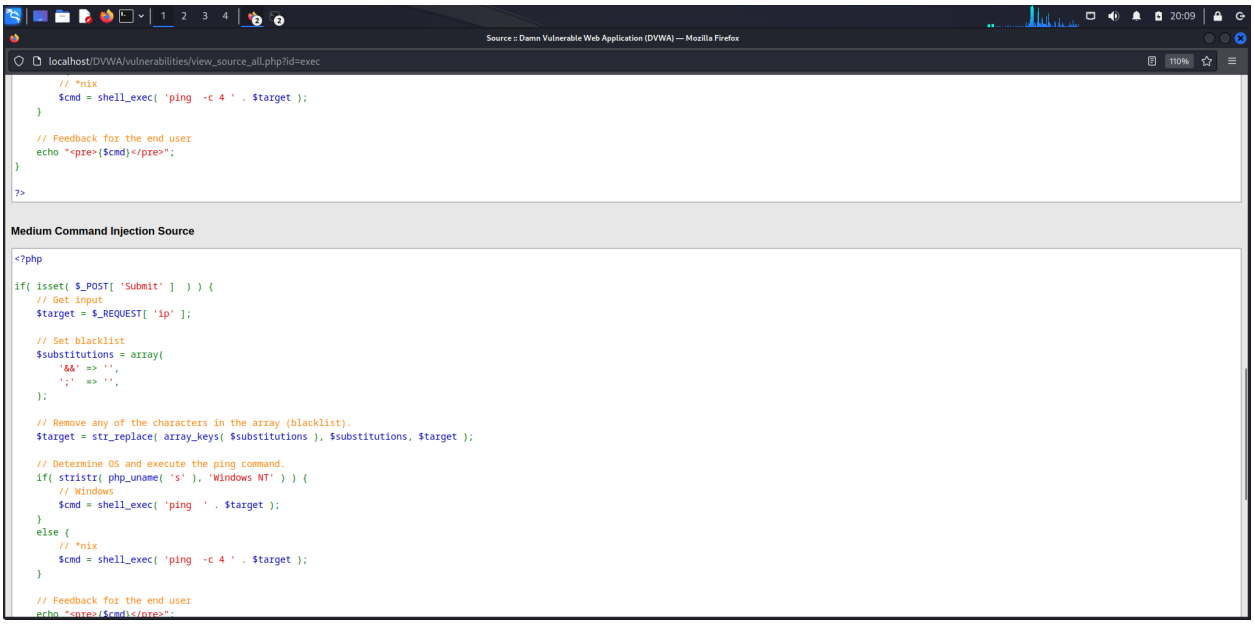




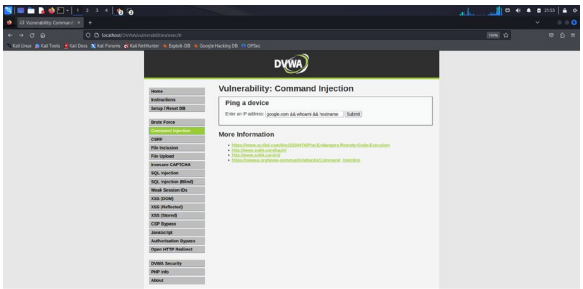
The output revealing *user www-data* and *hostname kali* confirms our successful command execution attack.

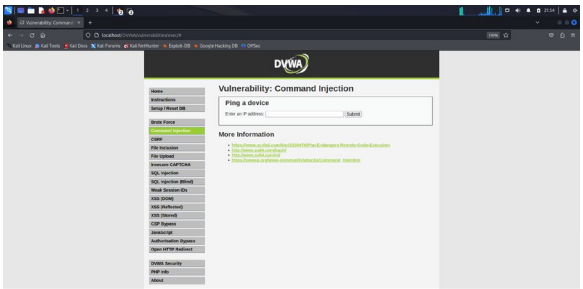
b. Medium

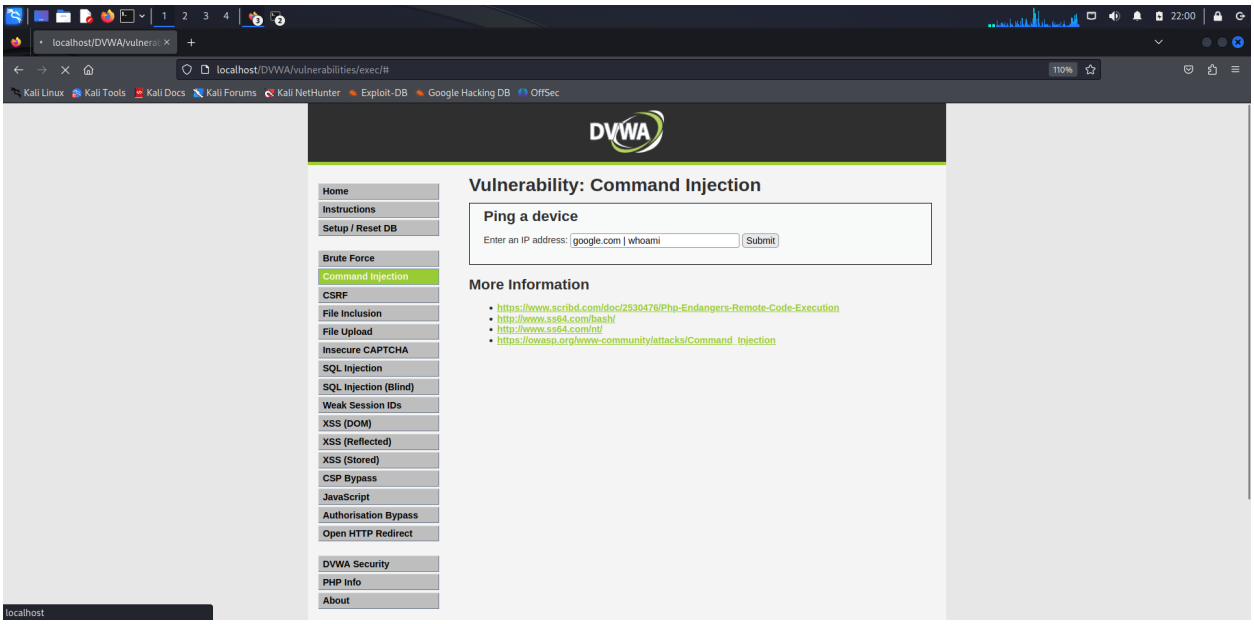
This level's source code is provided by the following:

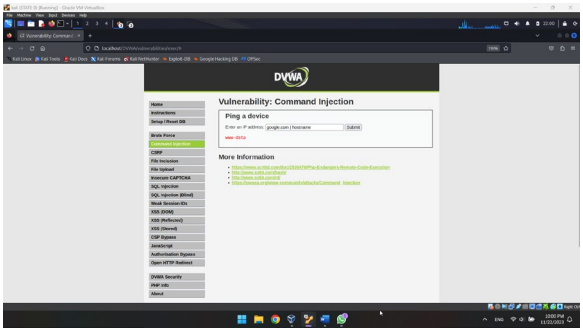


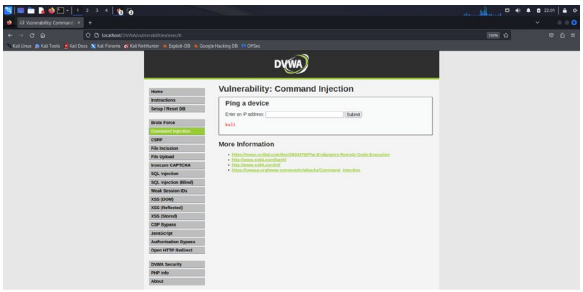
It shows that the concatenating characters *&&* and *;* are being substituted in the recorded input by an empty string. However, the concatenating character | were not being checked for substitution, which is why we can re-execute the attack using the same command but using '|' as a separator instead of '*&&*'. This limits the number of commands we can chain together, so we executed each command separately as shown in the below screenshots after testing the page's behavior against inputs containing '*&&*'.







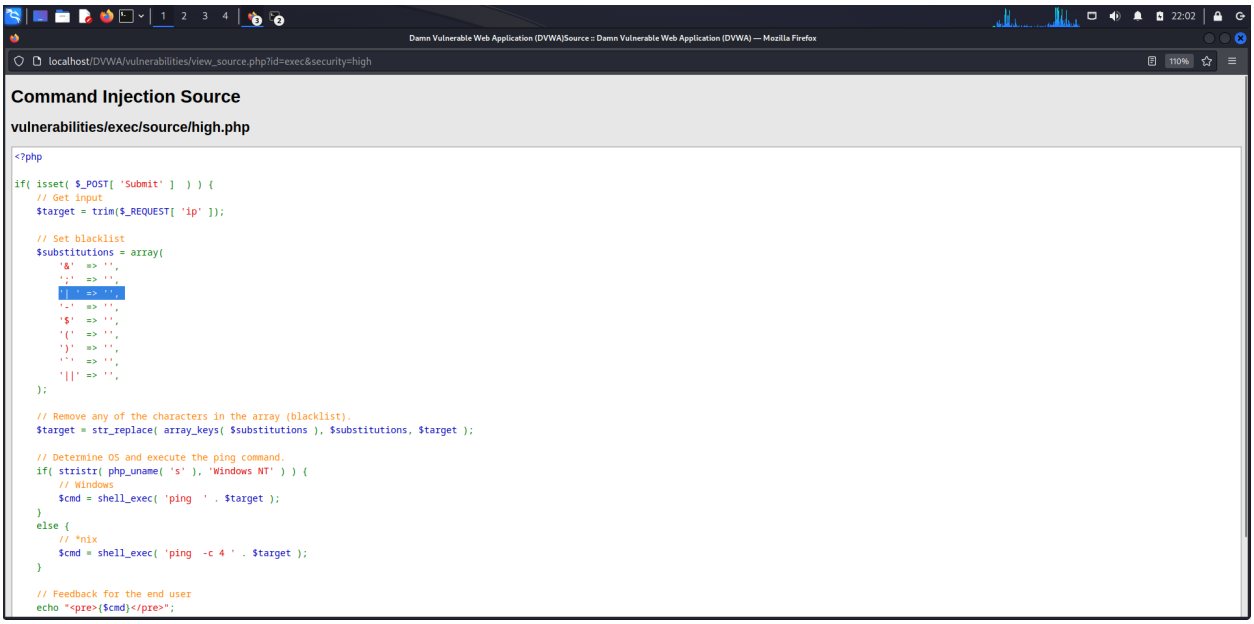




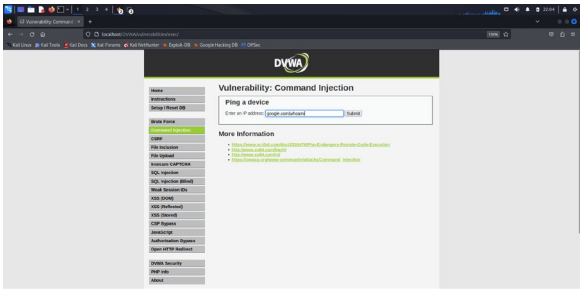
The outputs showing *user www-data* and *hostname kali* confirm our successful command execution attack.

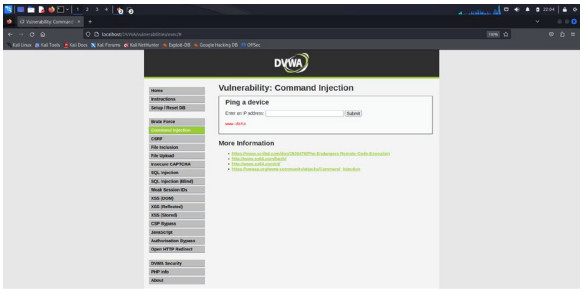
c. High

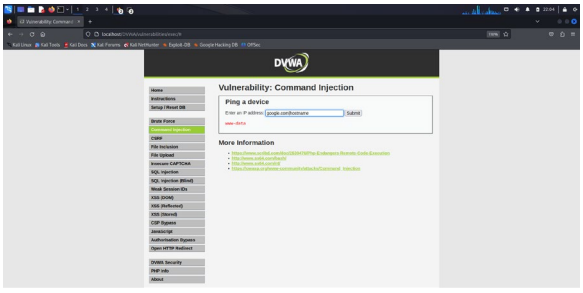
For this level, the developer has attempted to filter out all possible command separators by replacing each of this with an empty string. This can be shown in this level's provided source code:

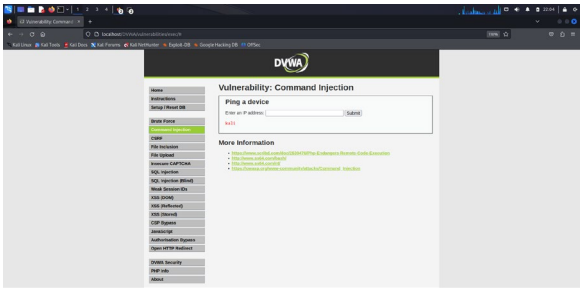


However, there's a mistyped separator '|' which includes an extra space character afterwards. This indicates that the '|' separator is not filtered out and can be used naturally within the commands to execute. Therefore, our solution for the medium-level challenge can be reapplied here but paying attention to have no spaces after the '|' separator.



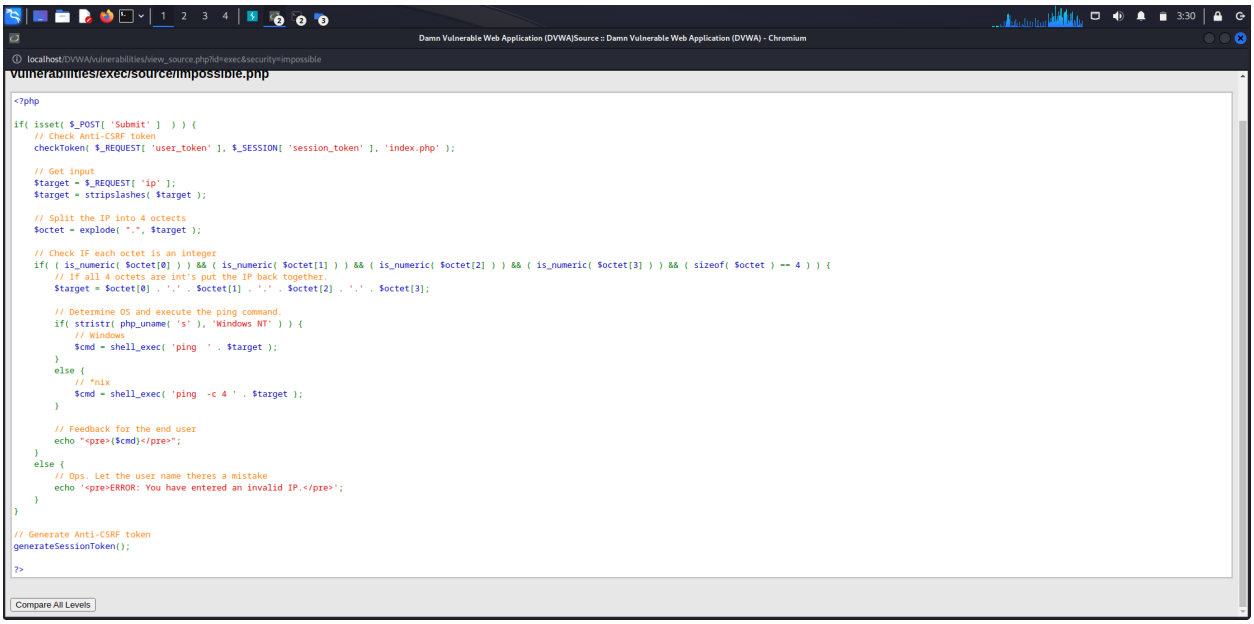






The *user* and *hostname* are shown to be *www-data* and *kali* respectively, which confirms a successful command execution attack.

d. Impossible



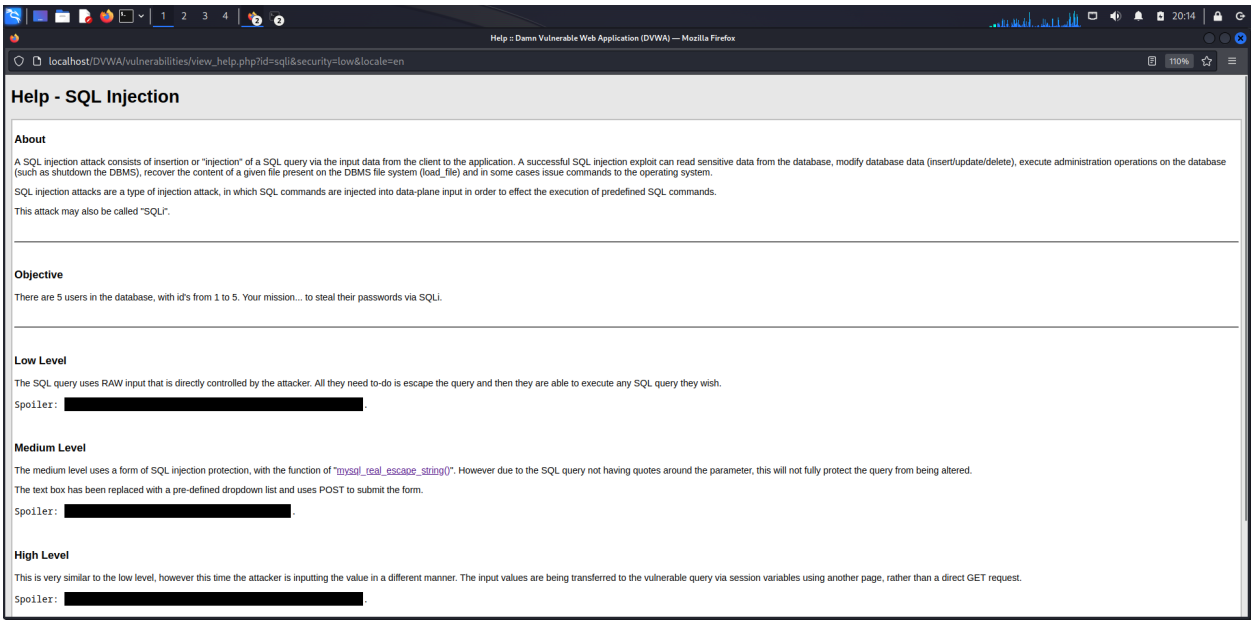
This code first removes any backslashes (\) from the input inserted by the user. It then breaks that input string on (.) into an array and makes sure that there are 4 numeric entries in the array (i.e. the parts that form the IP address). If those conditions are satisfied, the address is being pinged. Nothing happens otherwise.

By validating the format of the input (parts of the IP address), and that those parts are indeed integers, this code is virtually immune to typical command execution attacks.

2. SQL Injection (SQLi)

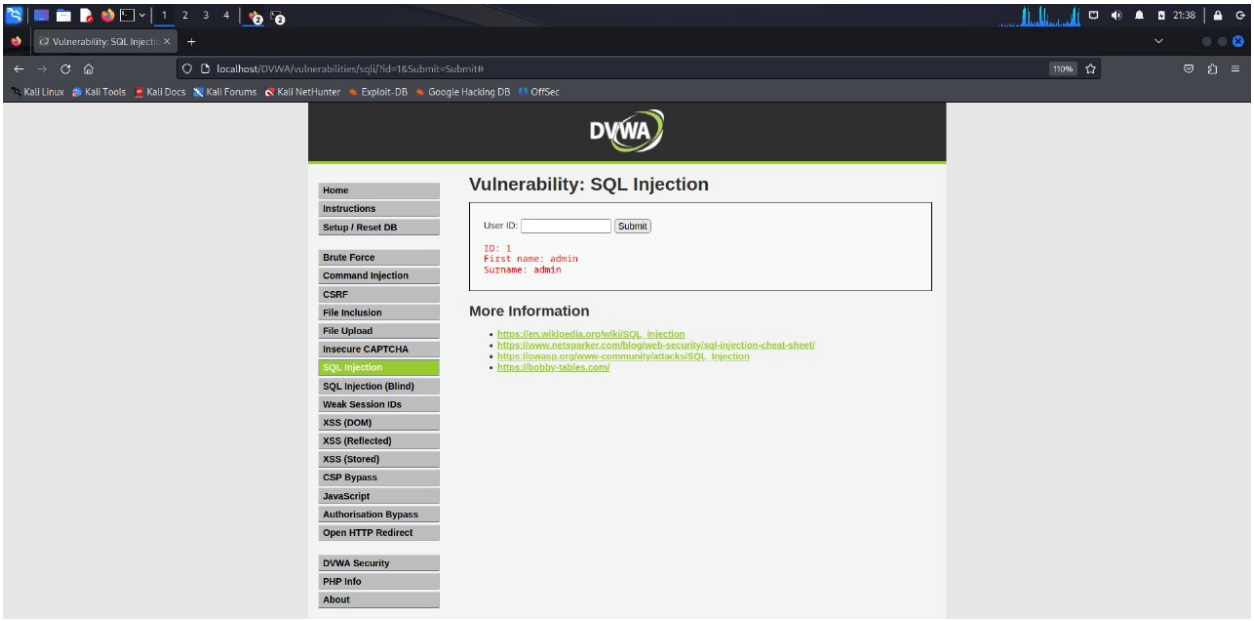
Referring to DVWA's *View Help* option, our objective is to steal the 5 users' passwords stored in the database using SQLi, which is a form of an injection attack consisting of injecting piggybacked SQL queries via user input means (such as HTML form textboxes).

The victims would be susceptible to anything their database is susceptible to. Examples range from data leaks, to data insertions and deletions or manipulation.

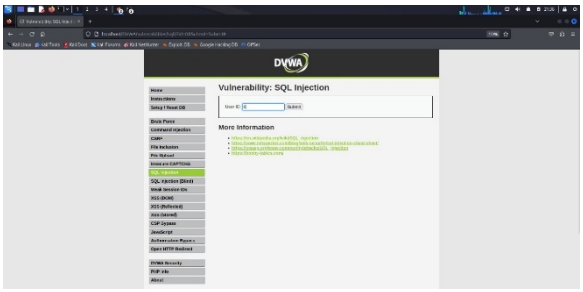


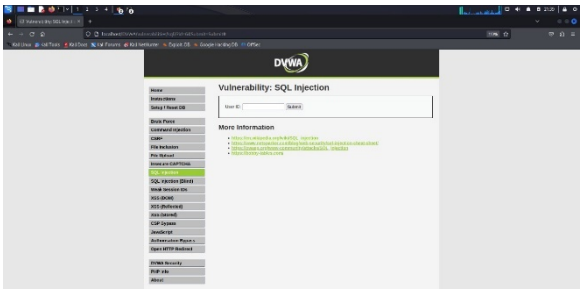
a. Low

Before beginning the attack, we tested to check the service's response on a valid input (existing user ID)

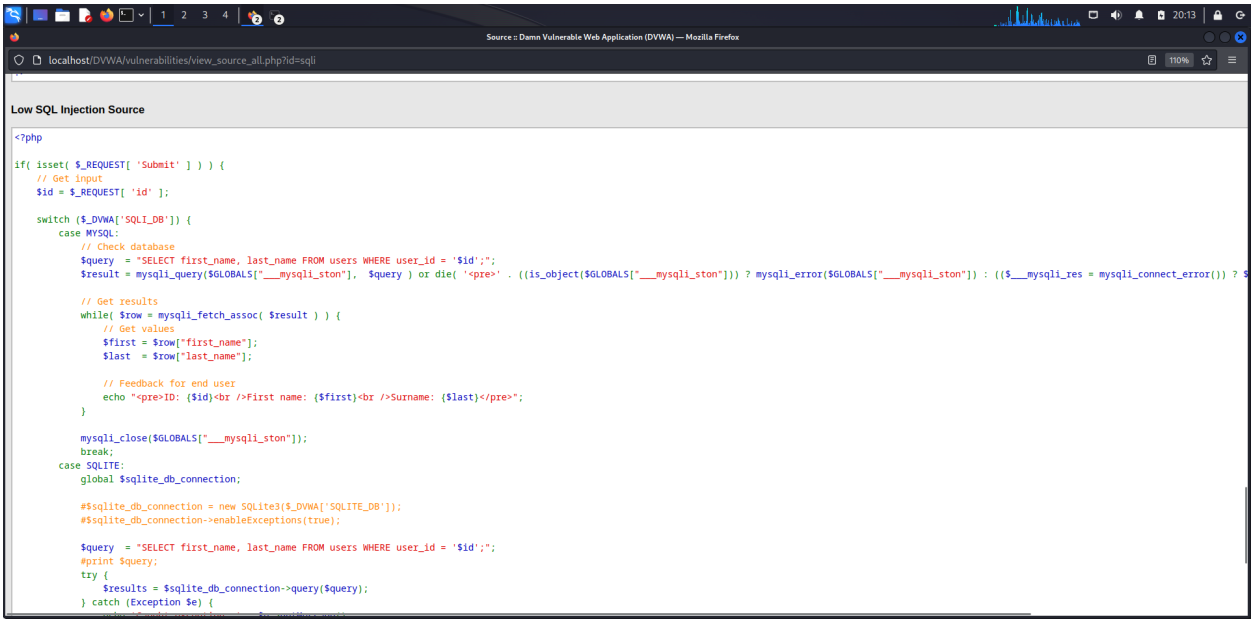


as well as its response on an invalid input (a user ID that does not exist in the database).

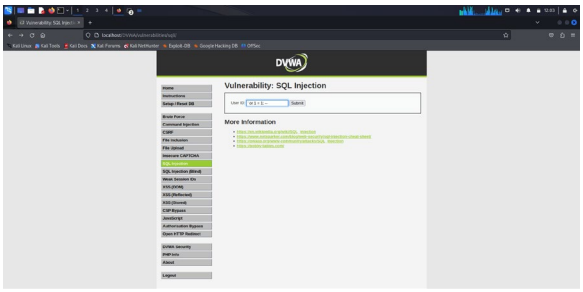




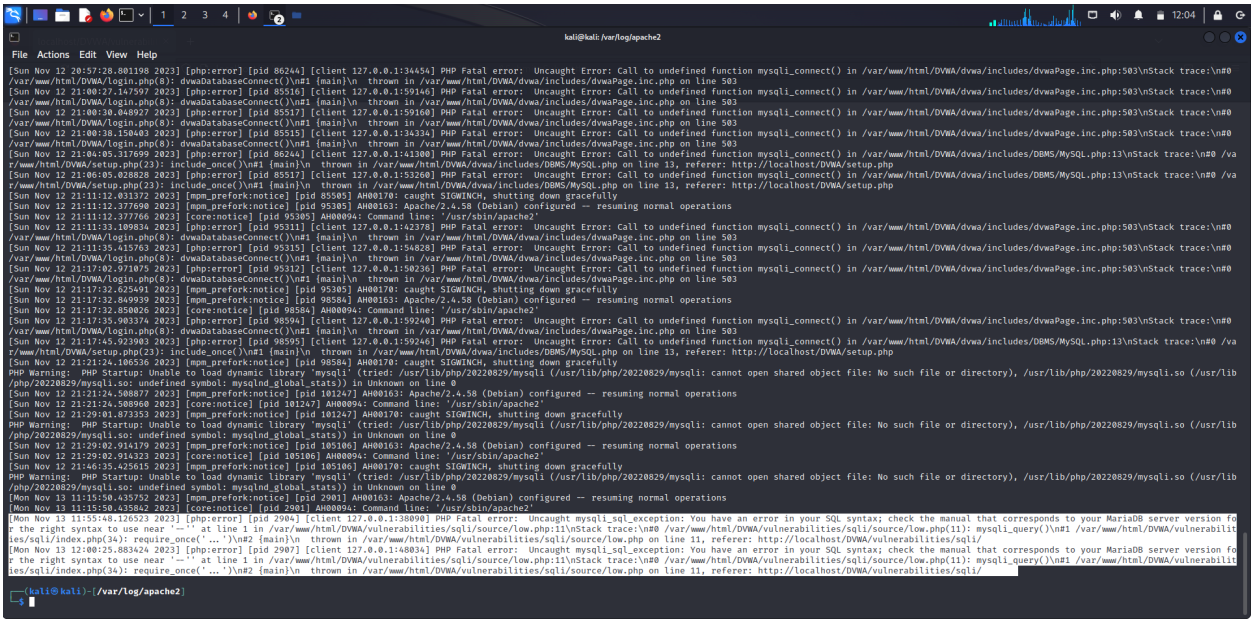
This level's source code is provided by the following:



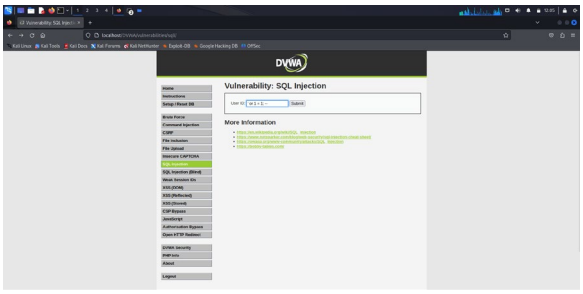
The query to be executed remains vulnerable as the *$id* is enclosed in quotes. This allows us to terminate the provided command early, append a tautology to it, and finally comment the rest of the line to avoid any potential syntax errors. To check for this vulnerability, we tested the exploit using the input *' or 1=1; --*.

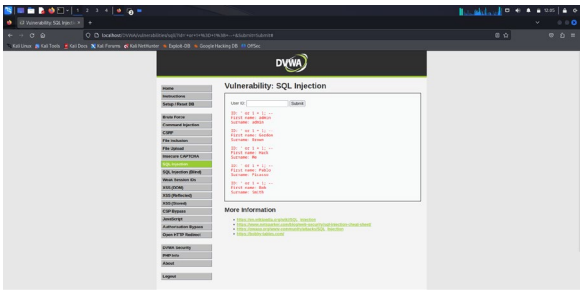


The output is different than the one we expected as the service redirected to a white screen. Upon inspection of the logs once again, and with a little research, we discovered that a space ‘ ‘ is mandatory after the -- in the comment line.



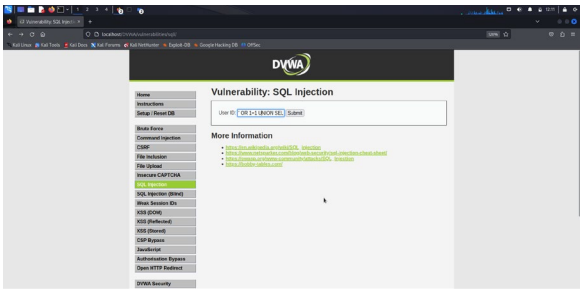
We then retried the query, keeping the comment syntax in mind, and were able to confirm the vulnerability.

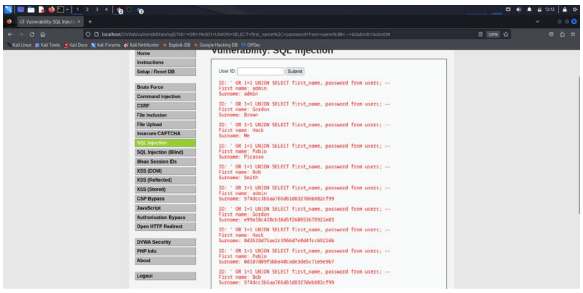




To steal the users’ passwords, and knowing that the query returns two columns from the database, we opted to join the provided query with our injected one using the following input:

‘ OR 1=1 UNION SELECT first_name, password FROM users; --

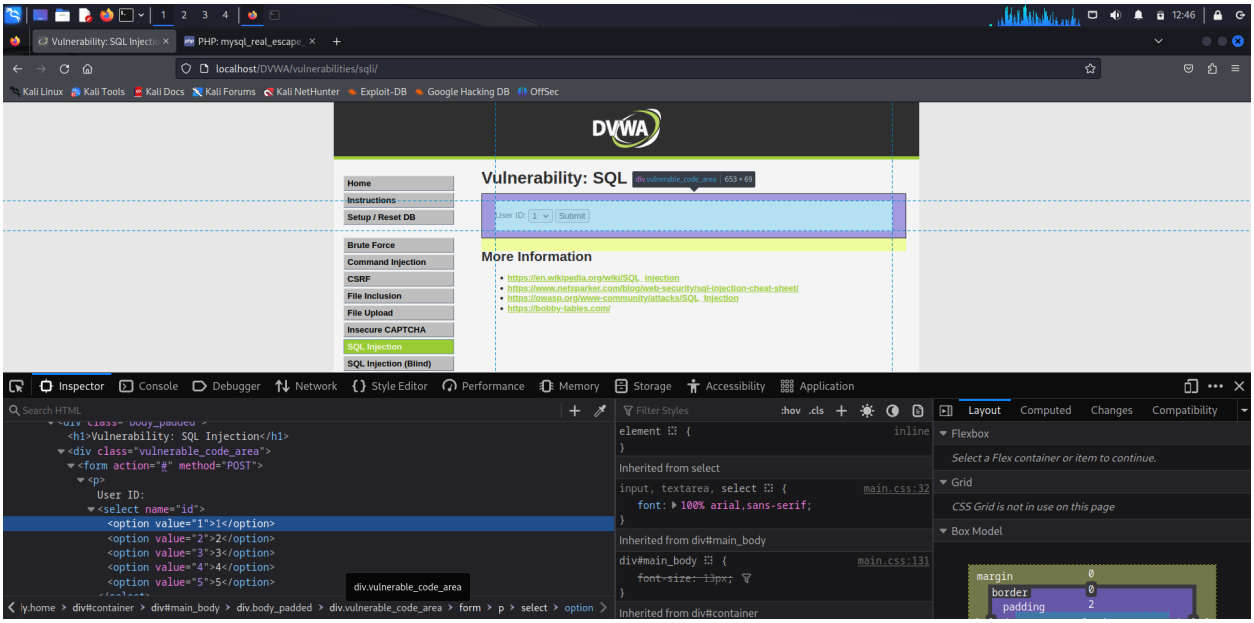




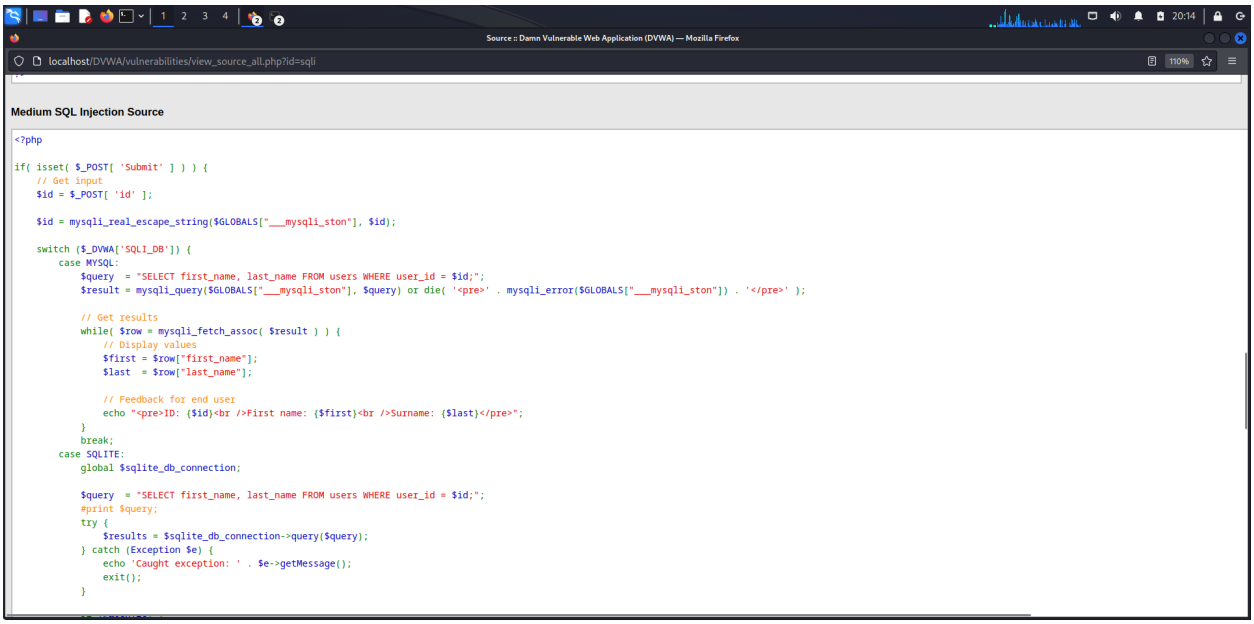
The revealed hashed passwords in the output indicate a successful SQLi attack.

b. Medium

In an attempt to restrict user input, the ids are provided in a dropdown for the user to select from.

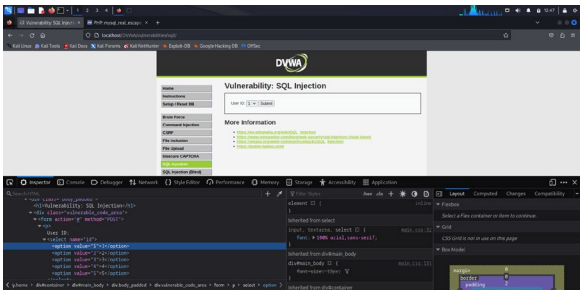


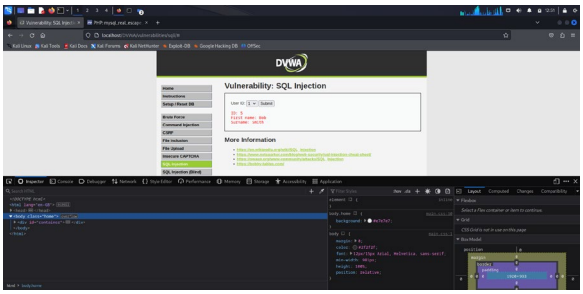
This level's source code is provided by the following:



The *$id* variable is not enclosed by quotes, which allows us to exploit this vulnerability by simply changing the *value* of the *option* tag using the *Inspect Elements* tools provided by the *Firefox* browser.

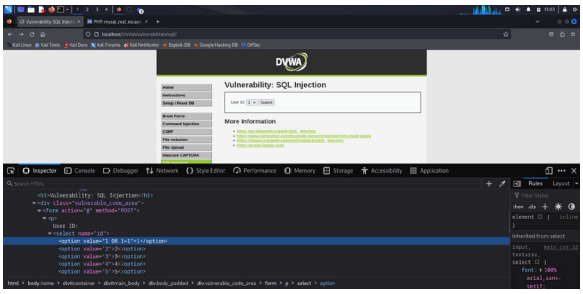
To test this exploit, we attempted to submit a value of 5 when selecting the option of value 1. The output confirms that we can submit values other than those provided in the dropdown by manipulating the elements using the inspect tool.

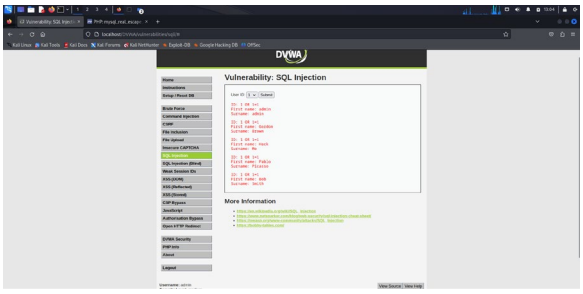




We also attempted appending a tautology by submitting the value: *1 OR 1=1*

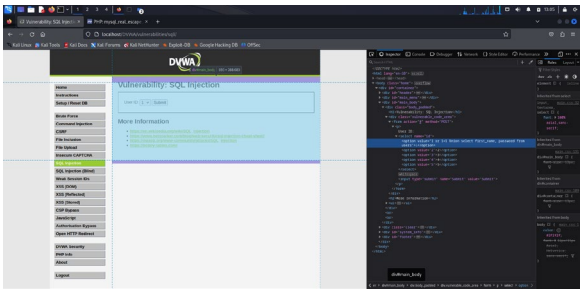
The output displays the first and surnames of all users in the database.

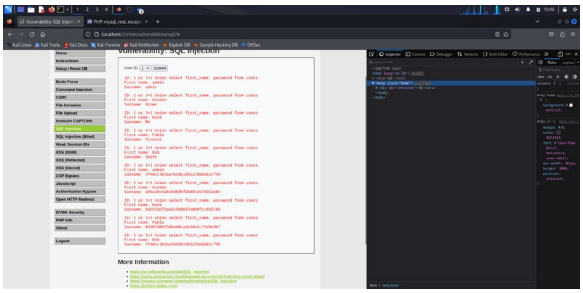




To extract the passwords of those users, we submitted the form with the following value:

1 OR 1=1 UNION SELECT first_name, password FROM users

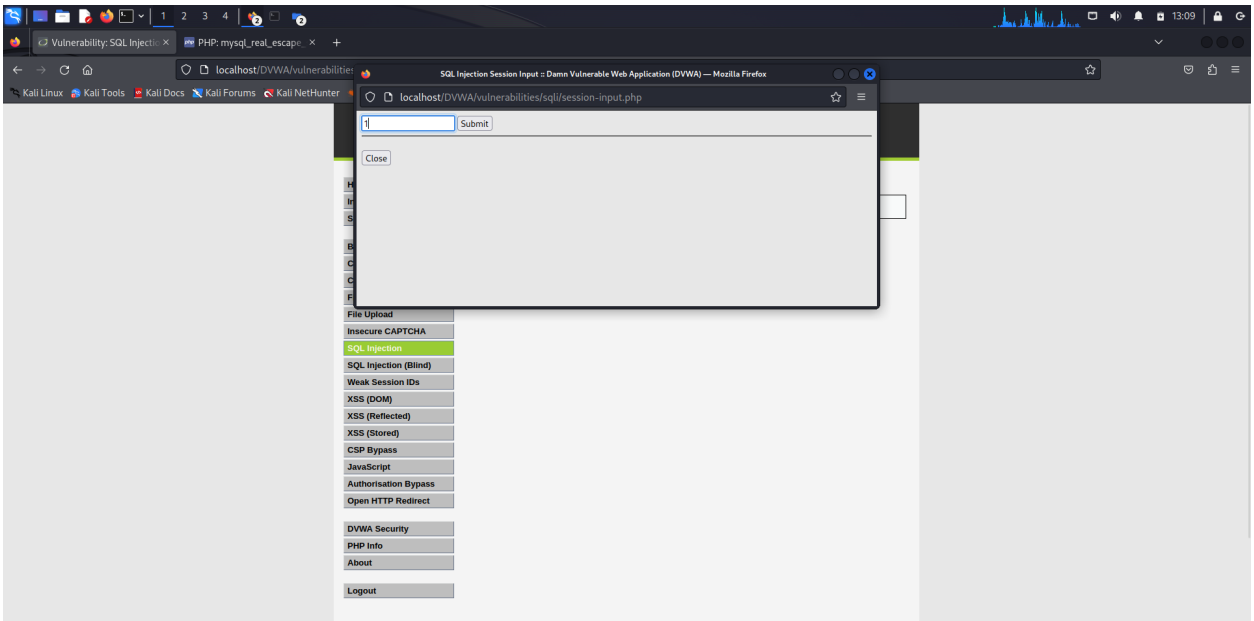


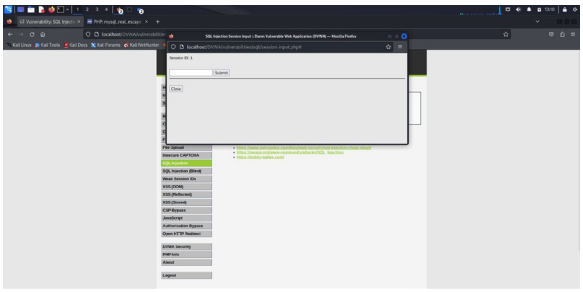


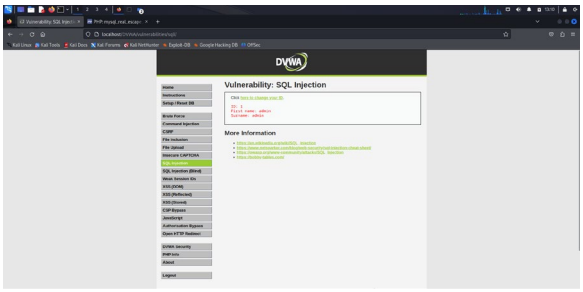
The output diaplying the first name and password of every user in the database indicates another successful SQLi attack.

c. High

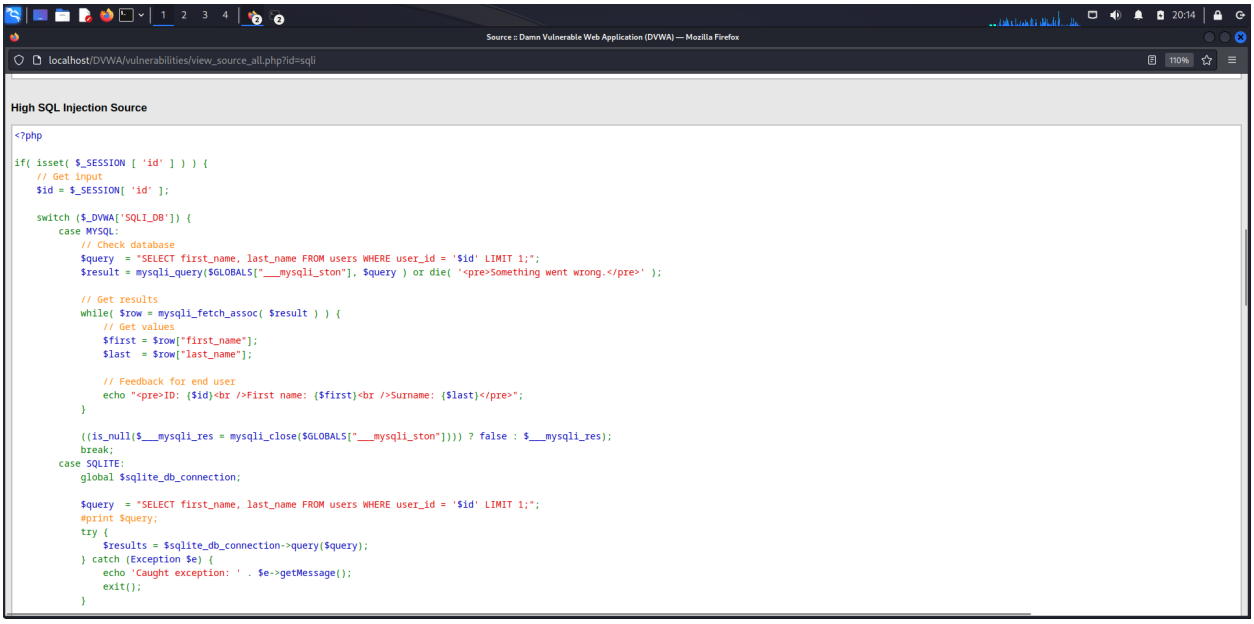
This level is similar to *low*, but records user input in a separate window on another page instead of direct GET requests. We then attempted to check the page's behavior regarding valid input (ID exists in the database: 1).



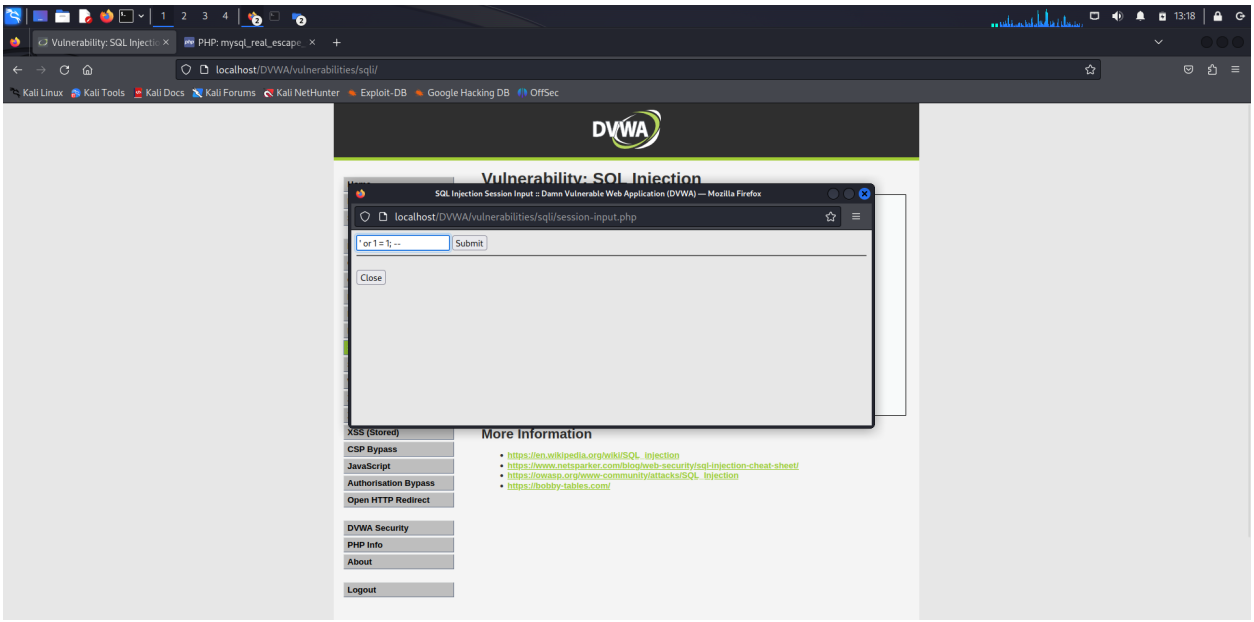


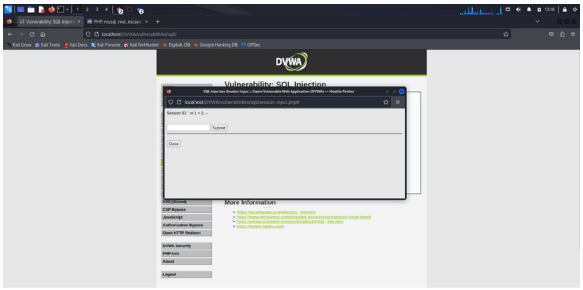


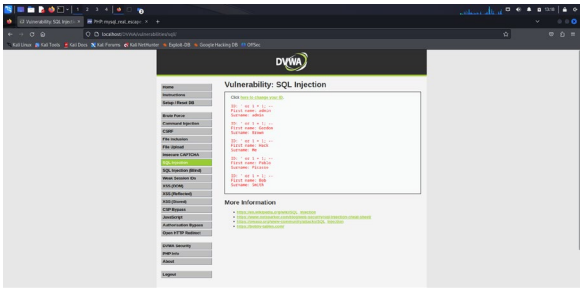
This level's source code is provided by the following:



The code shows that *$id* is, once again, enclosed in quotes but that the query is limiting the results returned to 1 row. We, thus, attempted to append a tautology and comment out the *Limit 1* part of the query to test the vulnerability.



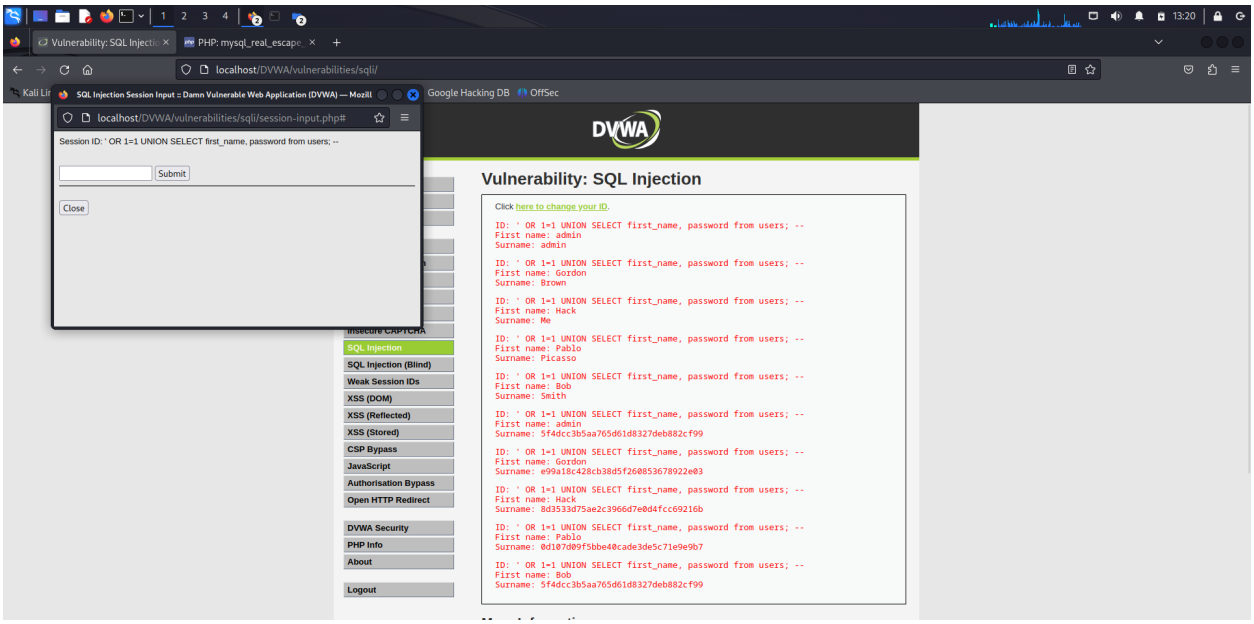




The output reveals the first and surnames of all users in the database, confirming a successful SQLi.

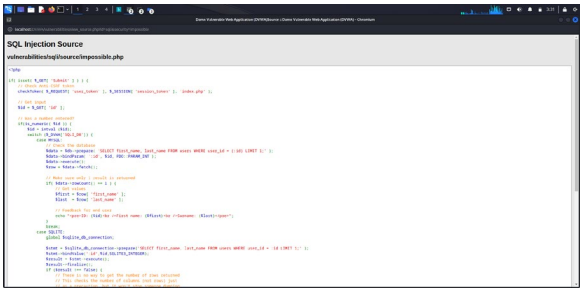
Our next step was then retrieving the passwords of all those users via the following input:

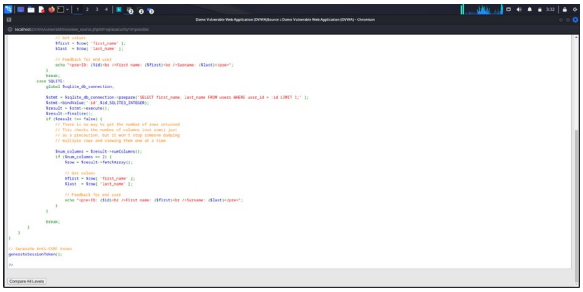
‘ OR 1=1 UNION SELECT first_name, password from users; --



The list of users along with their passwords indicates a successful SQLi attack.

d. Impossible





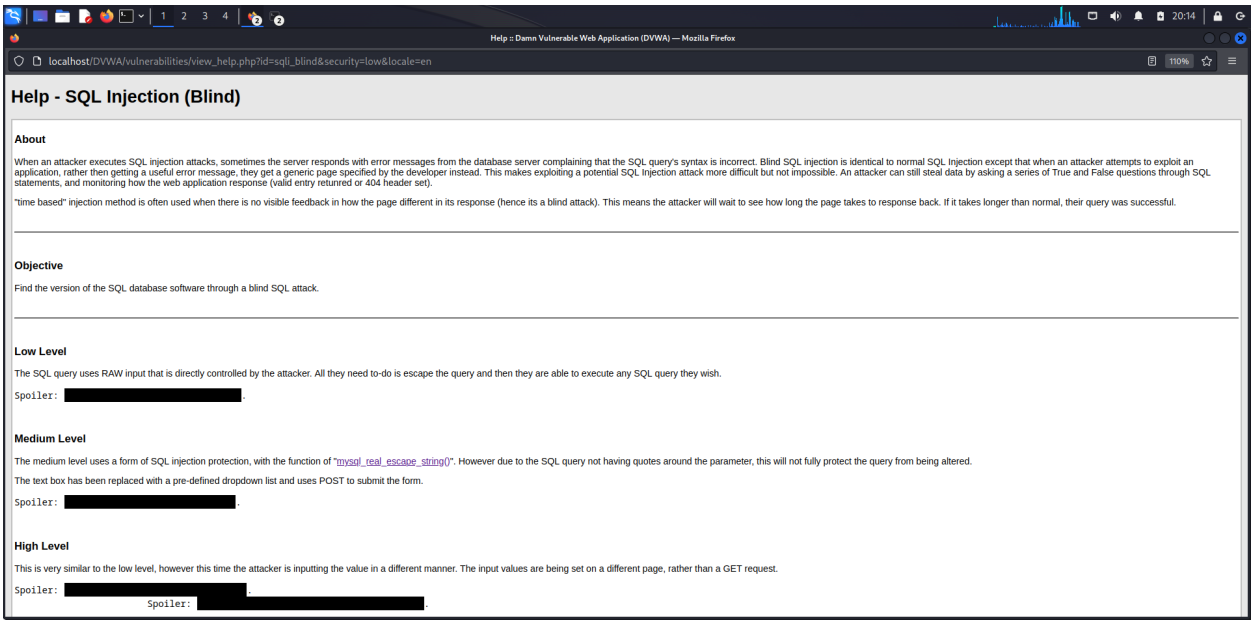
By binding a parametrized *id* variable inside a set without being enclosed in quotations and using PDO to make sure that the input is an integer value (validate the input's data type), the developer has blocked all typical SQL injection attacks against this web service.

3. SQL Injection (Blind)

A blind SQL injection is a type of SQL injection where descriptive errors are suppressed, forcing the attacker to ask a series of boolean-answer questions to reach the desired end goal.

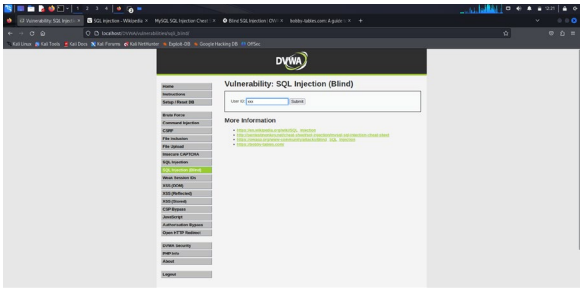
In our case, the objective of our blind SQLi attacks is to determine the version of MariaDB utilized by the web service.

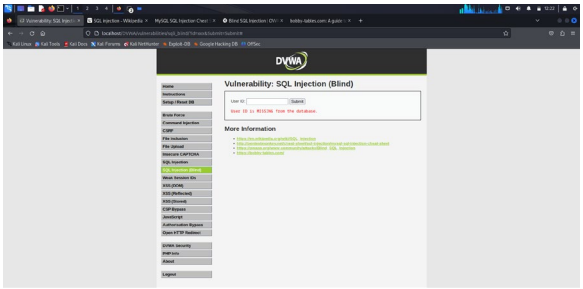
Since the only difference between SQLi and blind SQLi is the output shown to the user, similarly to victims of SQLi, victims of blind SQLi would be susceptible to anything their database is susceptible to. Examples range from data leaks, to data insertions and deletions or manipulation.



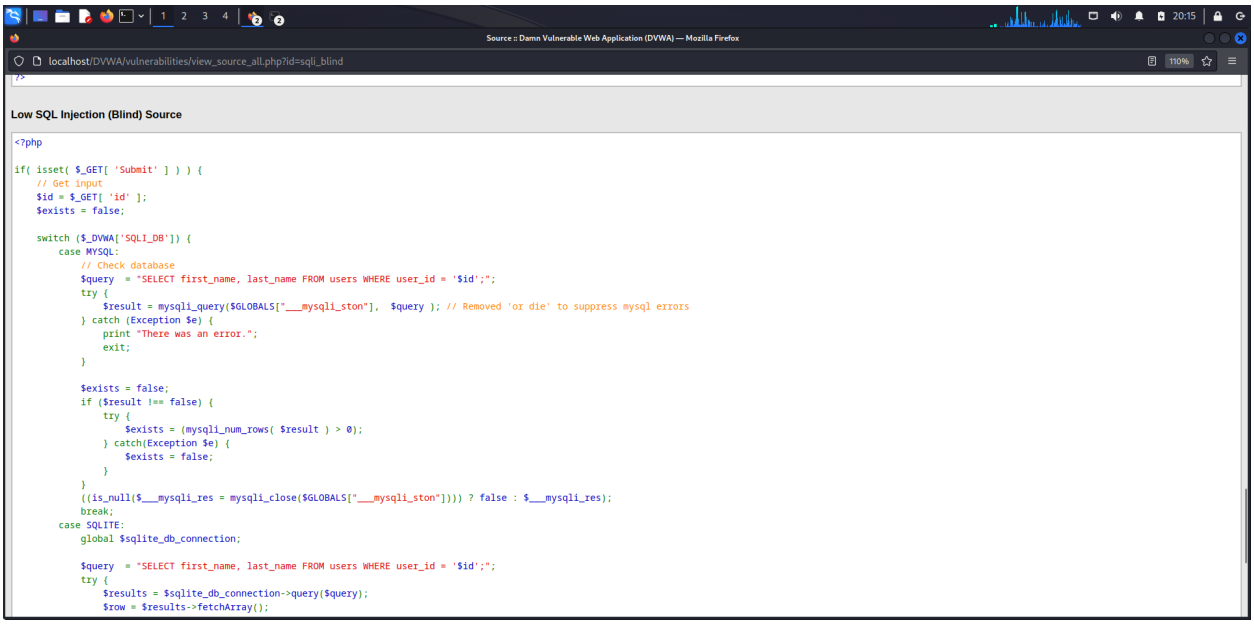
a. Low

Before beginning the attack, we tested the page's behavior against invalid input (a user ID that does not exist in the database: *xxx*).



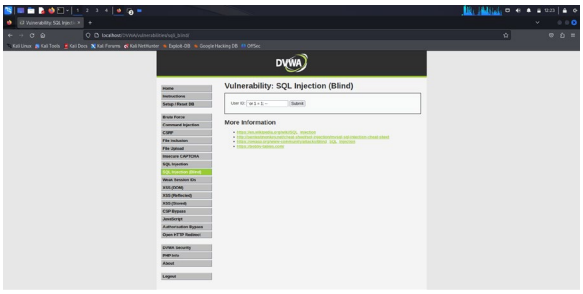


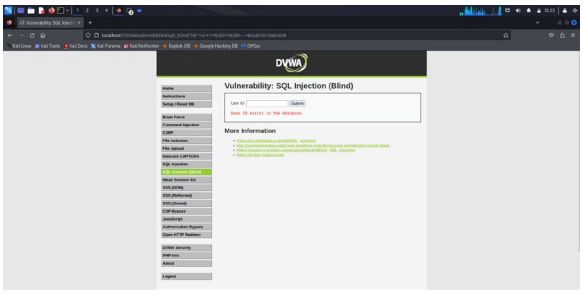
This level's source code is provided by the following:



Upon analyzing the given source code, we noticed that the page returns *User ID exists in the database* whenever the query returns more than 0 rows. Thus, we aimed to apply a UNION to return the row containing MariaDB's version. However, since this is a blind SQLi attack, we needed to test a series of comparisons by checking if the query's return value to the version we're testing against.

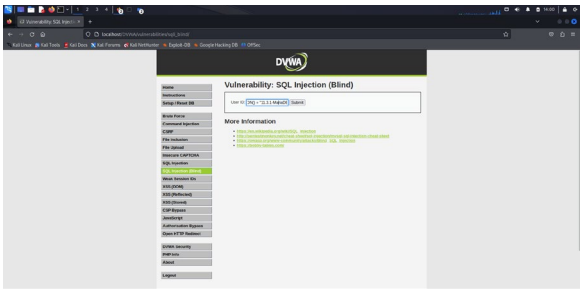
We first confirmed that the site is vulnerable to SQLi by inputting *' or 1=1;* -- into the textbox.

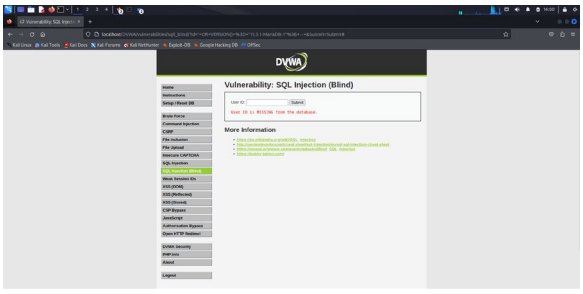




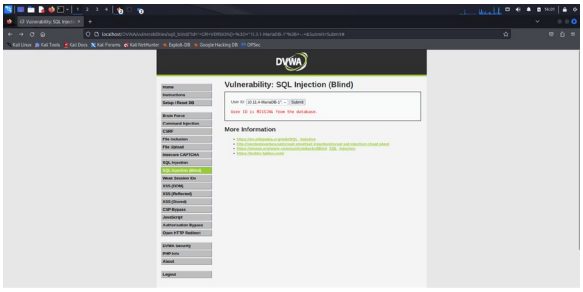
We then execute the query to check for the database version (starting with the most recent one):

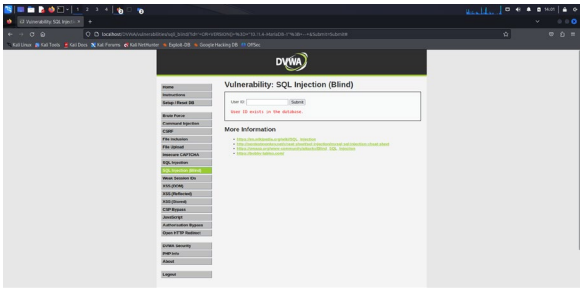
0' OR VERSION()="11.3.1-MariaDB-1" ; --





The output "*User ID is missing from the database*" indicates that the query returned a False value, so we kept on parsing through the different version in decreasing order until the release 10.11.4 displayed "*User ID exists in the database*". The query used is: *0' OR VERSION()="10.11.4-MariaDB-1" ; -- .*

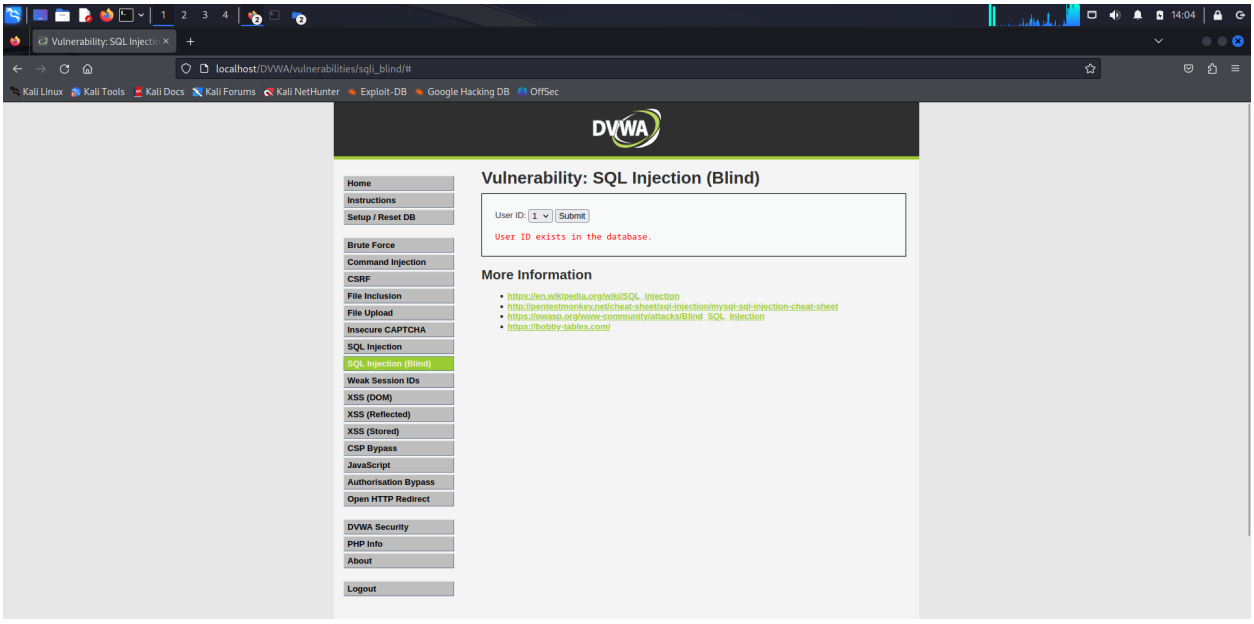


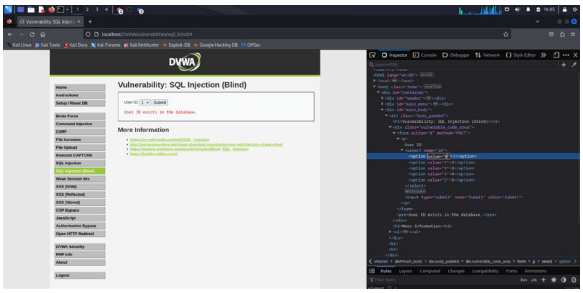


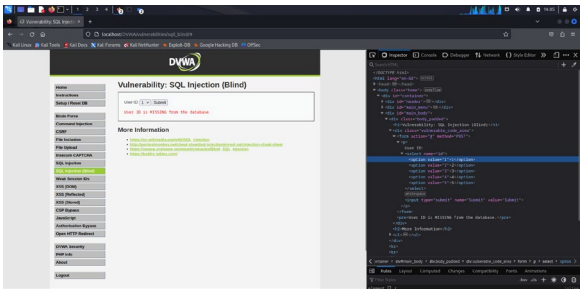
The confirmation message verfies our successful blind SQLi attack as well as the database's version.

b. Medium

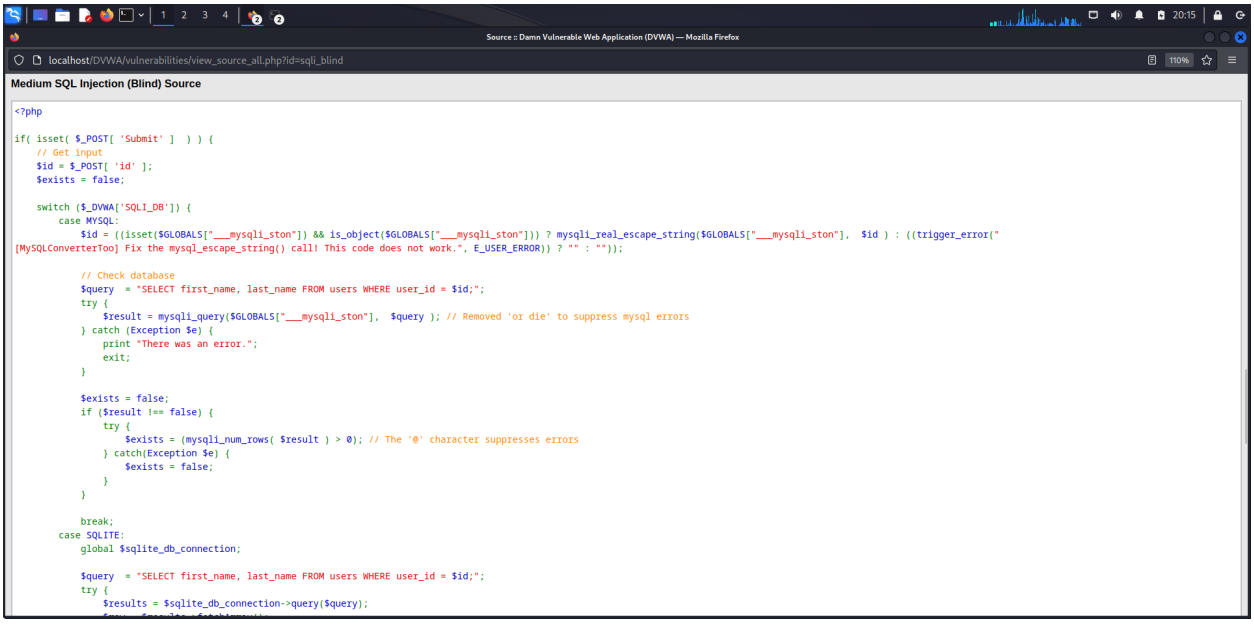
We tested the page's behavior against valid input (a user ID of 1 exists in the database) and invalid input (a user ID of 0 does not exist in the database) by changing the *value* of the *option* in the HTML *form* using the browser's *inspect tool* since the medium-level blind SQLi challenge attempts to restrict user input by replacing the text box with a dropdown, similarly to that of SQLi (in the previous section).





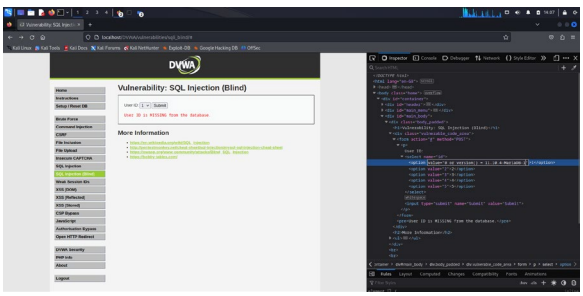


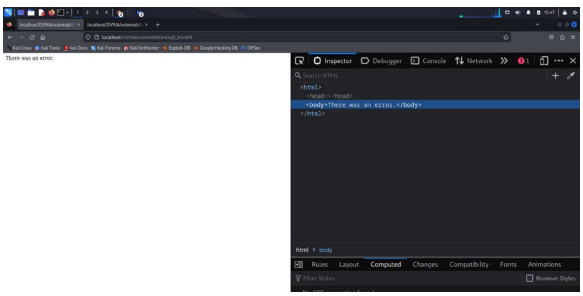
This level's source code is provided by the following:



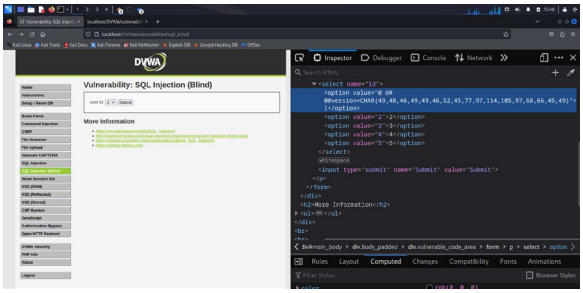
Upon analysis, we realized that the query was being filtered out of escape characters using the *mysqli_real_escape_string()* function. This made things harder particularly because we couldn't use quotations anymore. As we already knew the database's version from the low-level blind SQLi attack, we used that as a reference point.

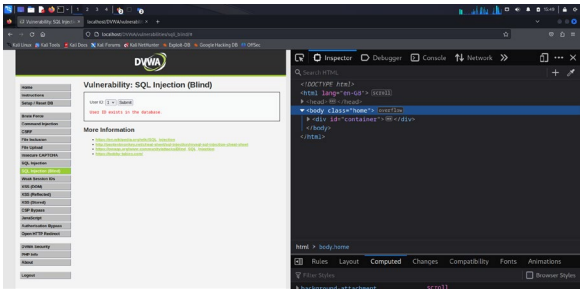
Due to the restriction of not using quotes, trying to compare the version using *0 or version()='10.11.4-MariaDB-1'* was not possible and instead lead to the same error page no matter what variation we tried.





We were able to maneuver around this restriction by using the built-in CHAR() function that converts ascii numbers to string characters. We transformed *10.11.4-MariaDB-1* to ASCII using an online tool and inserted the converted text into the CHAR() function. We were then able to compare the return value of *@@version()* to the return value of CHAR().



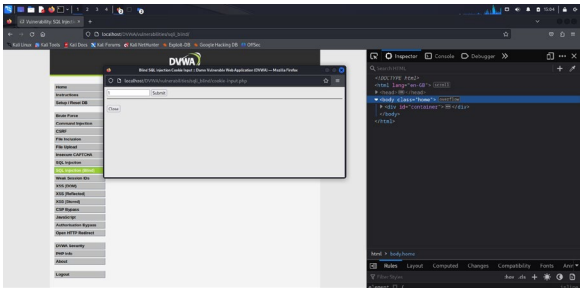


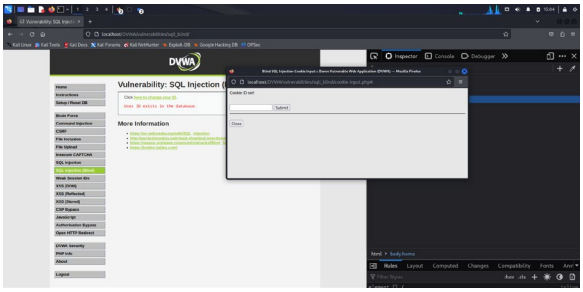
The "*User ID exists in the database*" confirmation message indicates our blind SQLi attack was successful and that the database's version is indeed *10.11.4*.

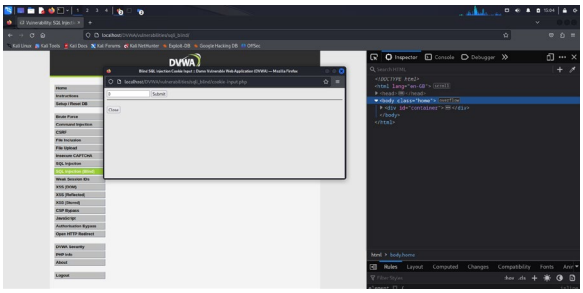
c. High

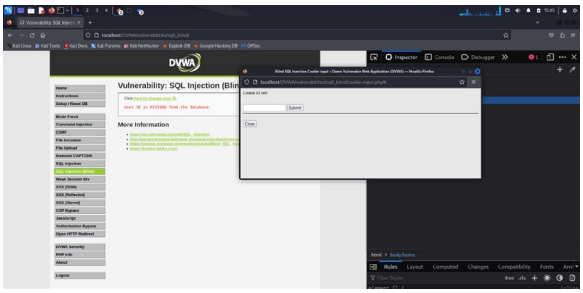
Similarly to the high-level of SQLi, blind SQLi incorporate a separate page that reads user input similarly to the low-level blind SQLi challenge.

We tested the page's behavior against an existing ID (1) and a missing ID (0).

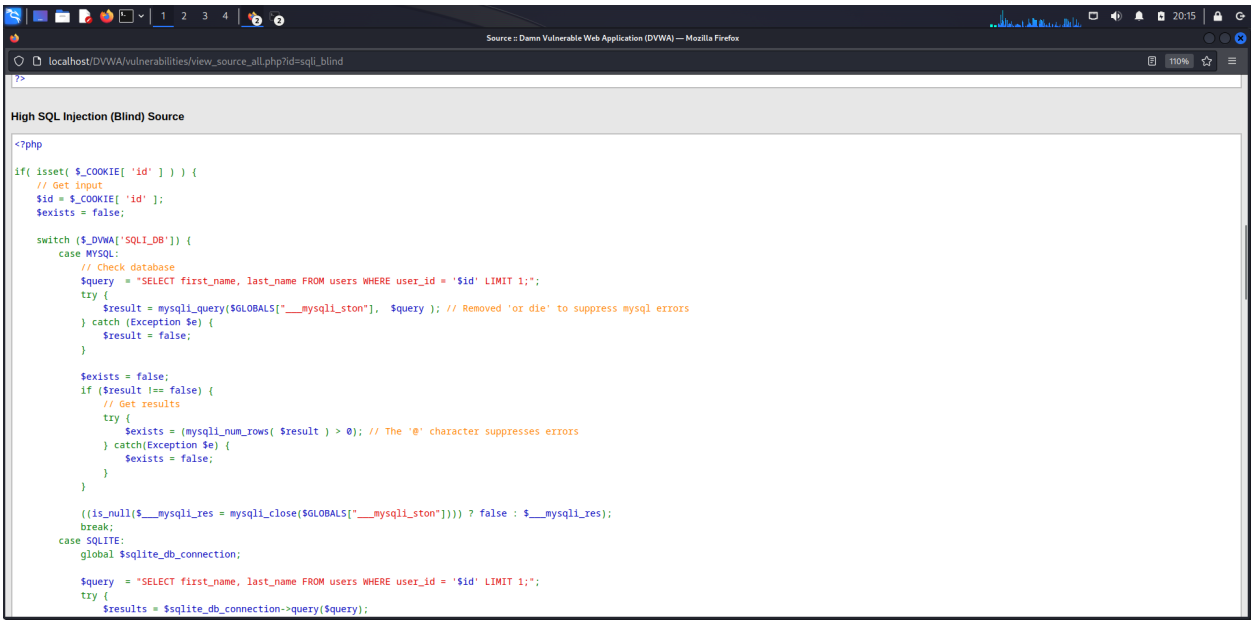






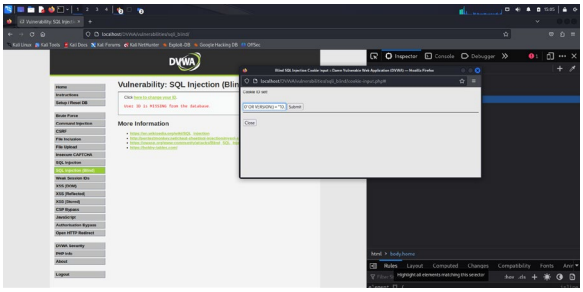


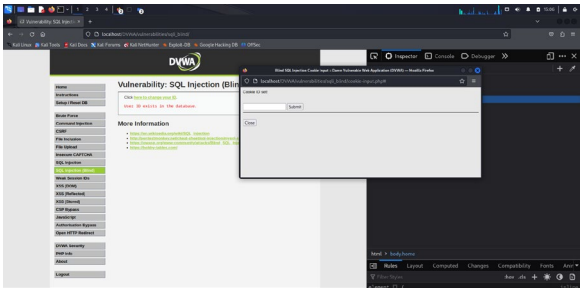
This level's source code is provided by the following:



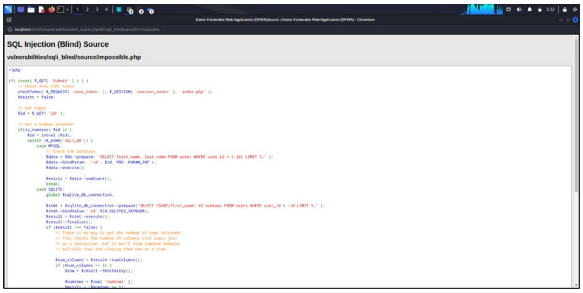
Our methodology follows the same logic as the high-level SQLi attack. Terminate the given query early with a single quote (‘) and chain an *OR* to check for the database’s version before commenting out the rest of the query to dismiss the *LIMIT 1* constraint of the query. The input would, thus, be:

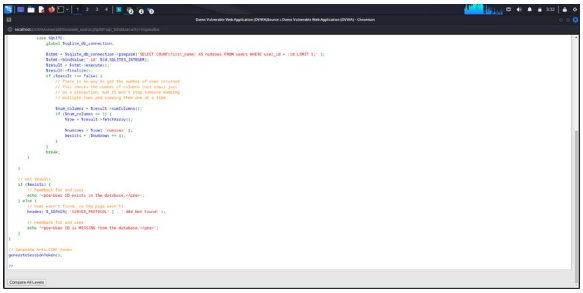
0’ OR VERSION()=”10.11.4-MariaDB-1”; --





d. Impossible





Since a blind SQLi is practically the same as a SQLi, with the exception of the output shown to the user, the same methods used to protect against SQLi attacks are used to protect against blind SQLi attacks.

Mainly, the data type of the input is being verified as an integer and instead of enclosing the *id* in quotes or have it concatenated within the string, it is parametrized in a bind.

III. Cross-Site Scripting (XSS) Challenges

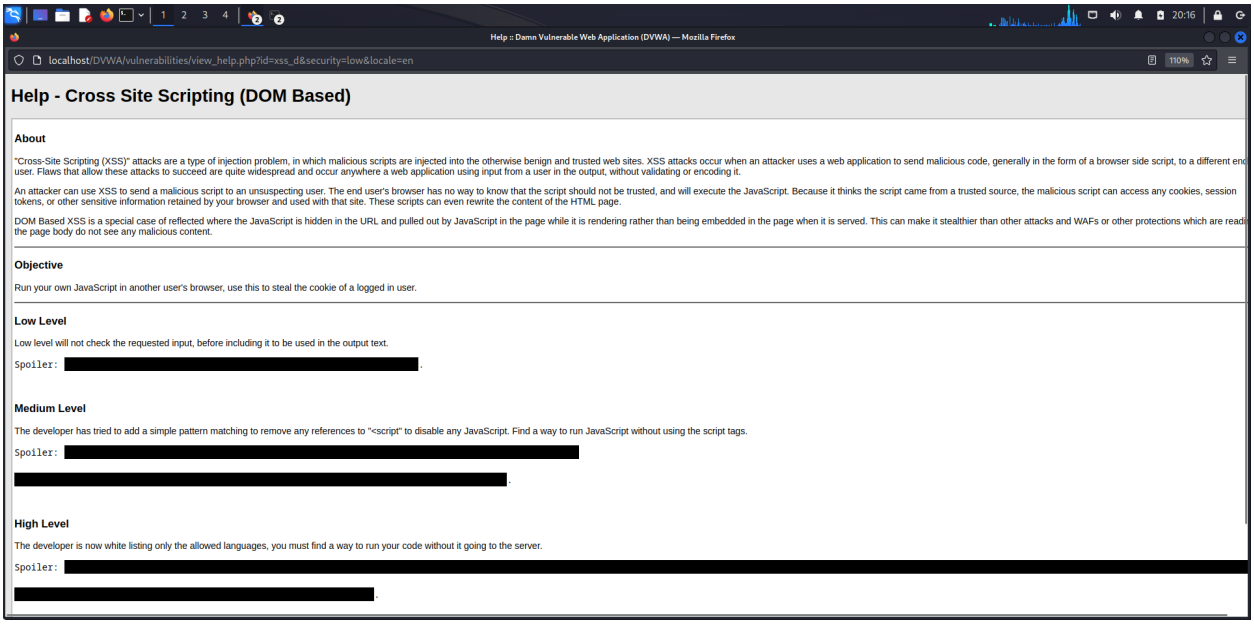
Cross-site scripting attacks are a type of injection attack where the input of one user is subsequently output to another user. The vulnerability arises from the fact that most websites or services typically trust the input or output enough not to validate.

Victims of such attacks would typically be vulnerable to whatever commands the attacker can sneak through the site's validation. As these attacks are based on websites not validating input effectively, visitors of these websites could be prone to information leakage, adware, clickjacking, or whatever the malicious script's payload is.

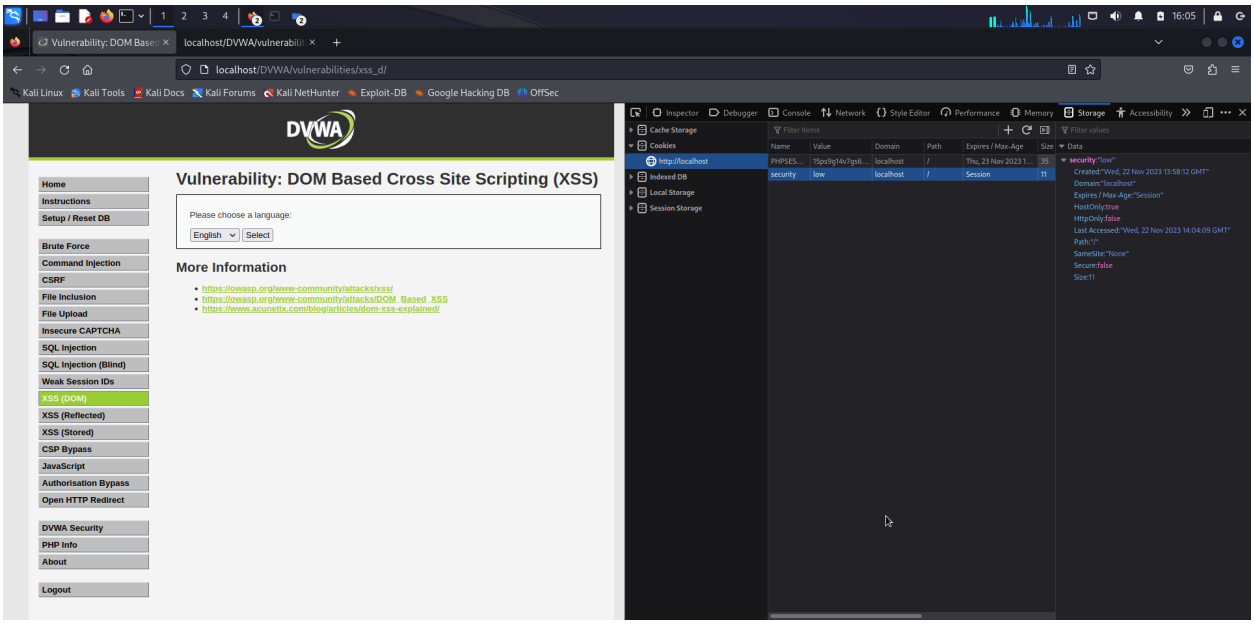
1. XSS (DOM)

DOM XSS is a special instance of Reflected XSS where the code is hidden somewhere in the URL.

Our objective is to the steal the cookie of a logged in user by running our own malicious JavaScript code in the victim's own browser.

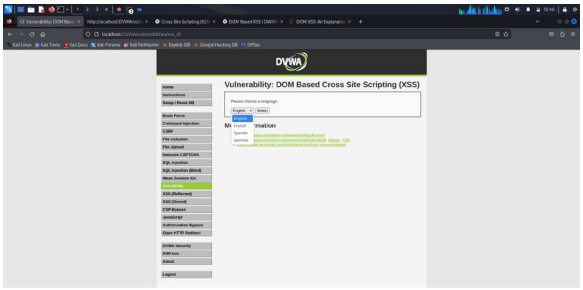


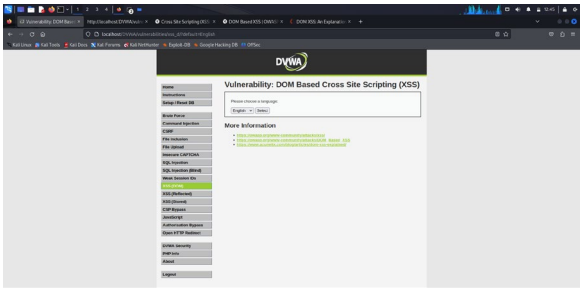
Before starting, we used the browser's storage tab in the inspection menu to check what cookies are available. We noticed that the security level of the challenge we're currently attempting to solve is set as a cookie, which we will thus be trying to retrieve.



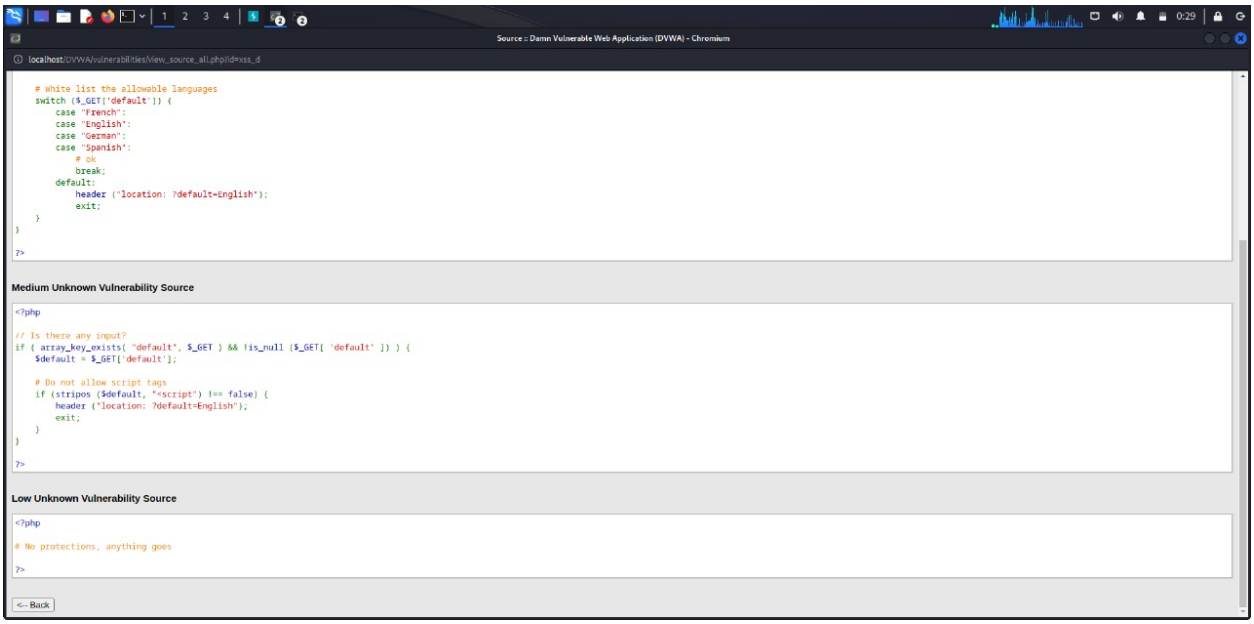
a. Low

Before executing the attack, we tested the page to try to analyze its behavior. Upon submission the URL changes as to add an extra *?default=English* section which indicates the default option to be highlighted in the HTML select menu.



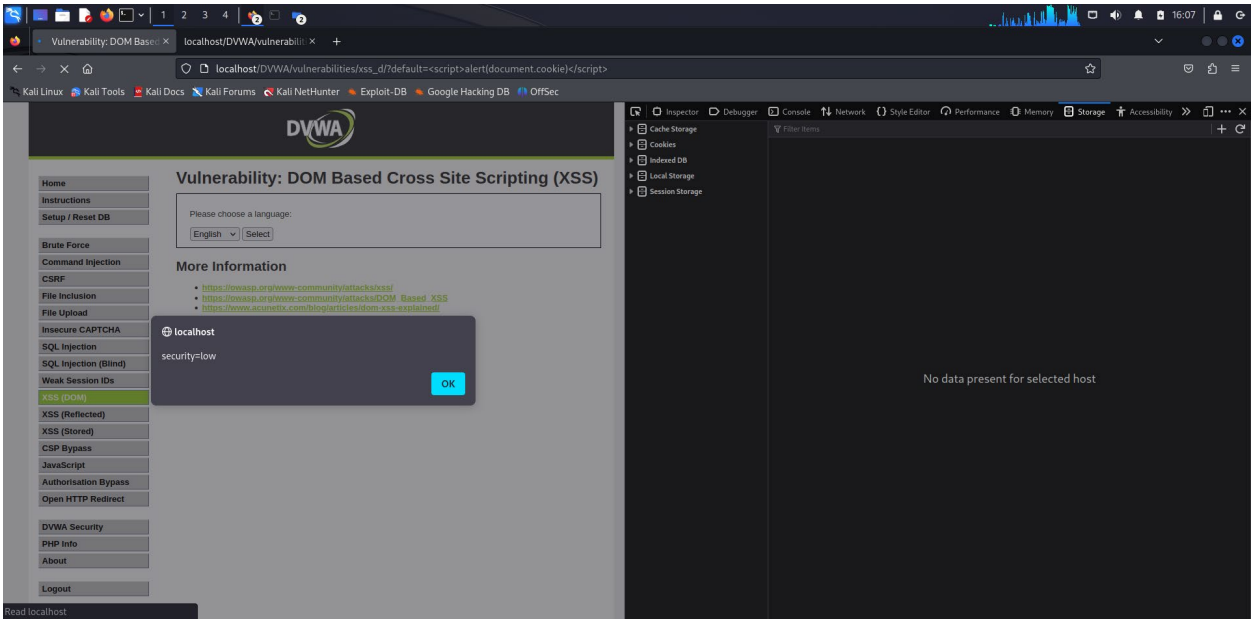


This level's source code is provided by the following:



Upon inspection, we notice that input validation does not execute for this security level.

The URL containing the malicious script appends *?default=<script>alert(document.cookie)</script>* to the base URL of the page.

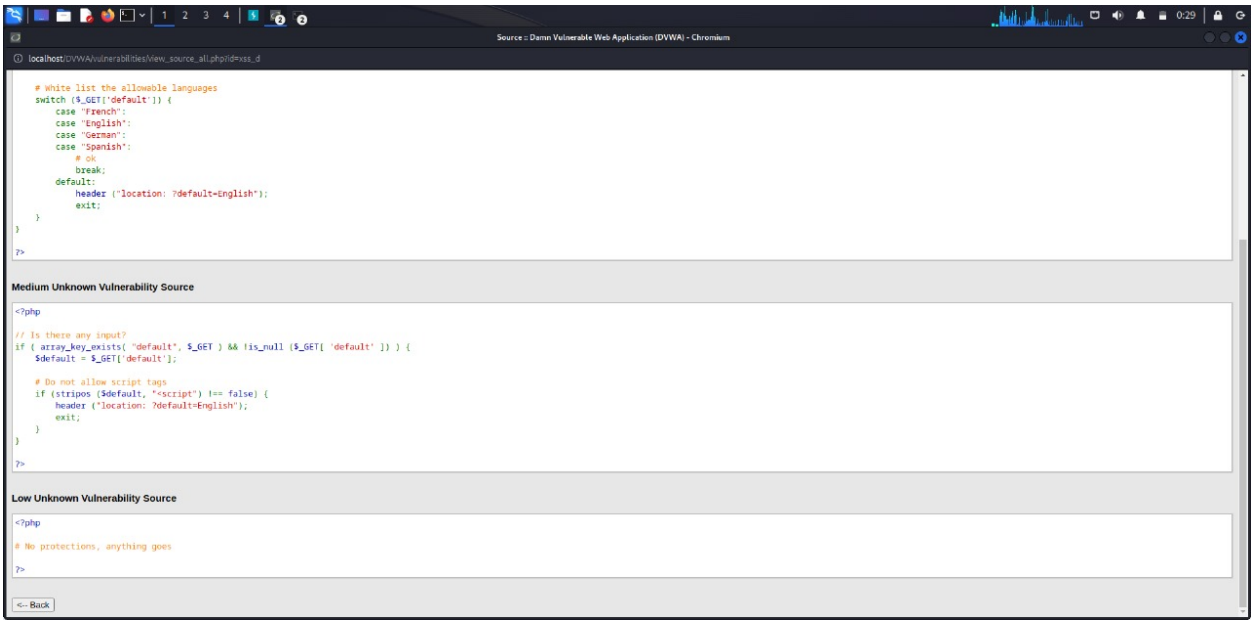


The alert containing the cookie of low security confirms our successful DOM XSS attack.

b. Medium

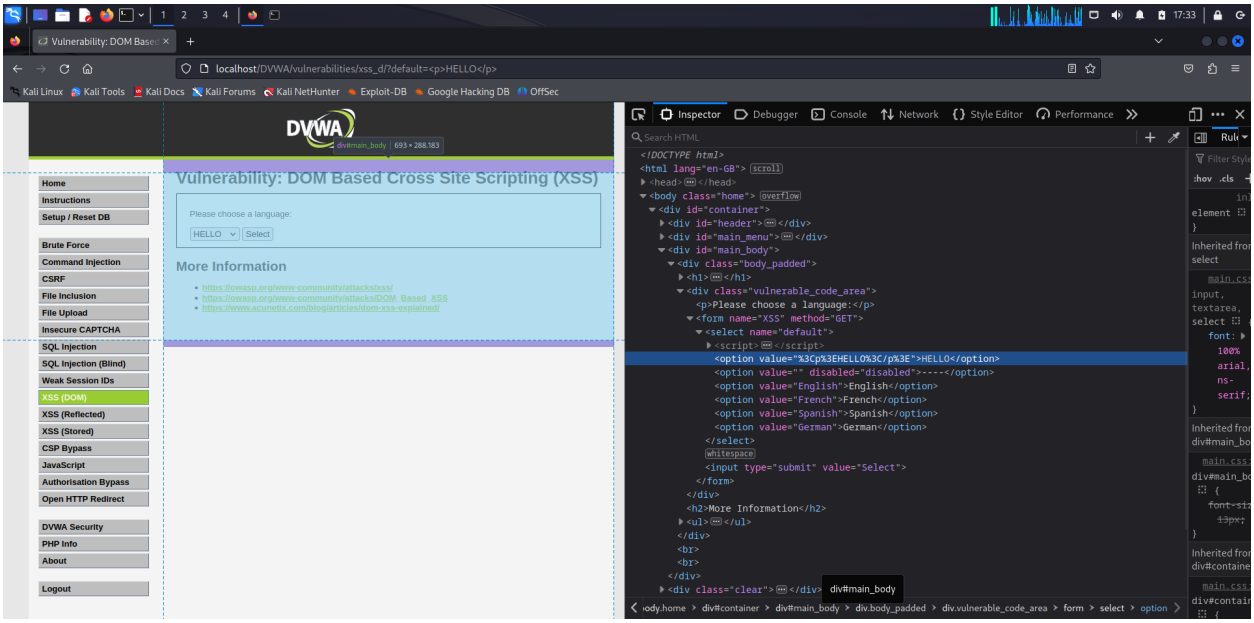
The medium-security challenge attempts to filter out any "*<script*"s found in the URL to disable any JavaScript.

This level's source code is provided by the following:

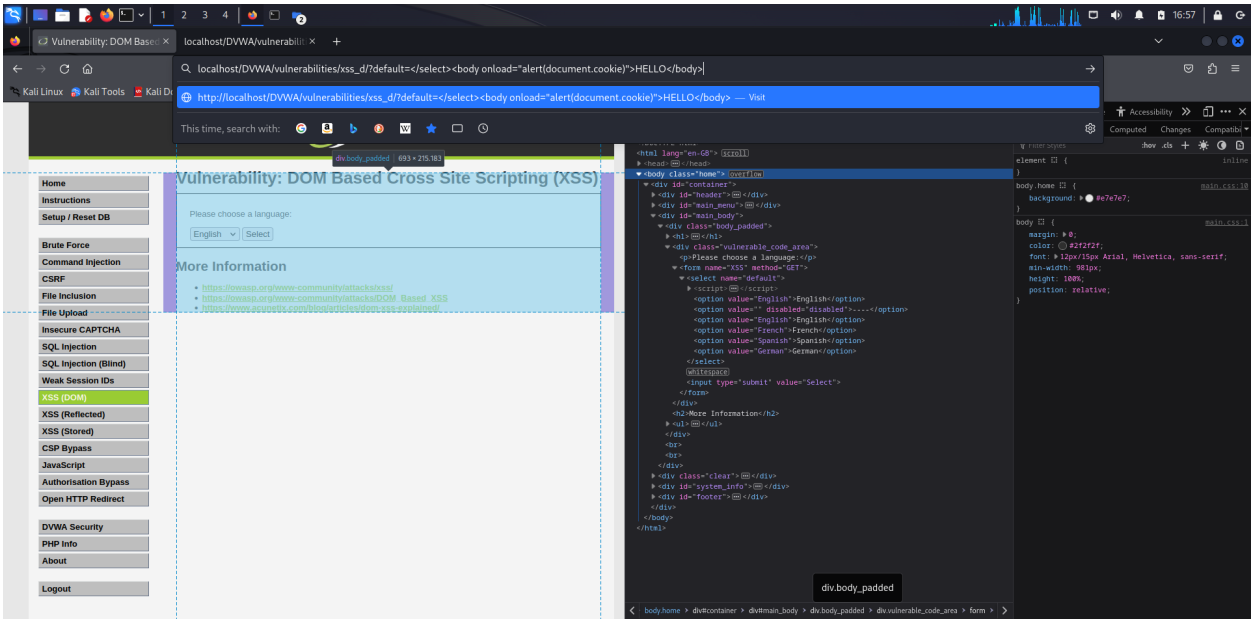


We are, however, able to execute JavaScript on page load by setting the *onload eventListener* on the *body* HTML element. This executes whatever code is provided as soon as the *body* is loaded.

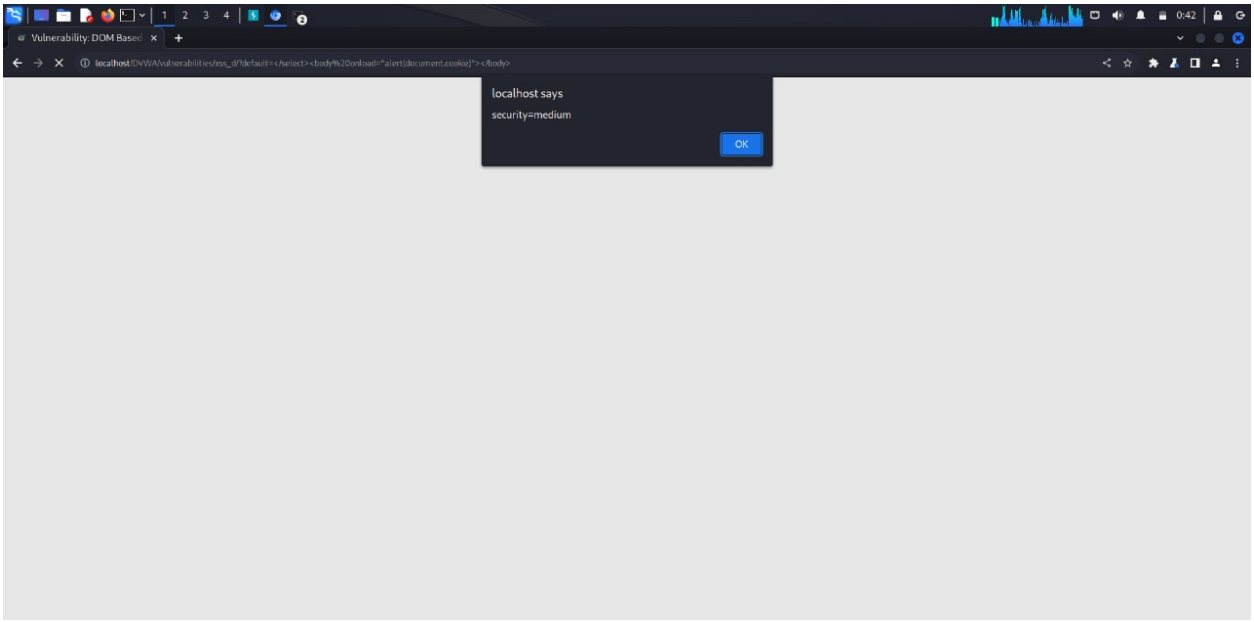
We tried inserting different HTML elements by replacing the <*script*> tags of the low-security level by the tags of elements we're trying to insert. However, upon using the inspect tool, we noticed that the code was not changing as we expect. The text within the tags was inserted into the DOM but the tags themselves.



The malicious section added to the base URL is as follows: "*?default=</select><body onload="alert(document.cookie)">HELLO</body>*" since the only tag that would alter the HTML page was closing *<select>*.



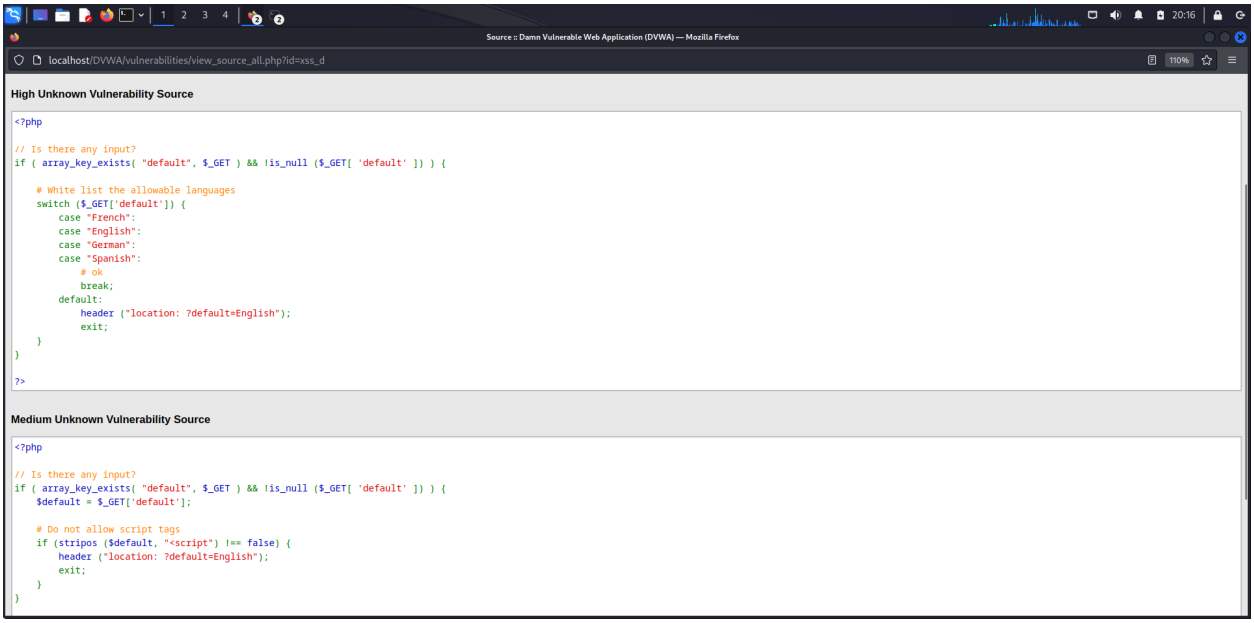
This overrides the attributes of the *body* element since only one body can exist per page. The resulting alert is shown in the next screenshot.



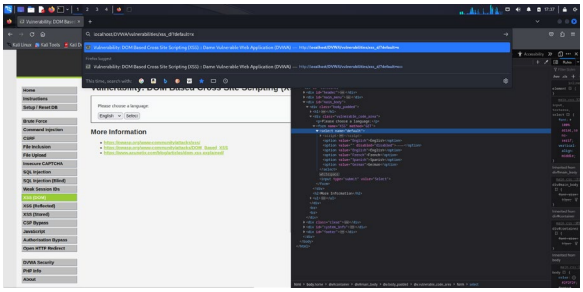
The *security=medium* alert message confirms our successful DOM XSS attack.

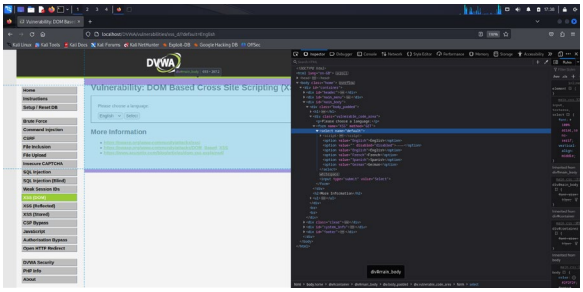
c. High

The high-security challenge whitelists only allowed languages to be submitted, so we need to run our script without it going to the server. The list of whitelisted languages is viewable in this level's source code provided by the following:

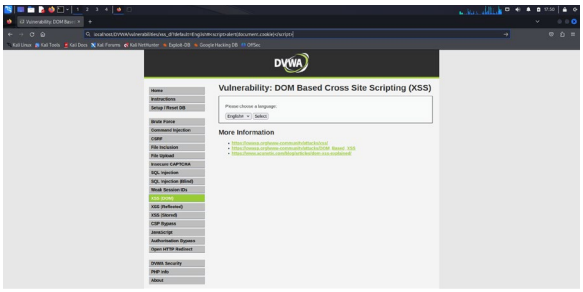


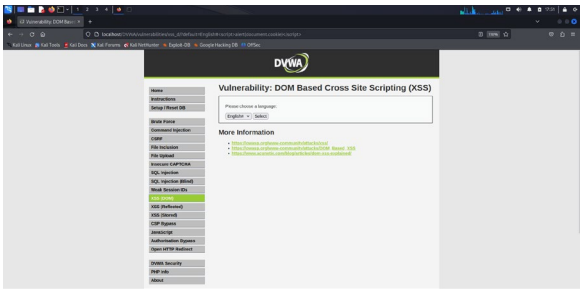
To test the page's behavior, we attempted to add *?default=x* to the base URL. The URL resets to *?default=English.*



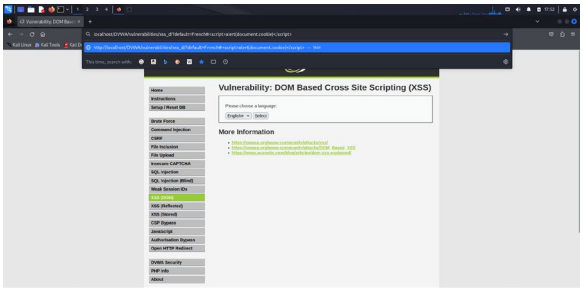


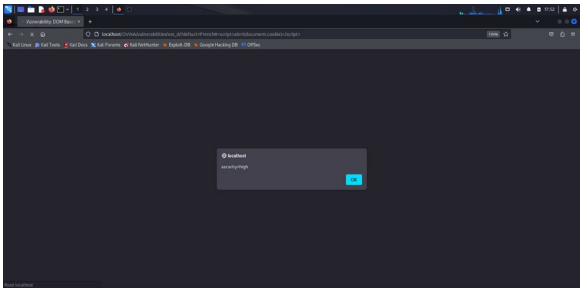
Referencing this challenge's spoiler, anything that follows the # symbol in the URL is not run by the server, so the malicious URL is now: *?default=English#<script>alert(document.cookie)</script>*





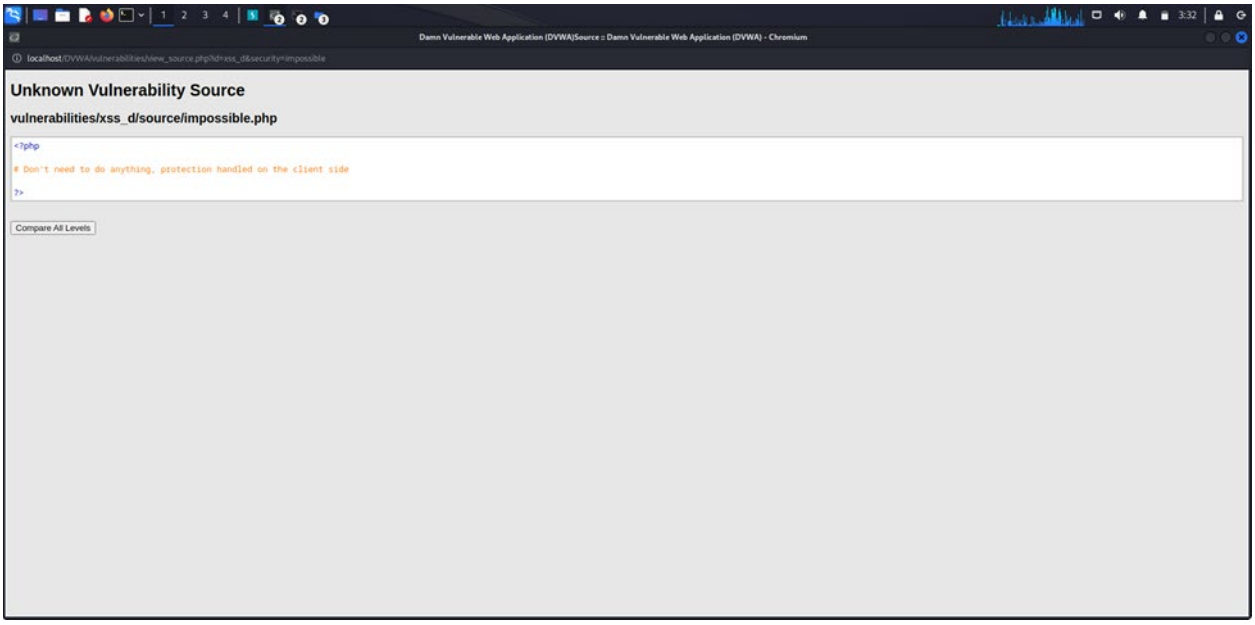
However, since the page was already cached in memory, the malicious script did not execute. A different result appeared when we changed the language from English to French, hence resulting in the following part of the URL: *?default=French#<script>alert(document.cookie)</script>*





The *security=high* alert message confirms our successful DOM XSS attack.

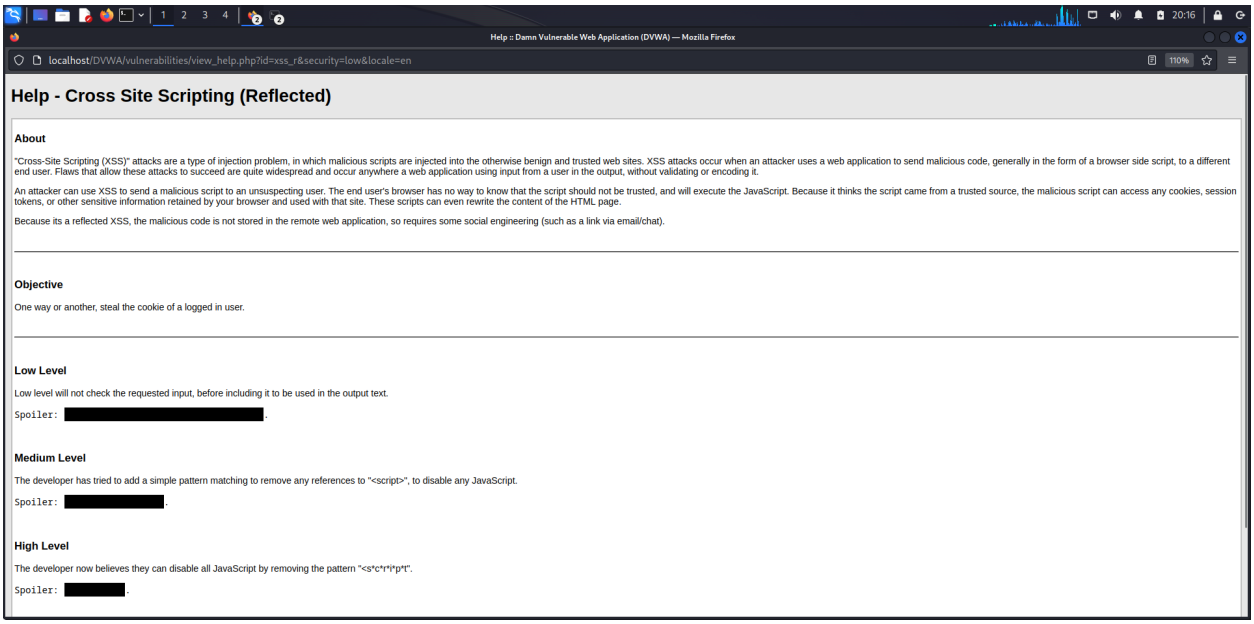
d. Impossible



2. XSS (Reflected)

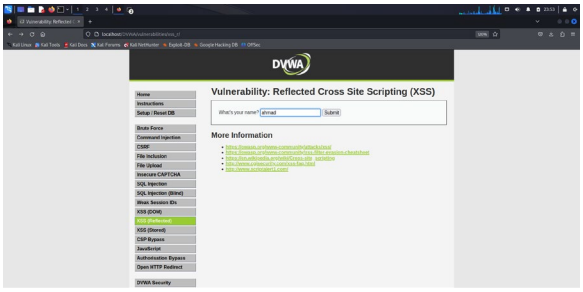
Reflected XSS includes malicious code that is not stored in the remote web application.

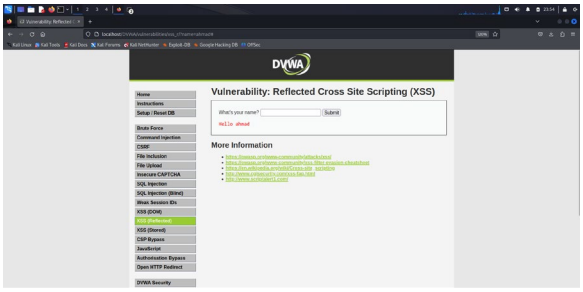
Our objective is to, once again, steal the cookie of a logged in user.



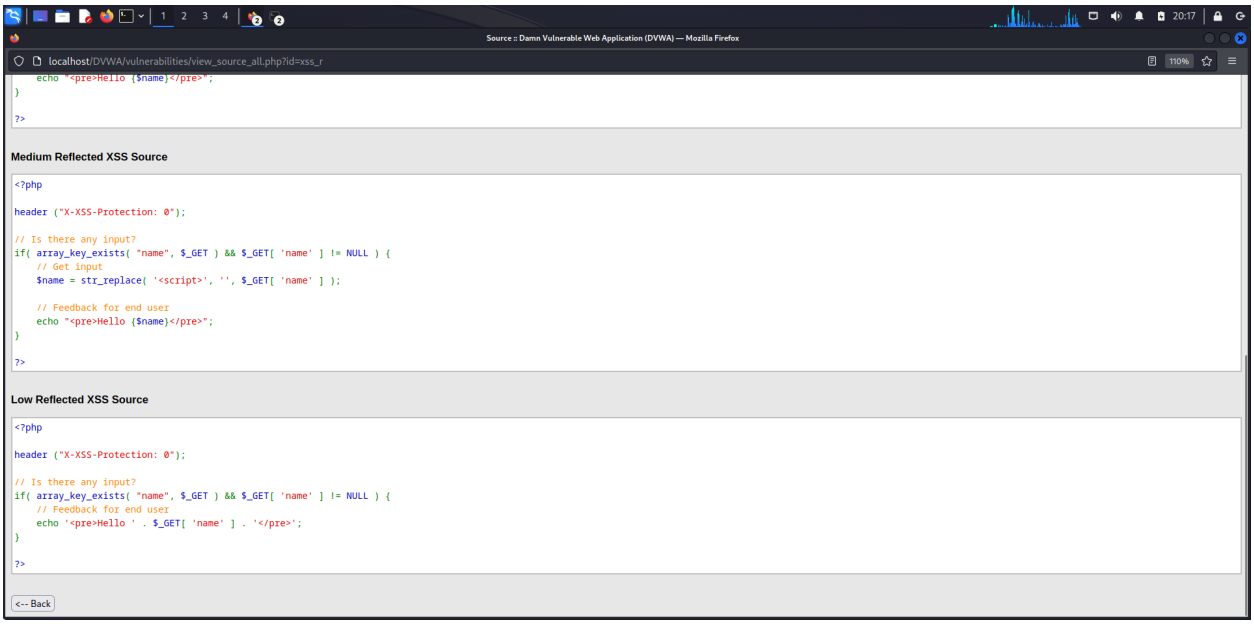
a. Low

Before implementing the attack, we tested to check the page's behavior against typical input.

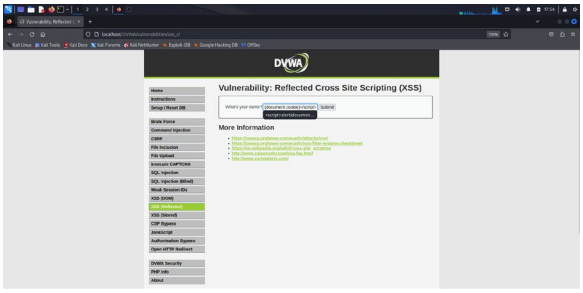


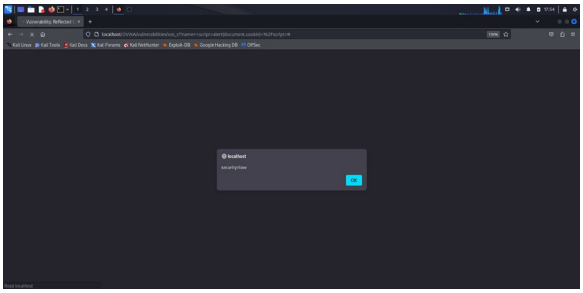


The source code for this level is provided by the following:



Since no input validation is performed, we were able to extract the cookie by inserting *<script>alert(document.cookie)</script>* into the textbox.

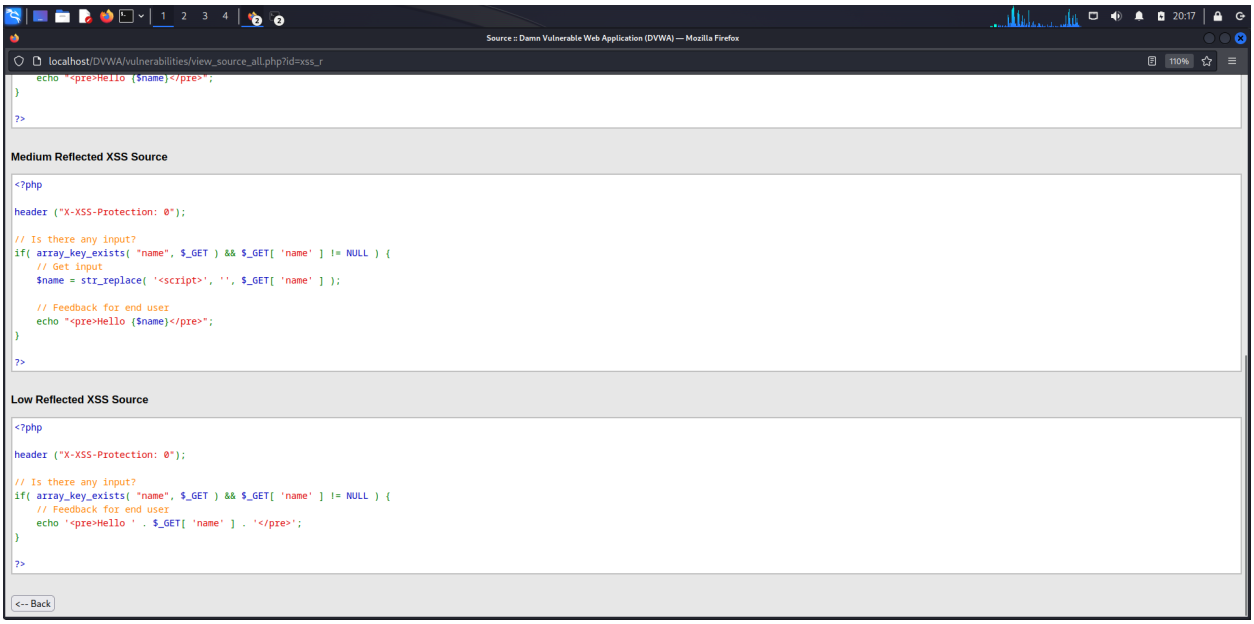




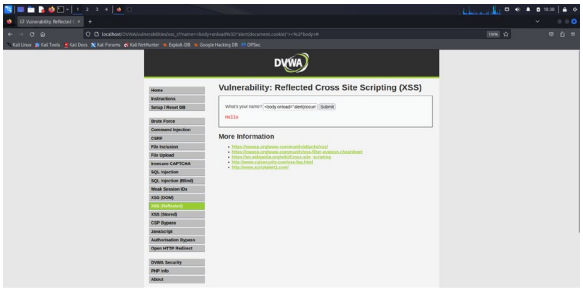
The *security=low* alert message indicates a successful Reflected XSS attack.

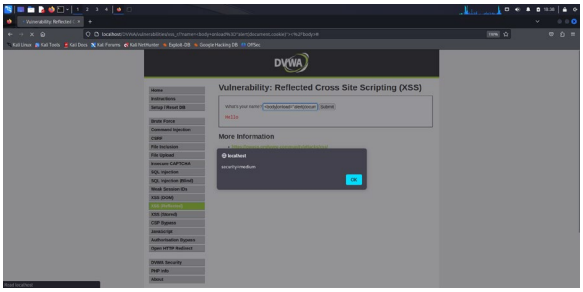
b. Medium

The level of the challenge attempts to restrict user input by filtering out any *<scripts>* recorded as input as provided by the following source code:



Similarly to the DOM XSS, however, we were able to maneuver around this restriction by overriding the *body*'s onload attribute to *onload=”alert(document.cookie)”* via the input *<body onload=”alert(document.cookie)”></body>*

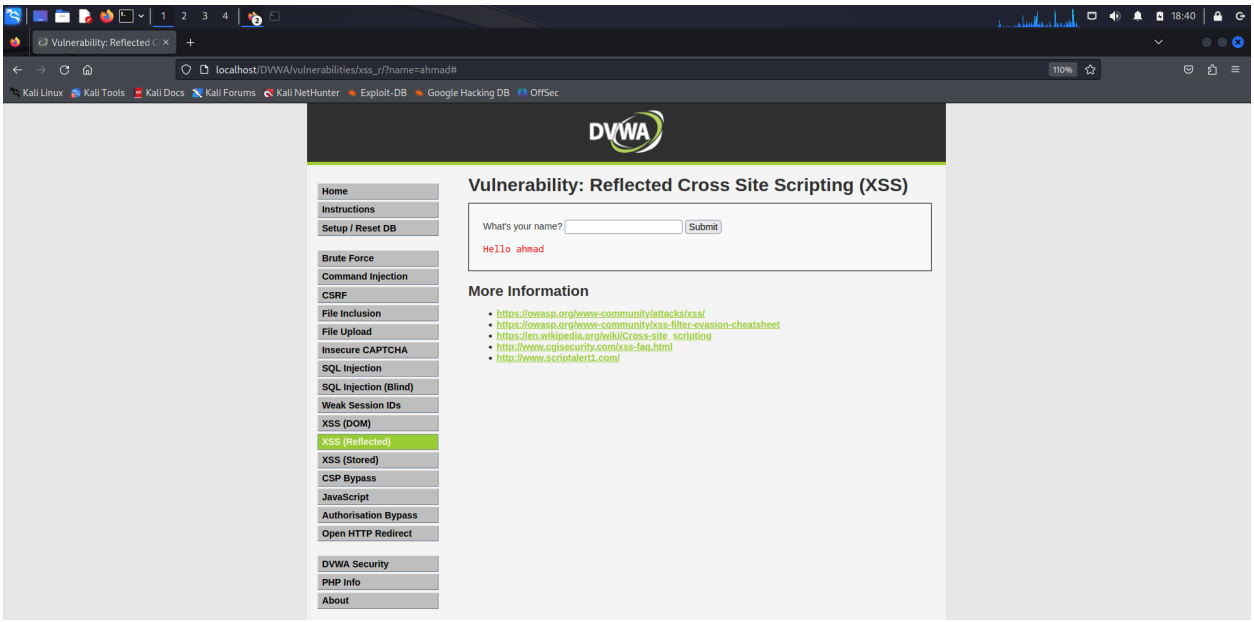




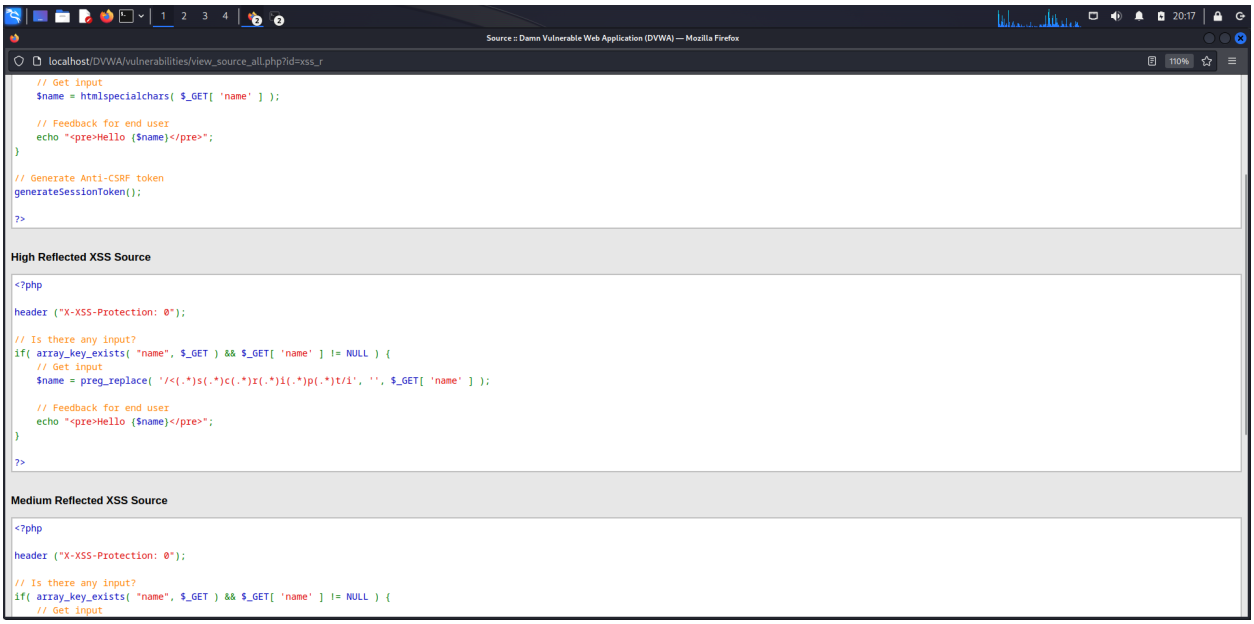
The *security=medium* alert message confirms our successful Reflected XSS attack.

c. High

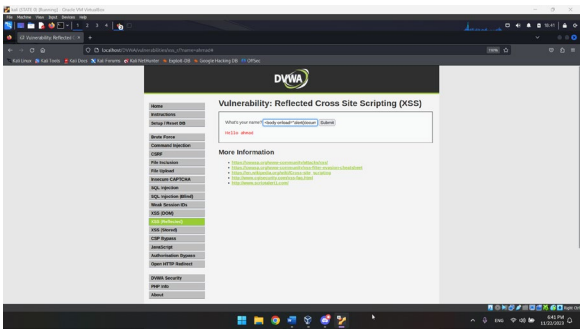
We, again, tested the behavior of the page against typical input from the user.

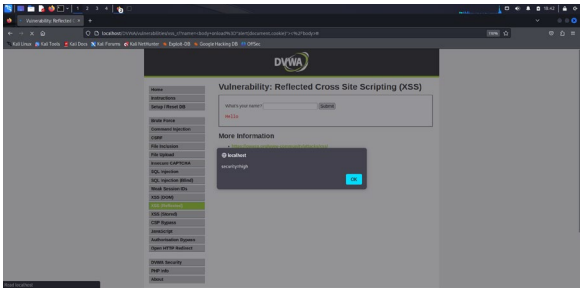


This level of the challenge uses a more advanced pattern to filter out any combination of <*script*>. This level's source code is provided by the following:



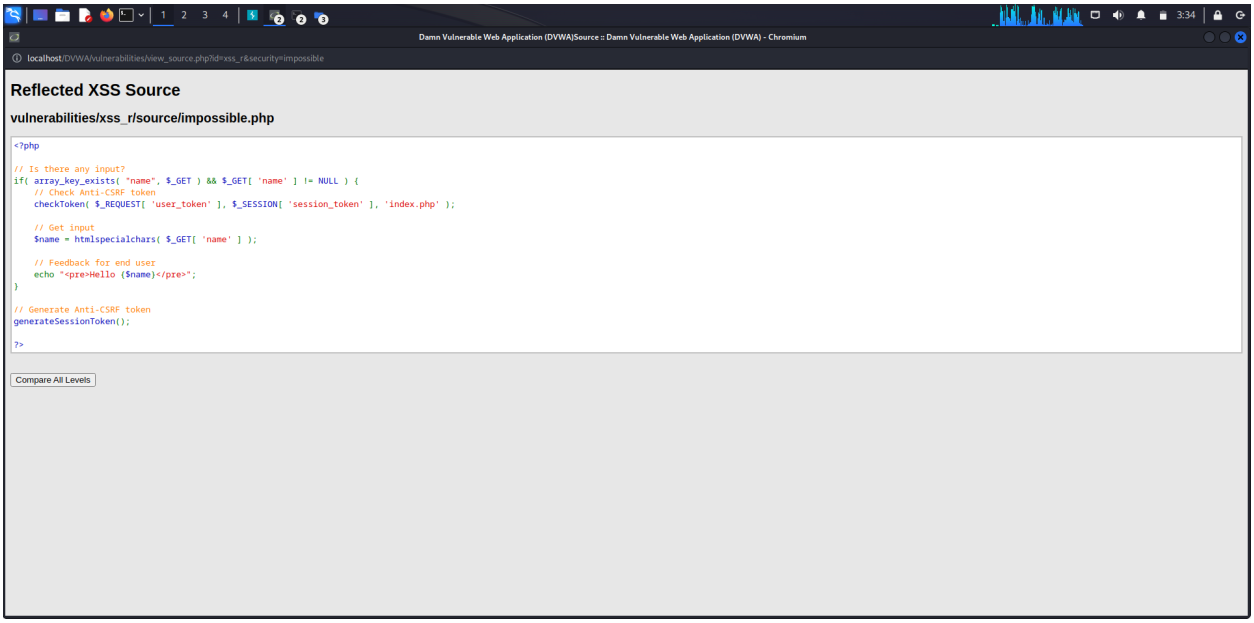
This restriction does not affect us since we've already completely disregarded the <*script*> tags when solving the medium-level challenge. Therefore, we simply reiterate over our method for solving the medium-level challenge: override the *onload* attribute of the *body* element to *alert(document.cookie)* by inserting into the textbox: *<body onload="alert(document.cookie)"></body>*





The *security=high* alert message confirms our successful Reflected XSS attack.

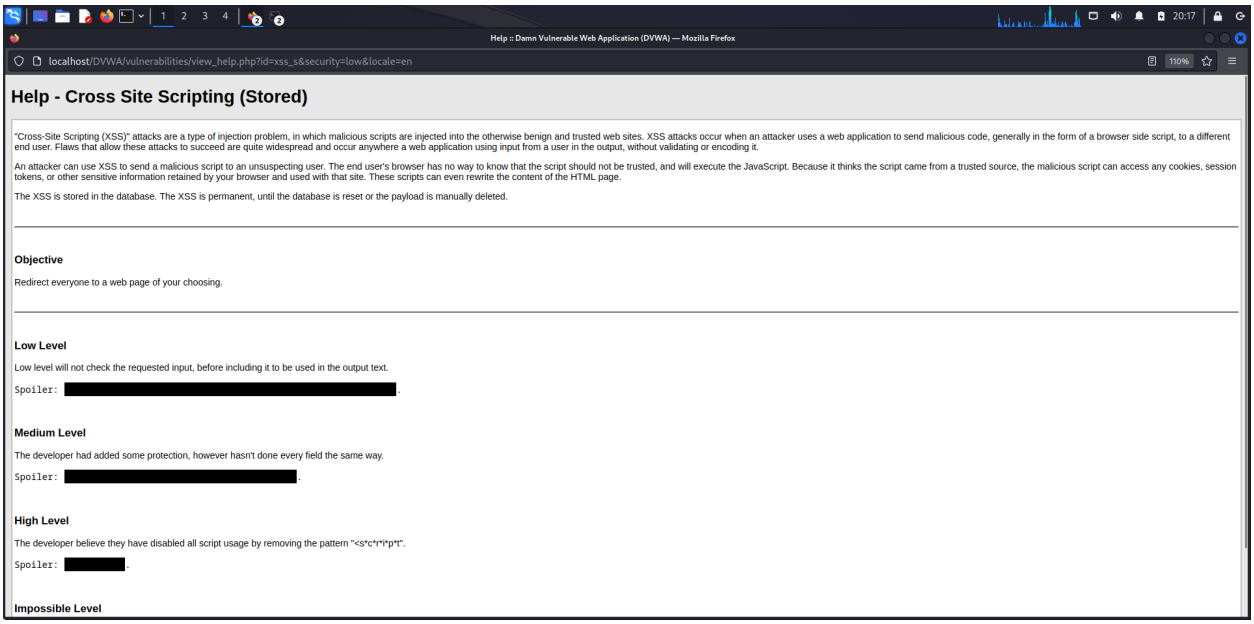
d. Impossible



This level's provided source code shows that the user input is handled by the *htmlspecialchars* function to transform special characters into their HTML string representation. This hinders any possible attempts to run scripts (the less than '<' and greater than '>' symbols can no longer be used in the input as well as both single and double quotation marks) and protects the service against Reflected XSS attacks.

3. XSS (Stored)

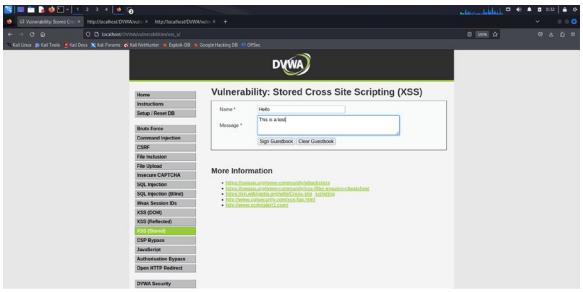
Stored XSS attacks are permanent since they are stored in the database of the web service. Consequently, these attacks are executed every time a user visits the page (regardless of whether malicious content was added to the URL).

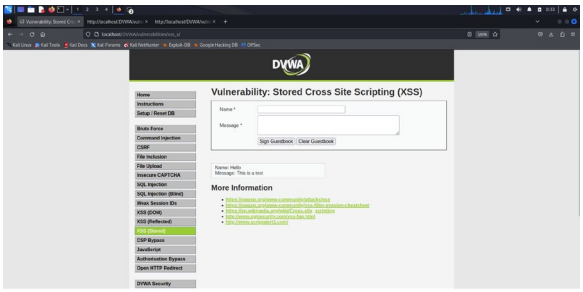


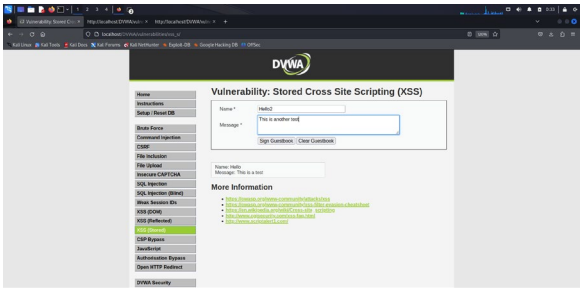
Our objective for this XSS attack is to redirect users to the LAU website.

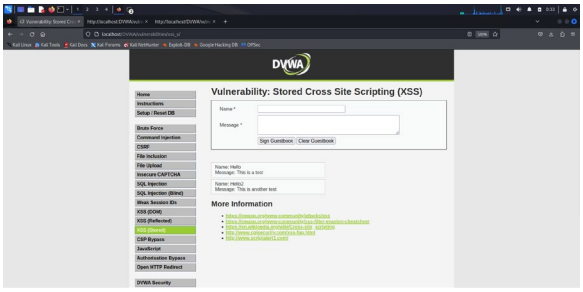
a. Low

We first analyzed the behavior of the page against typical input. The combinations of name and message are saved (permanently) in the page, and more accurately, in the database.

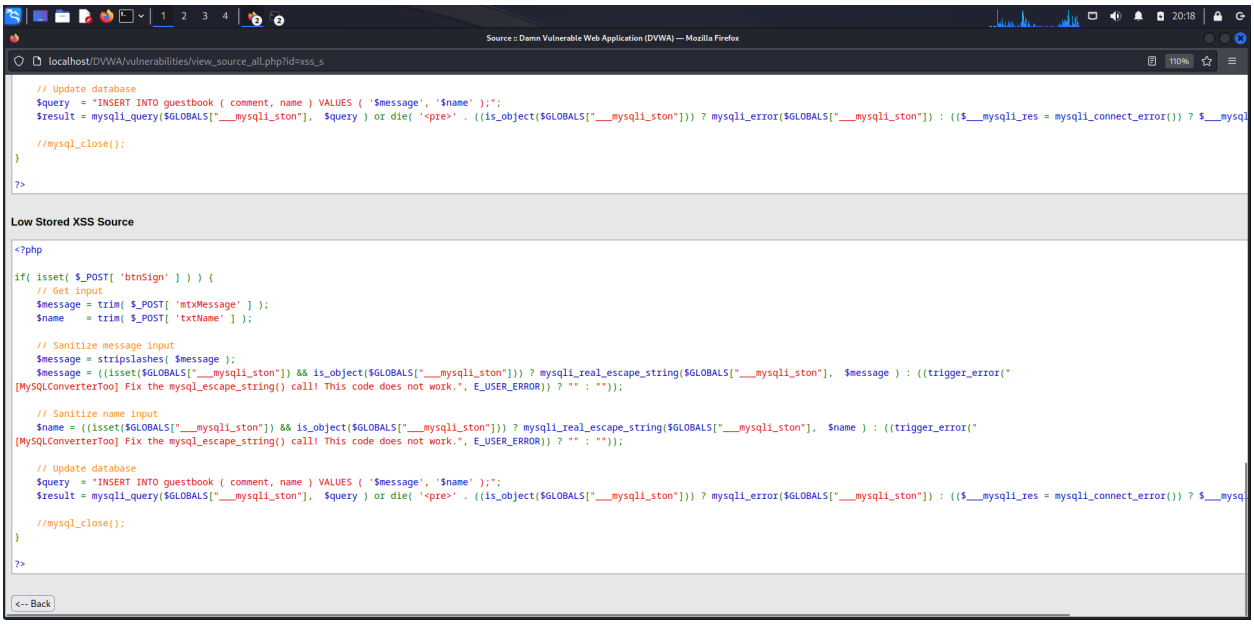






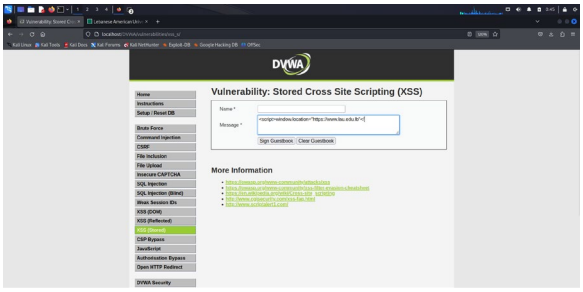


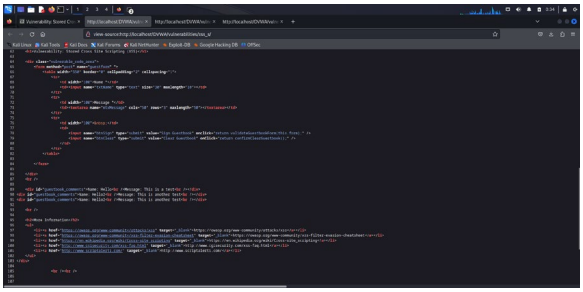
The low-level challenge, as usual, does not perform any form of input validation of input checks. This level's source code is provided by the following:



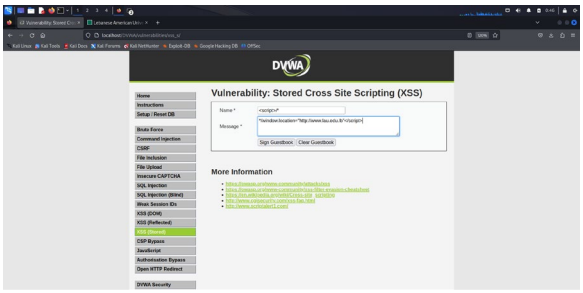
Since no input checks are implemented, we opted to simply insert another *alert script* as input into the textbox. However, upon inspection of the code, we realized that the maximum length of characters in the name and message fields are set to 10 and 50 characters, respectively. The malicious script is as follows:

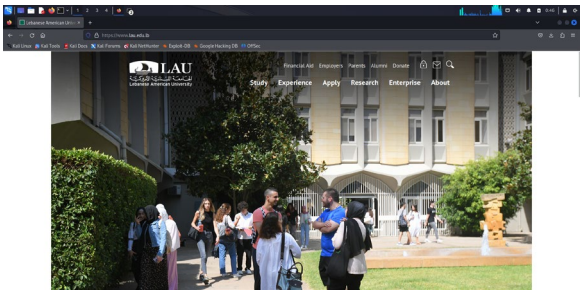
<script>window.location="https://www.lau.edu.lb"</script>



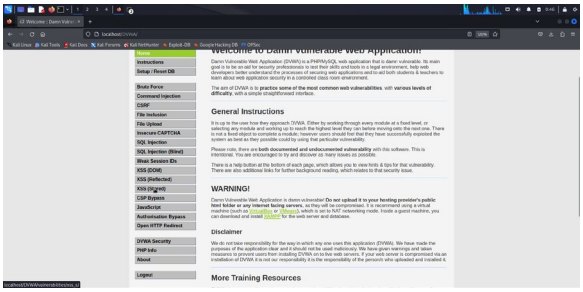


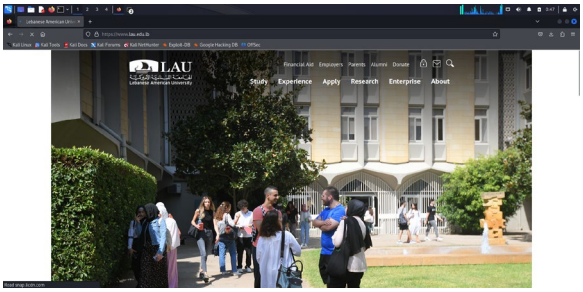
The complete script will fit in the combined length of both fields though, so we opted to start the script in the name field, comment out everything between the two fields, and complete the script in the message field. The redirection is successful and the LAU website loads successfully.

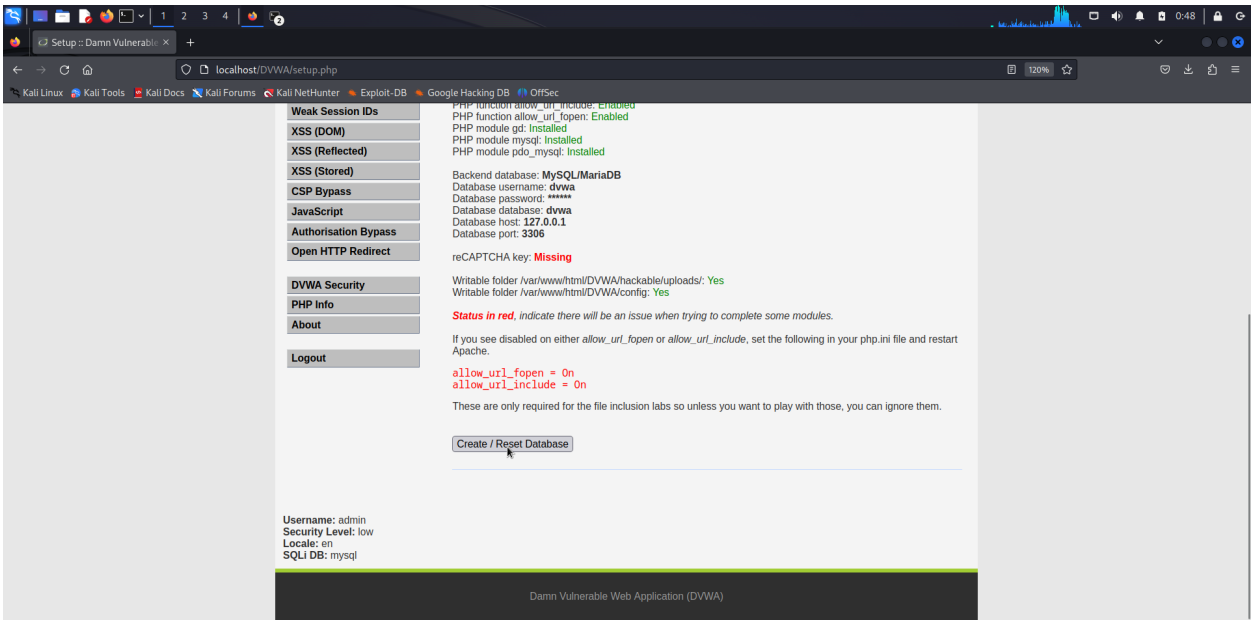




As the XSS is permanent, the redirection occurs every time we navigate back to the XSS page, so we reset the database before moving on to the medium-level challenge.



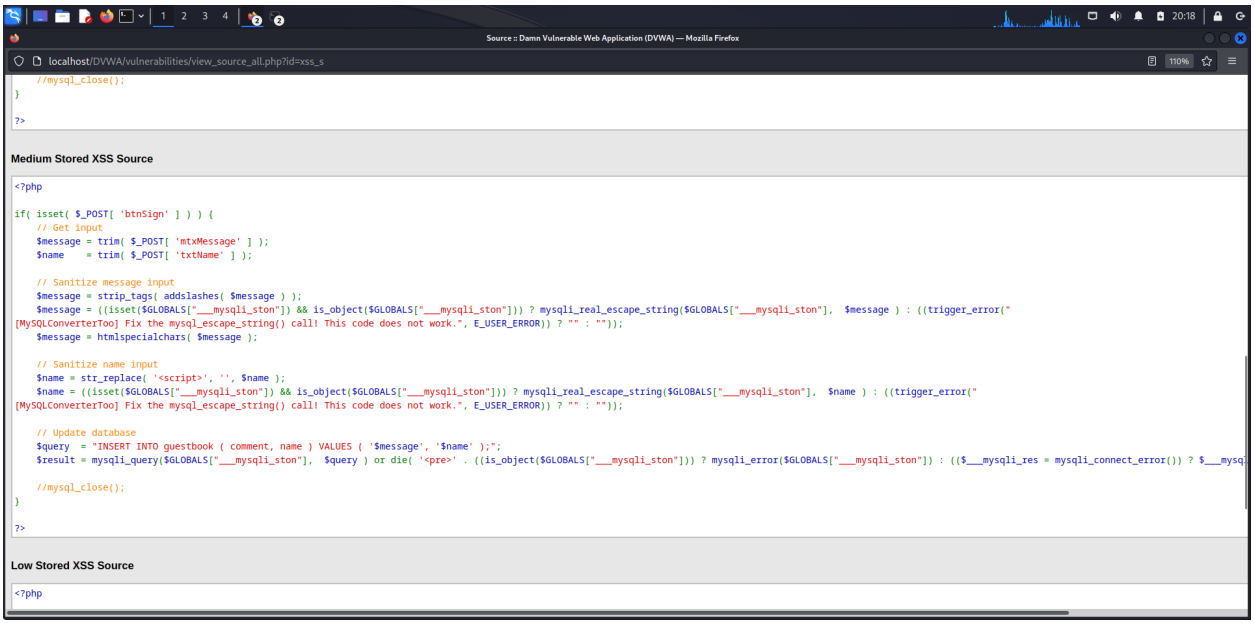




The persistent redirection indicates our successful Stored XSS attack.

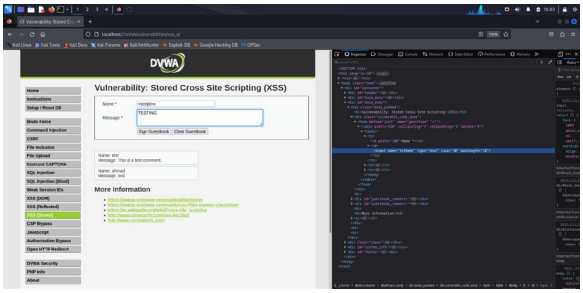
b. Medium

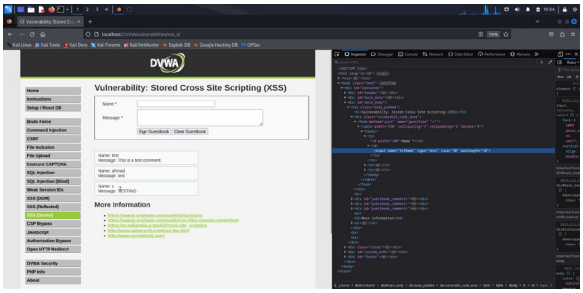
This level's source code is provided by the following:

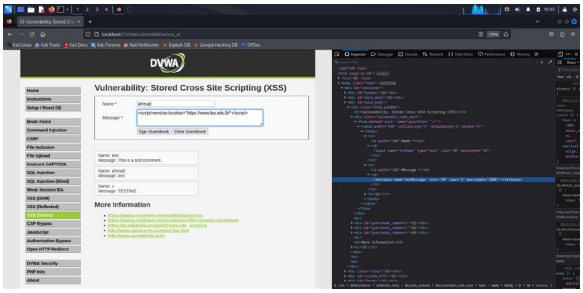


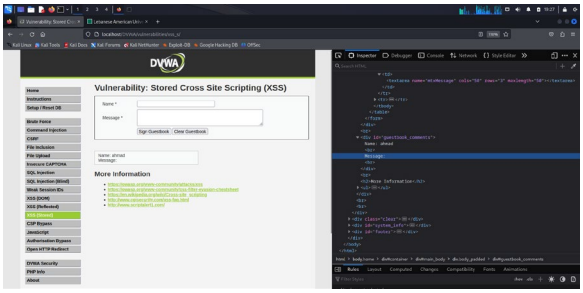
Upon its analysis, it appears that the developer is filtering out all tags in the *message* field input but only the <*script*> tag from the *name* field input.

We, then, tested different inputs to check how they would be handled.



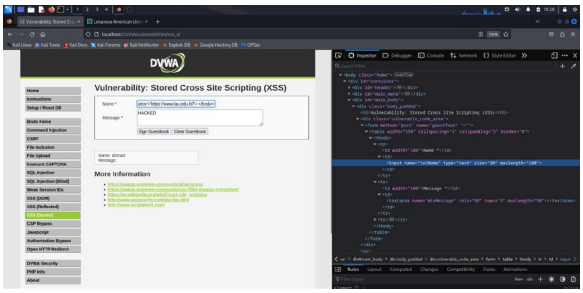


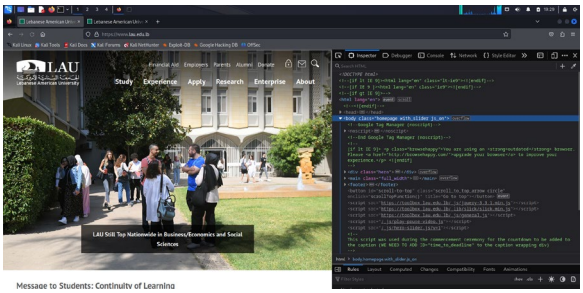




A simple workaround would be to, similarly to the previous XSS attacks, override the *onload* attribute for the *body* element to *window.location=”https://www.lau.edu.lb”* in the name field since all tags are being stripped in the message field. To ensure the input will fit in the name field, its *maxlength* attribute is set to 100 using the browser’s inspect tool. Therefore, the input would be:

<body onload=”https://www.lau.edu.lb”><body>

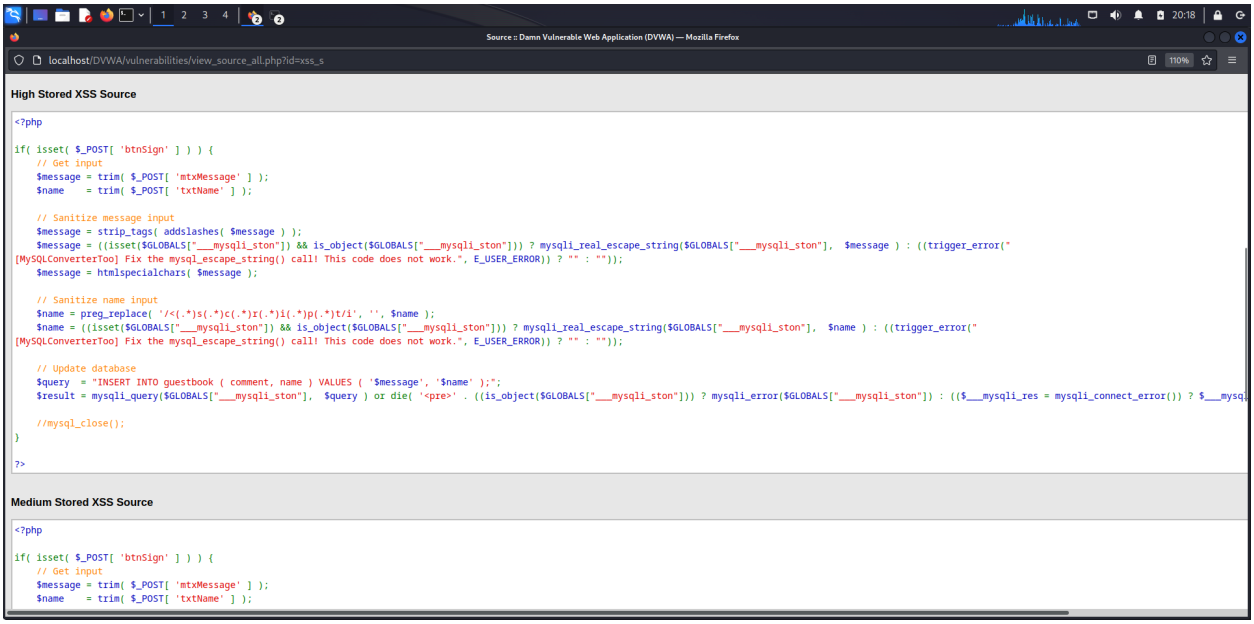




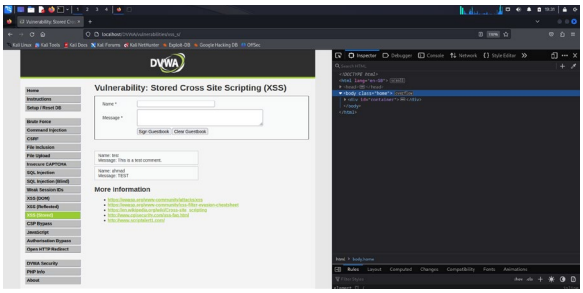
The resulting redirection confirms our successful Stored XSS attack.

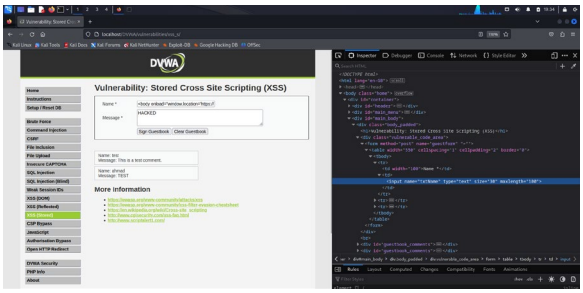
c. High

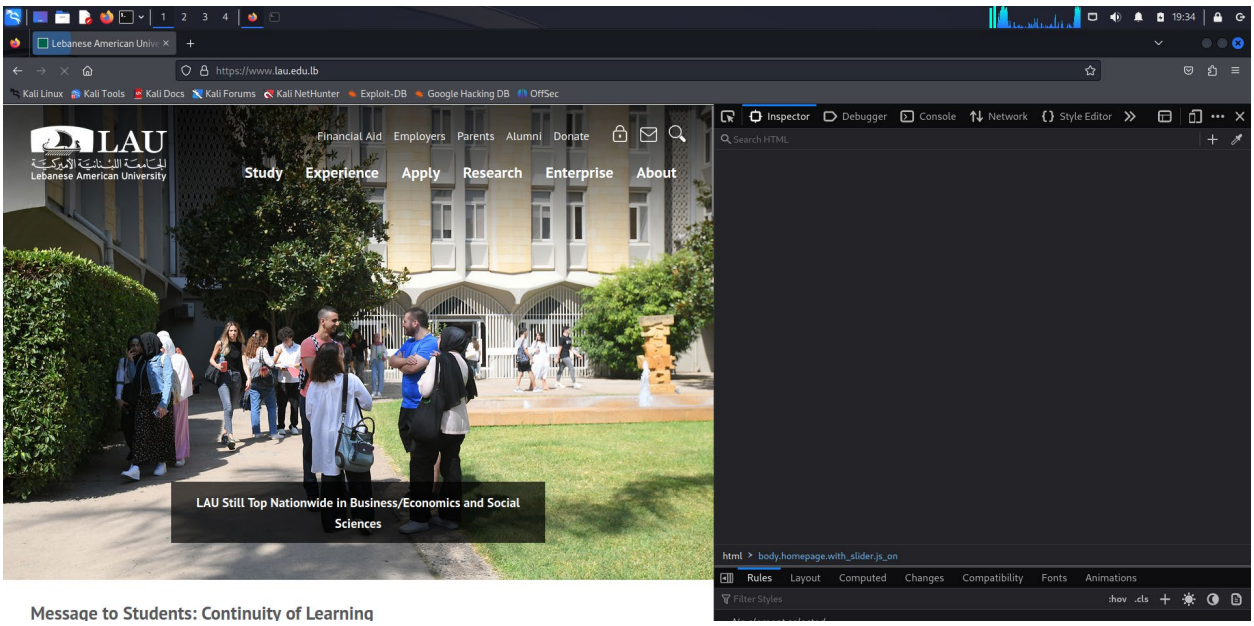
Similarly to the previous XSS attacks, a relatively advanced pattern is used to filter out all possible variations of *<script>*. This is shown in this level’s source code provided by the following:



As our medium-level challenge solution completely disregards the *<script>* tag, we opted to re-implement the same solution for the high-level challenge by changing the *maxlength* attribute of the *name* field to 100 and recording *<body onload=”window.location=’https://www.lau.edu.lb’”></body>*.

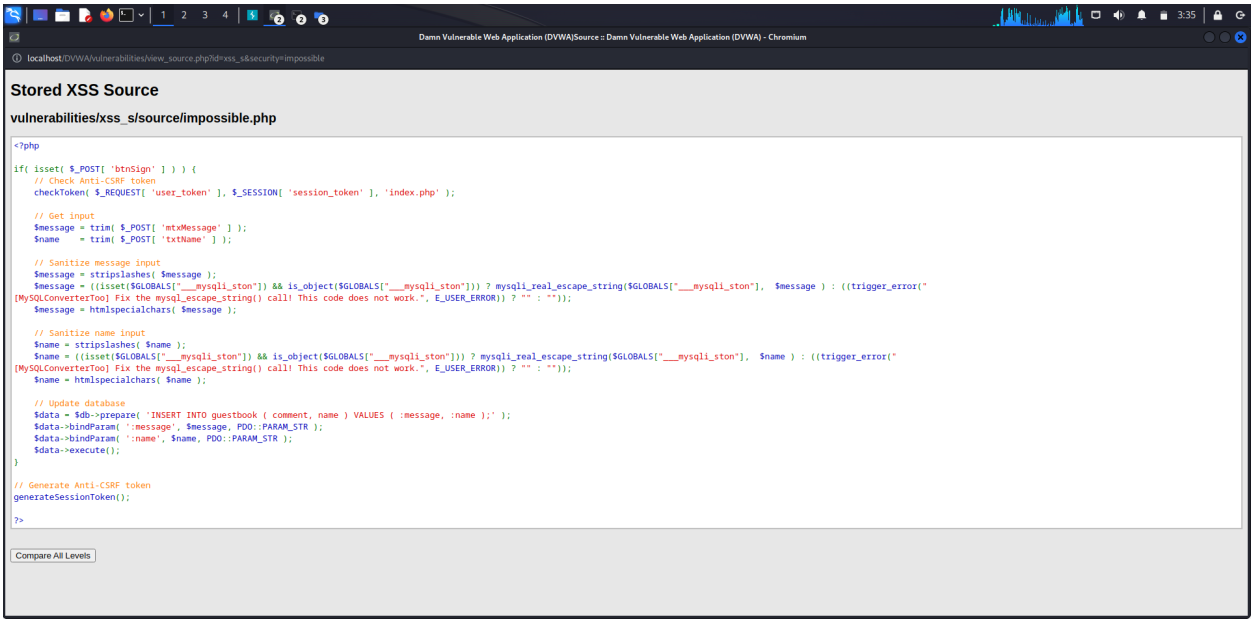






The resulting redirection, again, confirms our successful XSS attack.

d. Impossible



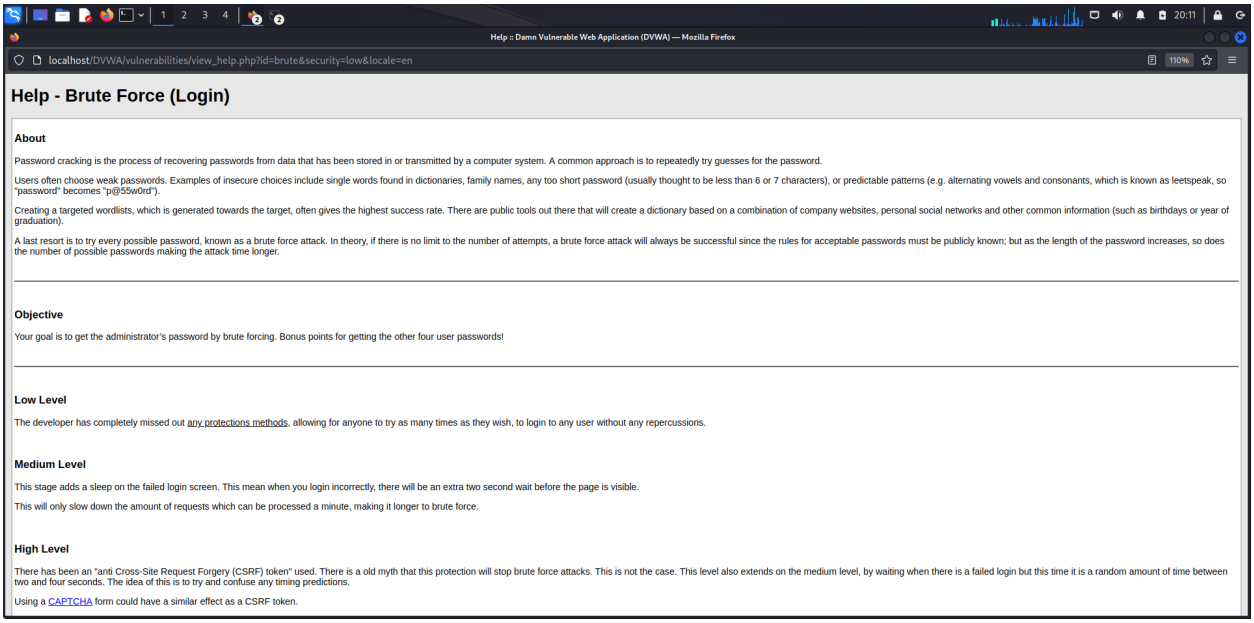
Both user inputs are passed to the *trim()* function that removes any leading or trailing whitespaces as well as certain escape characters found in the string. The inputs are then passed to the *stripslashes()* function that removes all backslashes from the given string. This combination completely filters out any possible variations one might use to reference a page's address.

IV. Other Challenges

1. Brute Force

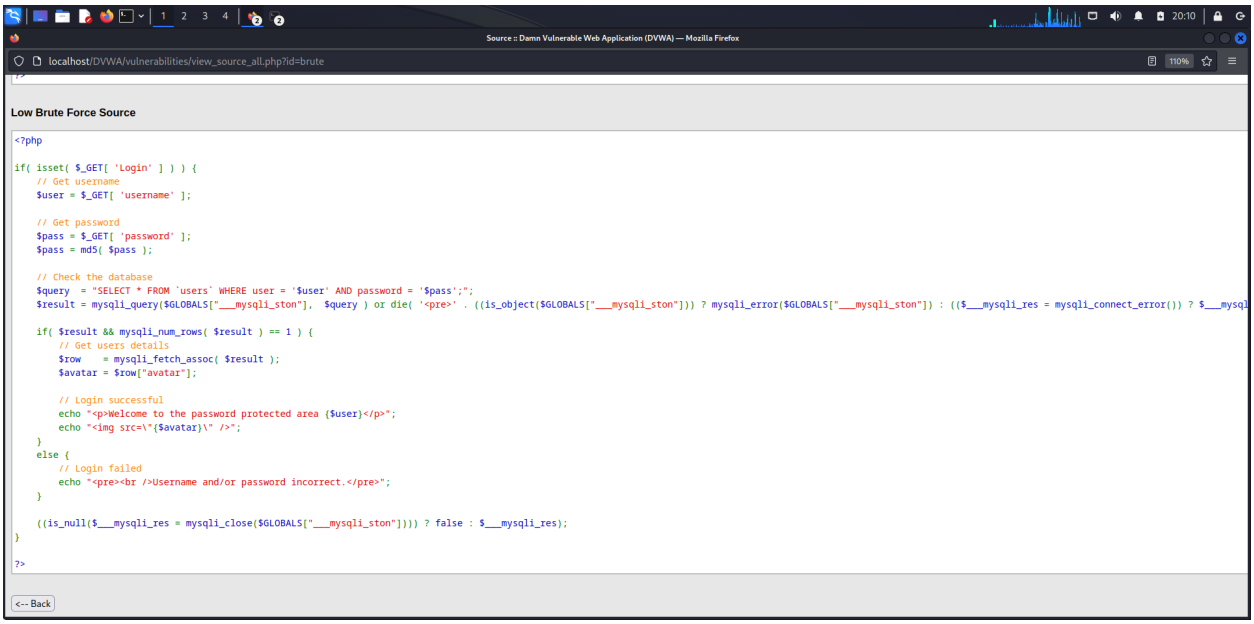
Brute force is a technique where the attacker repeatedly tries to guess the password until it proves successful. Attackers can either try all combinations of the characters available to use to create the password or try a list of common passwords or already compromised passwords. In order to perform the brute force attack, we decided to utilize the *burp suite* tool which is used to intercept, edit and send web requests.

Our aim in this attack is to find the password of the admin user.

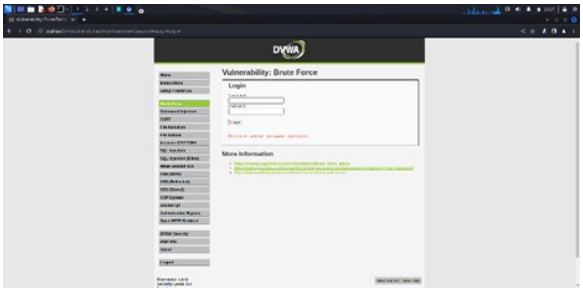


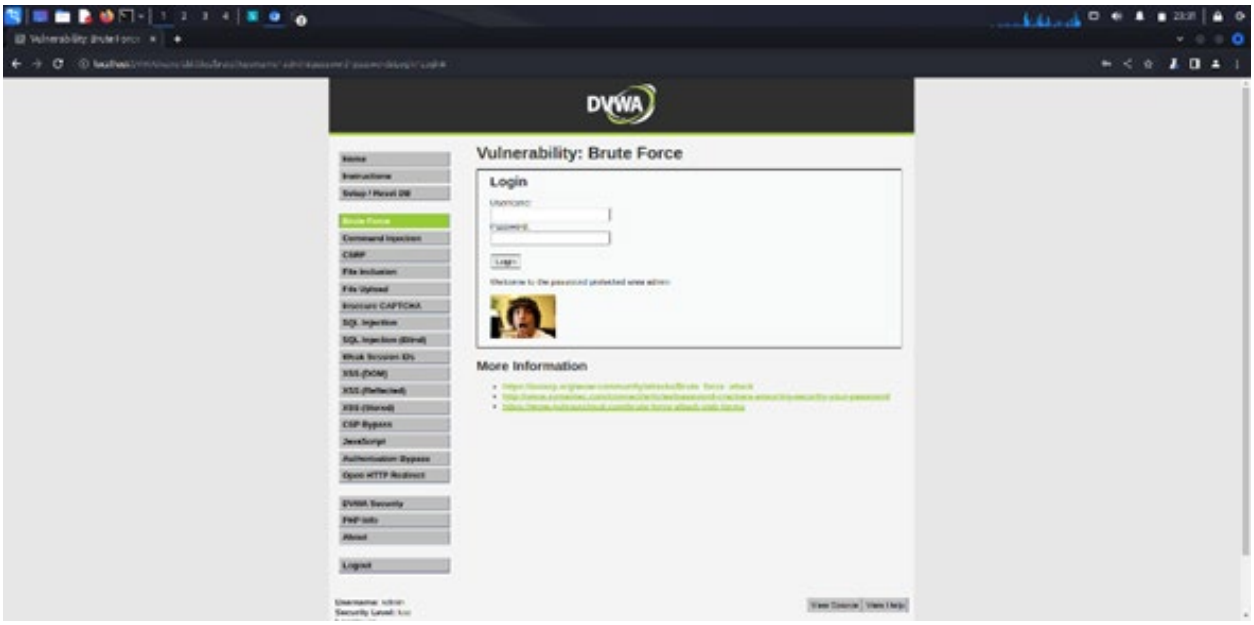
a. Low

At this security level, the website does not perform any validations to check if the user is sending a large number of requests in a sequential manner as shown in the provided source code below:

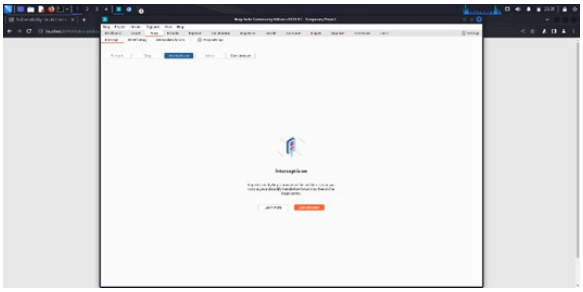


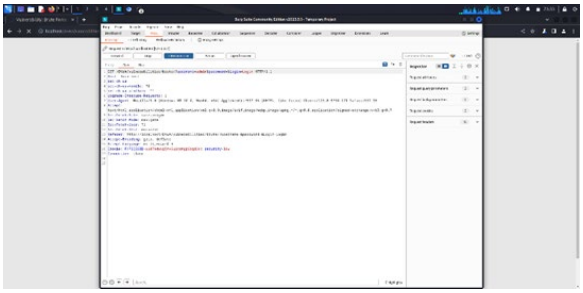
We first decided to test the functionality of the form, so we sent a form which gives a wrong value and a form which successfully logs into the page.



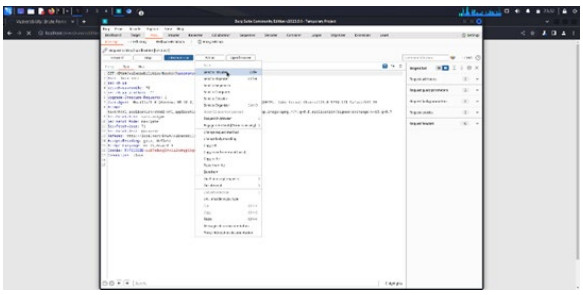


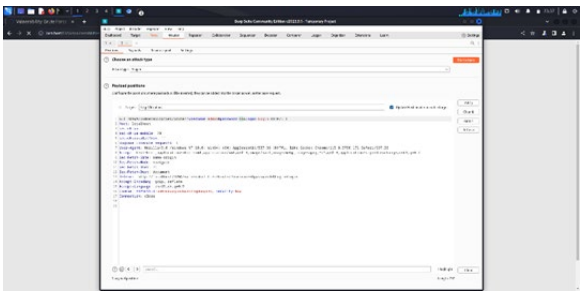
To initiate the attack, we started *burp suite* and enabled request interception to intercept the request being sent by the form.



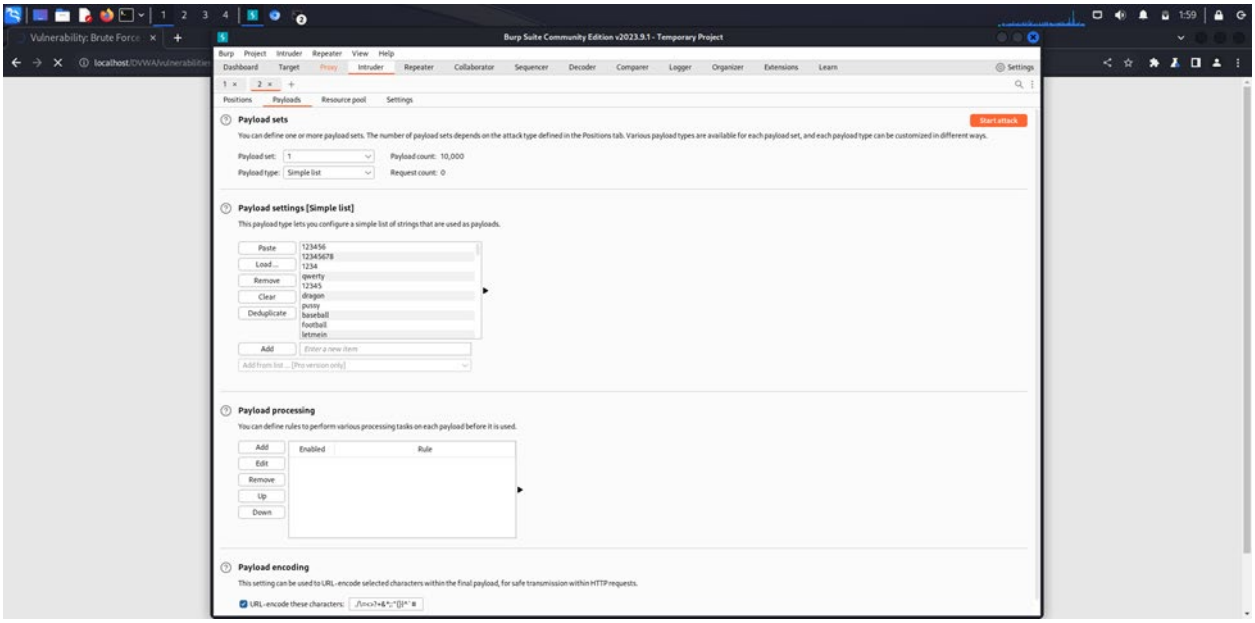


We, next, sent the request to the intruder tool and loaded it within the tool and added a payload position within the password field.

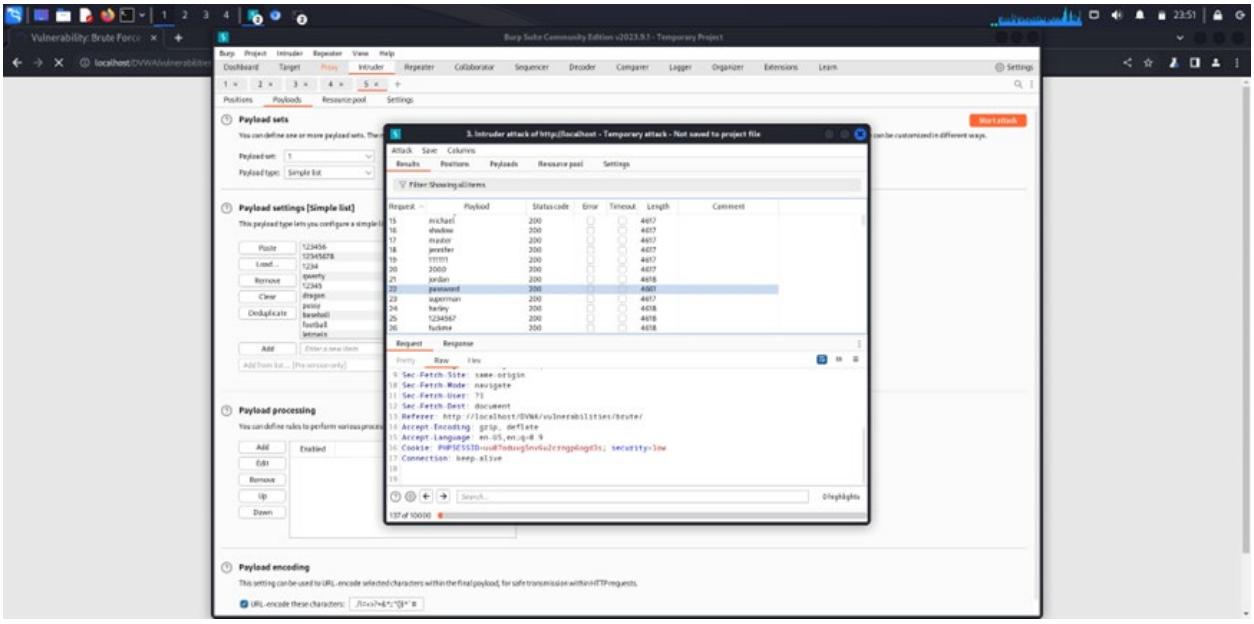




We, then, loaded the list of passwords we would like to try in the form by utilizing a file we downloaded containing the 10000 most common password in a list as a text file.

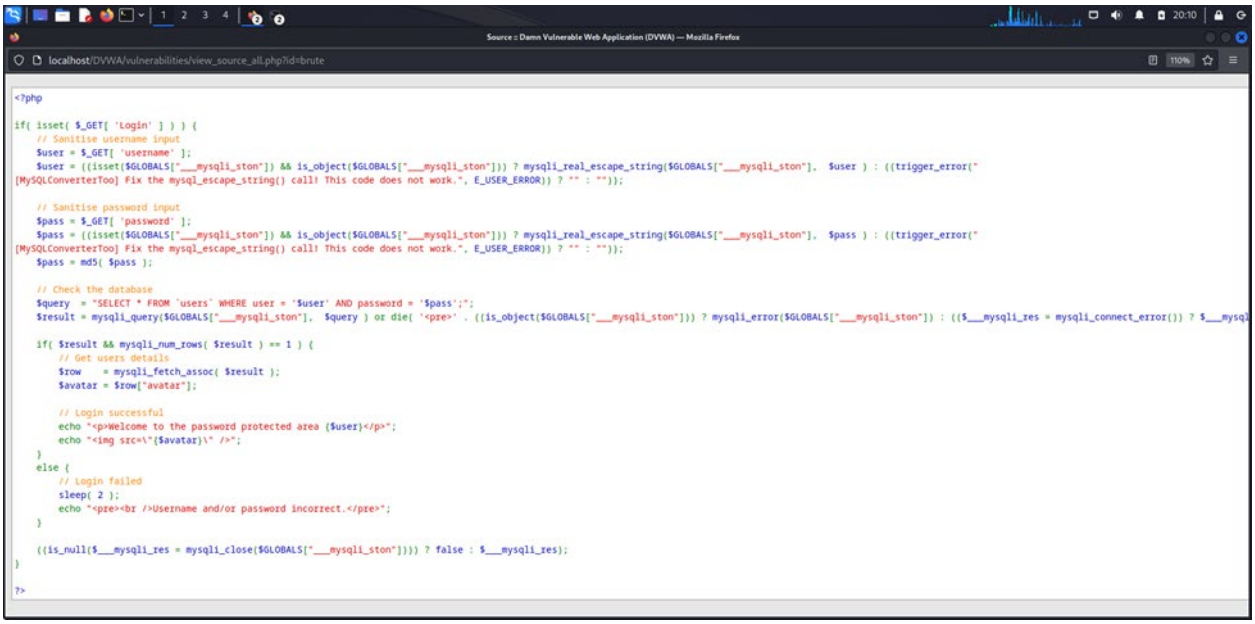


We then started the attack by using the start attack button which sends a request with the username as admin and the password changes according to the value present in the list sequentially. When the attack reaches the password: “password”, the attack returns a response which has a larger length than the rest of the responses which indicates that the attack managed to login using the password “password” and thus revealing the password of the admin user.

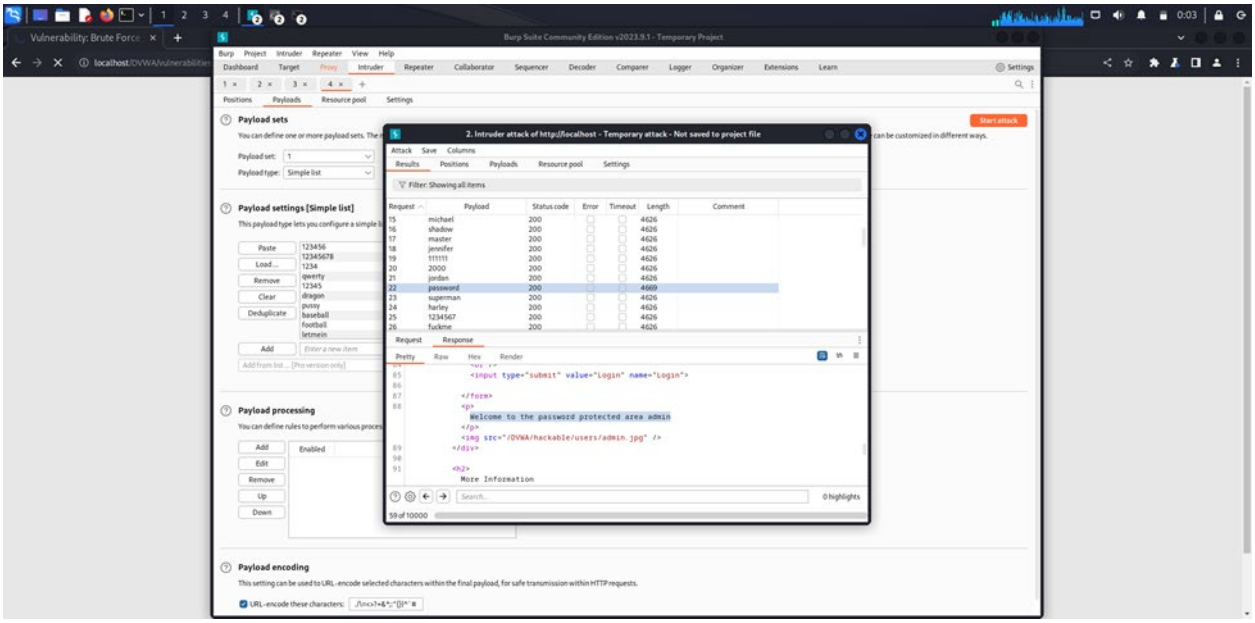


b. Medium

At the medium-security level, the website is checking if the request has failed and punishes the user by not receiving any requests for a 2 sec timeout interval. This measure does not stop the previous attack but helps slow it down. The source code is provided by the following:



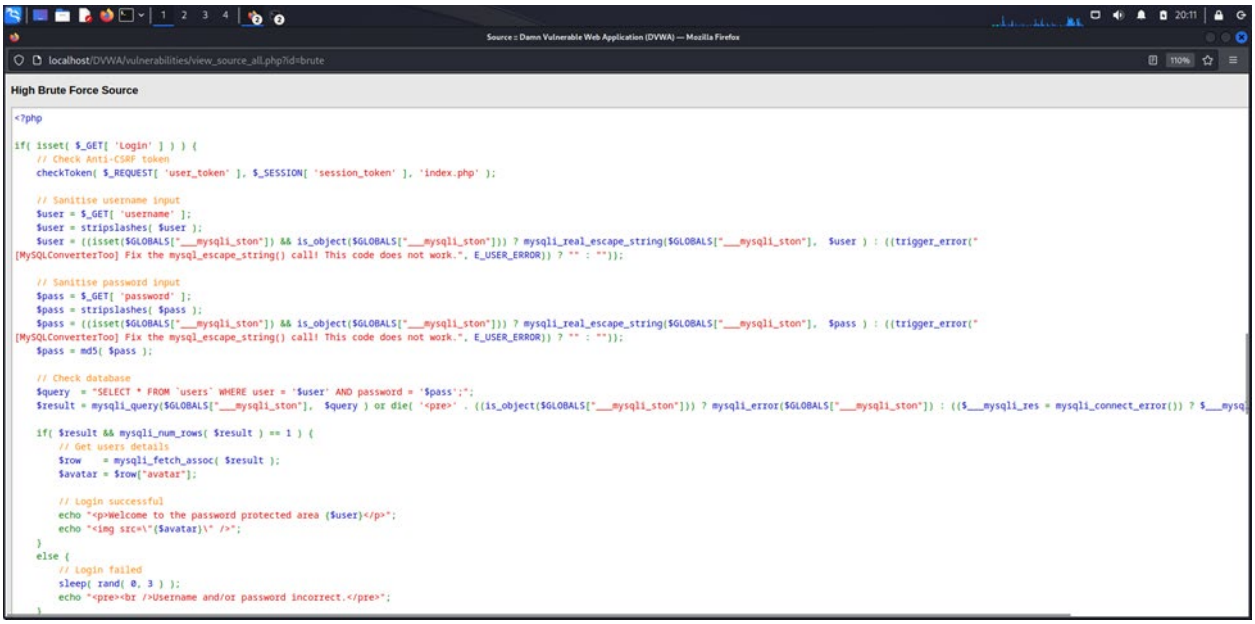
Thus, to complete this challenge, we use the same technique we used in the low challenge.



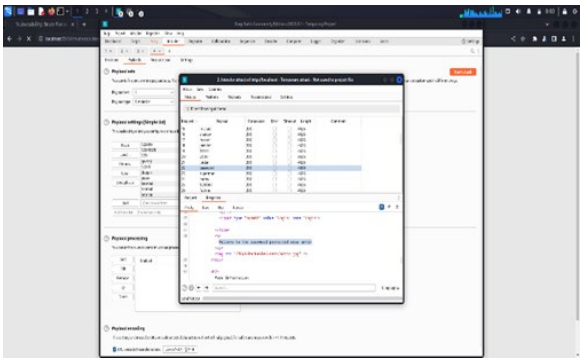
The output shows that we've managed to receive the password of the administrator eventually.

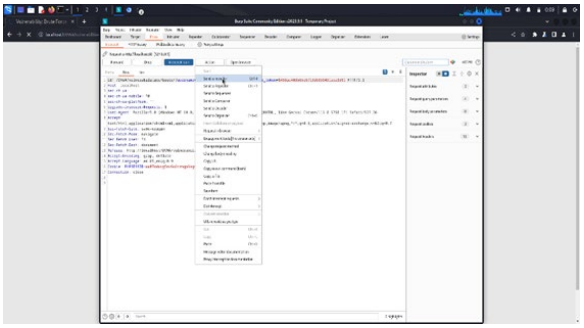
c. High

At this security level, a CSRF token is used when sending a request in order to check the legitimacy of the request being sent. However, whenever a request is sent, a new CSRF token is returned which becomes the valid token to be used. This can be analyzed in the provided source code below:

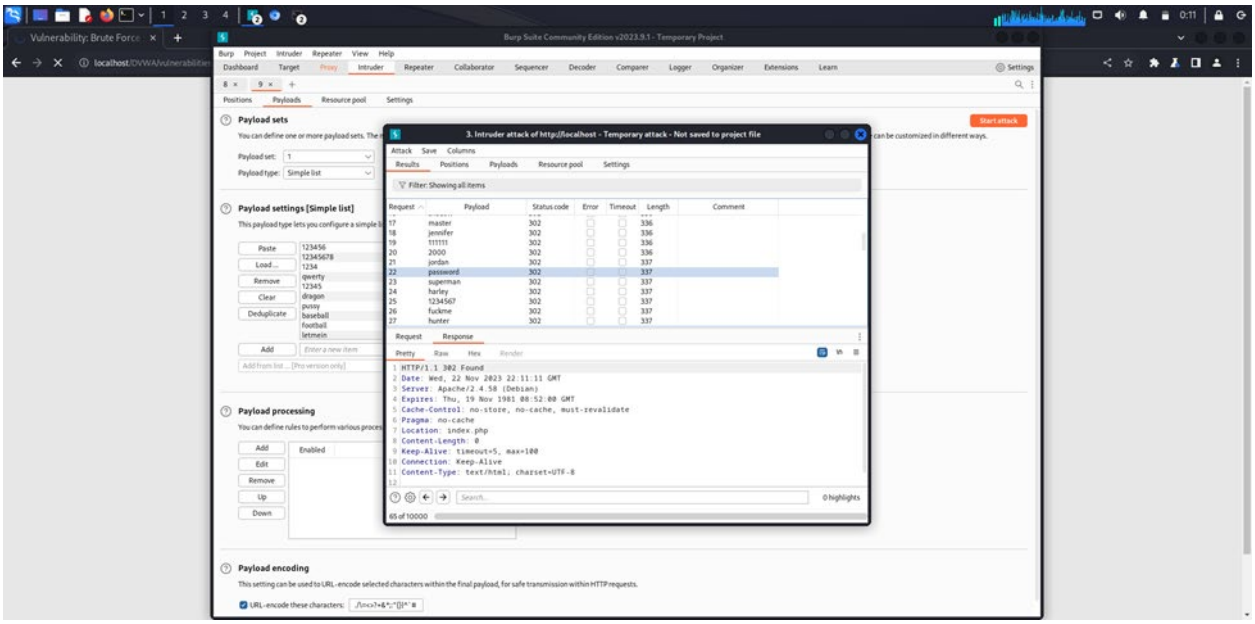


In order to receive the password of the administrator, we once again use the burp suit tool. So, we intercept the request and load it into the intruder tool.

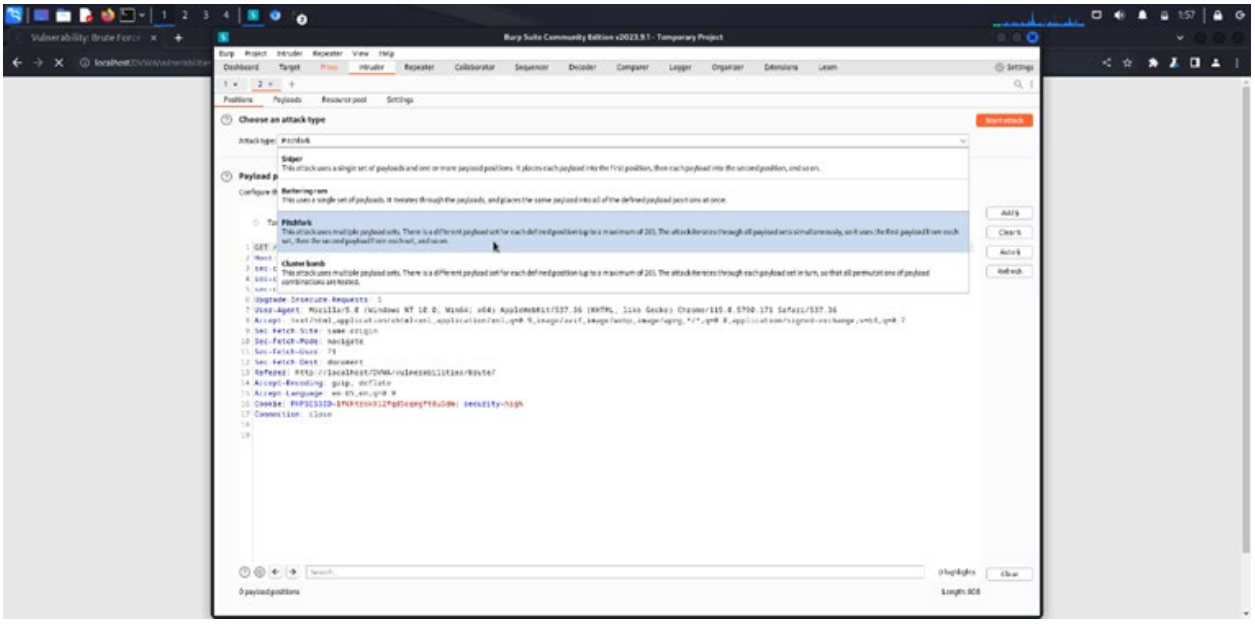




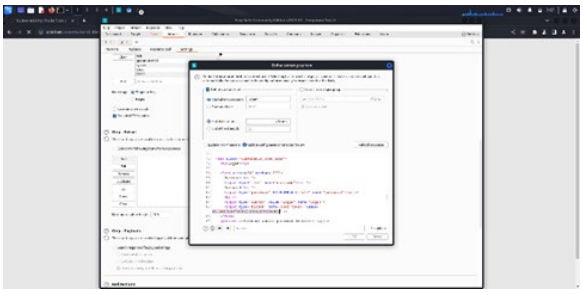
We tried to run the same brute force method as before but it fails, as expected, because it tries to use the same token for all requests.

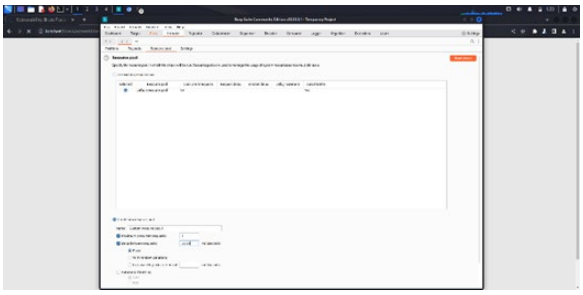


In order to have the brute force method to work, we use the pitchfork attack and select the password field and the user_token field as the payload parameters for this attack.

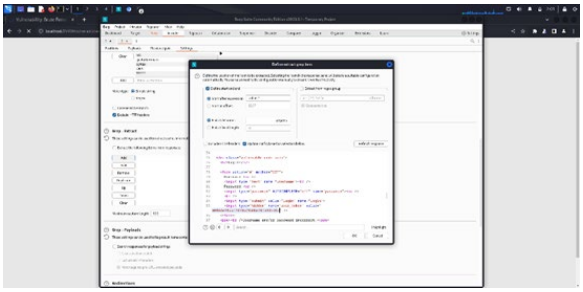


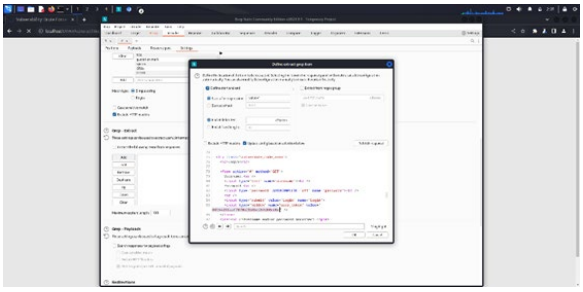
We then loaded the list of most common passwords into the first payload which represents the password field. Then we create a new recourse pool which sends a request one at a time in order to receive the token present in the response of the previous request.



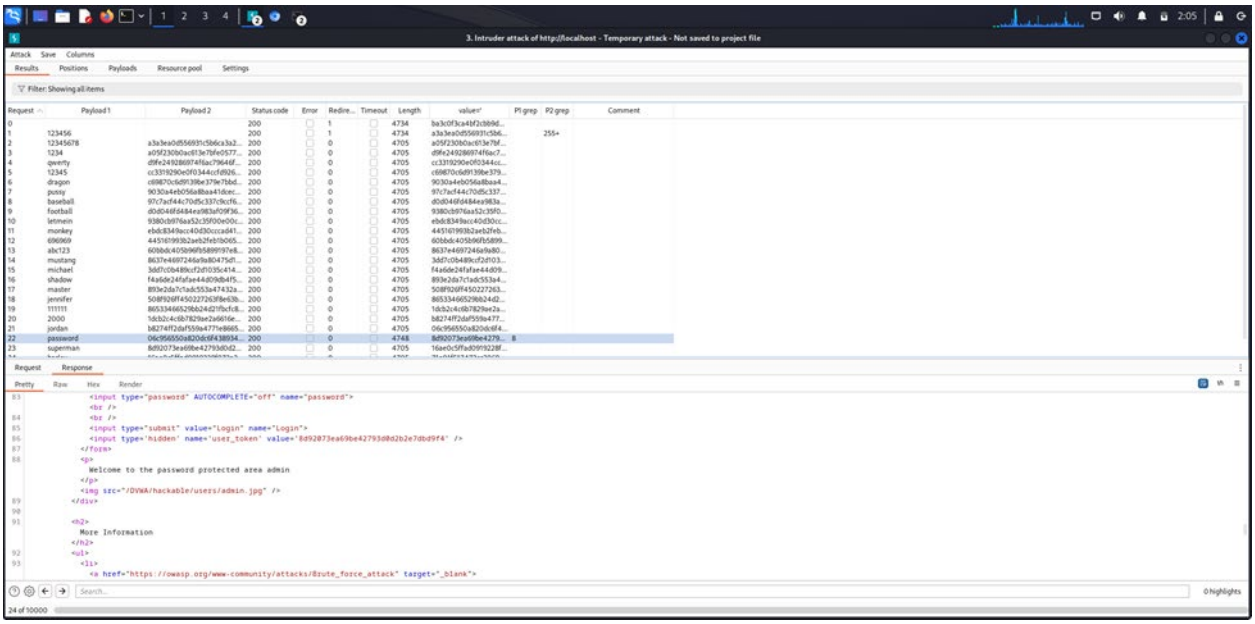


We define a new grep item which returns the value of the token to be using in the next request and set redirections to on-site only in order to perform the necessary redirection to the next line. The second payload then takes uses the recursive grep and set the regular expression matching to the one defined in the grep settings.

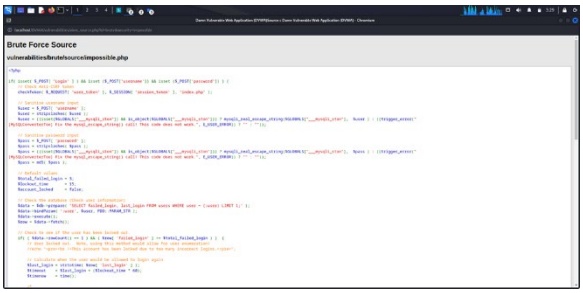


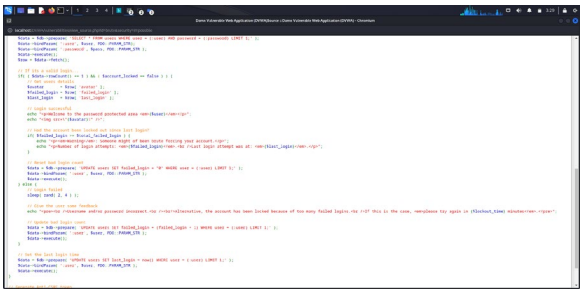


We finally run the attack which gives a login when the password used is “password”.



d. Impossible

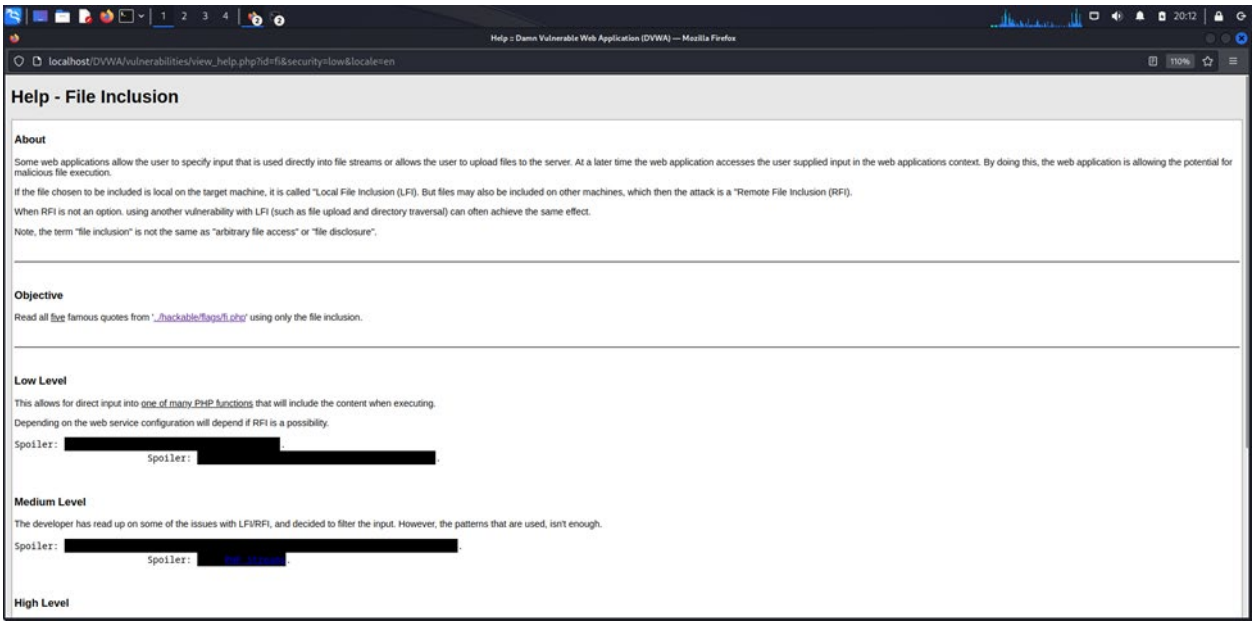




This code shows that the developer locks the account of the user if he tries and fails to login within three tries. This could be clearly seen through the query under the //Update bad login count which counts the number of times the user tried to login to the website.

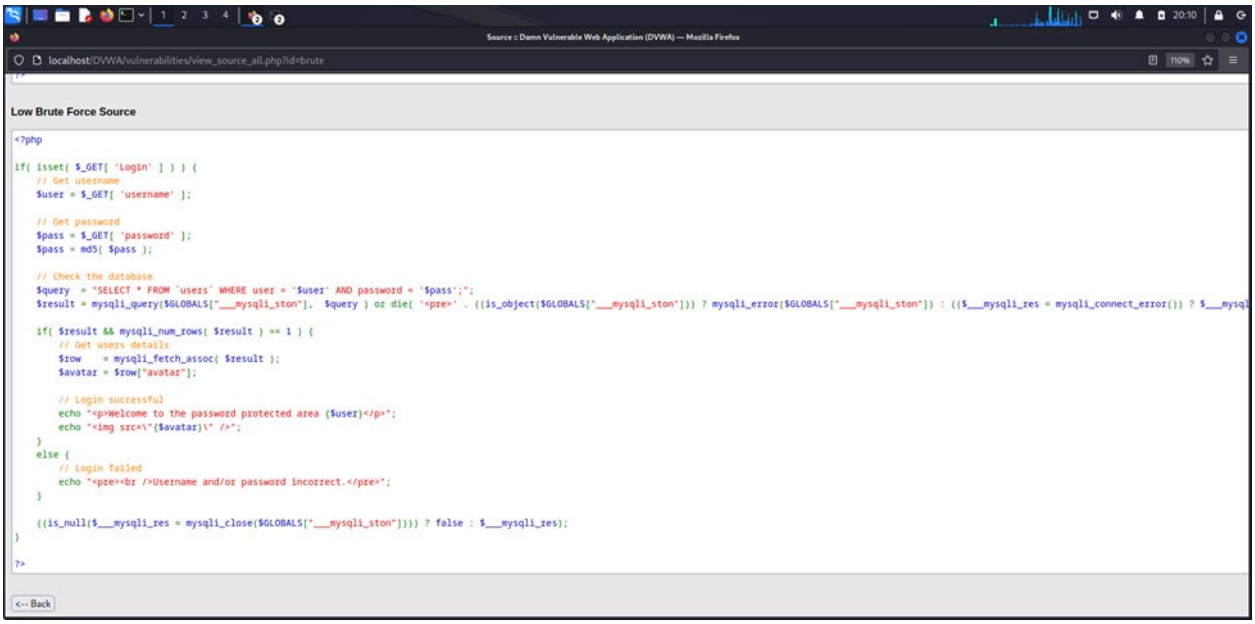
2. File Inclusion

A file inclusion attack occurs when the website gives the user access to an input which is used in order to load files on the server. Our objective is to find the file which is stored in "../hackable/flags/fi.php".

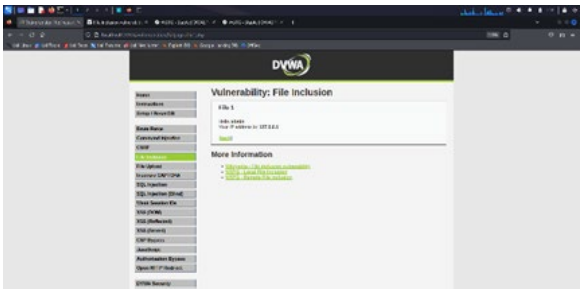


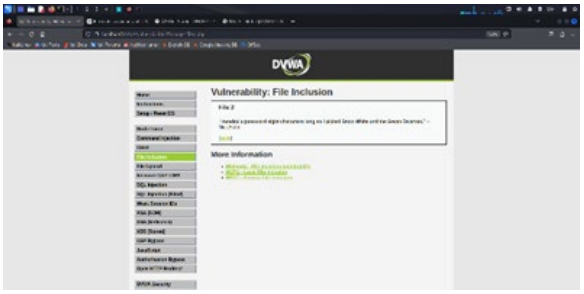
a. Low

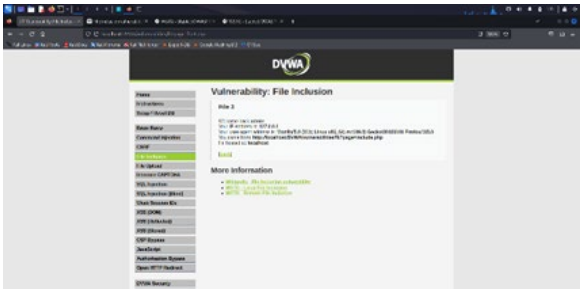
At this level, the website allows access to the file system without any checks being made which ensure the safety of the system.

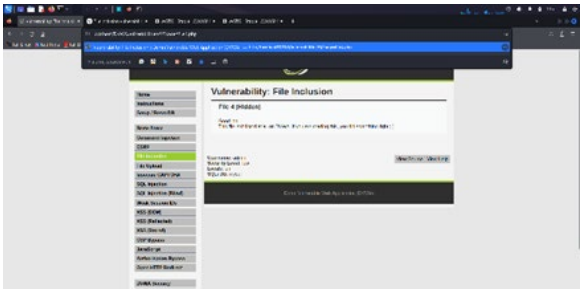


Thus, we first start by checking out the functionality of the website.



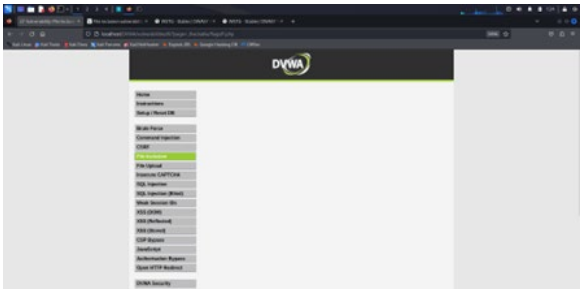


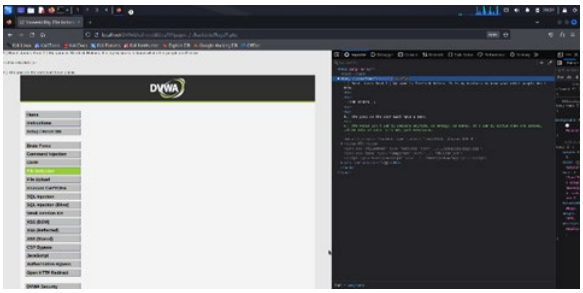




We found file 4 by changing the URL to have *path=file4*.

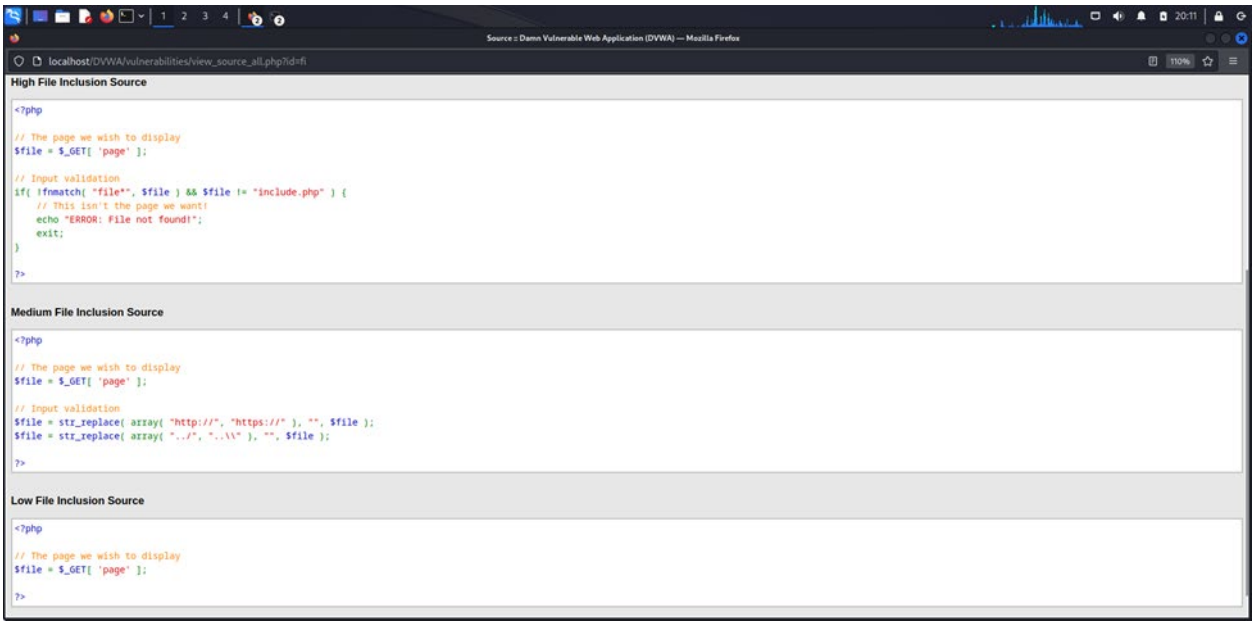
In order to open the fi.php file to find all the quotes, we modified the URL to the include the following path "../hackable/flags/fi.php" however, the file was not found, so we tried to add another ../ which showed the content of the files.



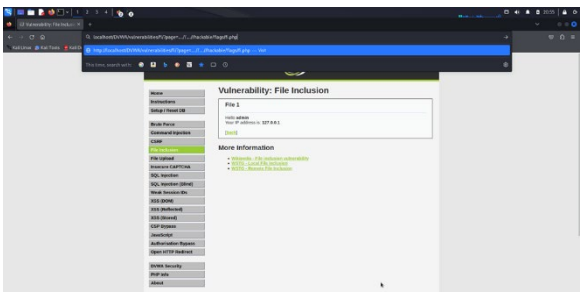


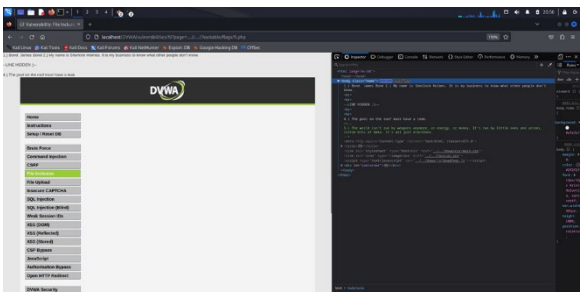
b. Medium

In this level, the website checks for ../, ..\\ and http://, https:// substrings within the path and replaces them with empty quotations.



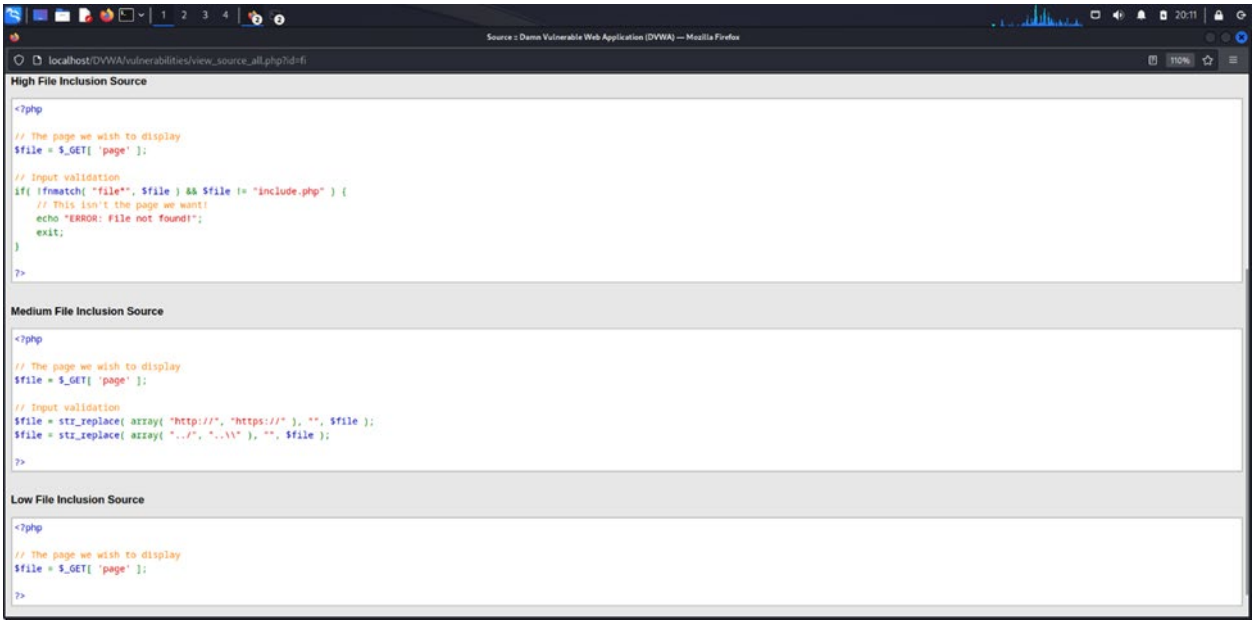
In order to bypass this level, we replaced ../ with …..// because the function will only detect the middle three characters "../" and keep the ".." and the "/" which will then combine to make "../"





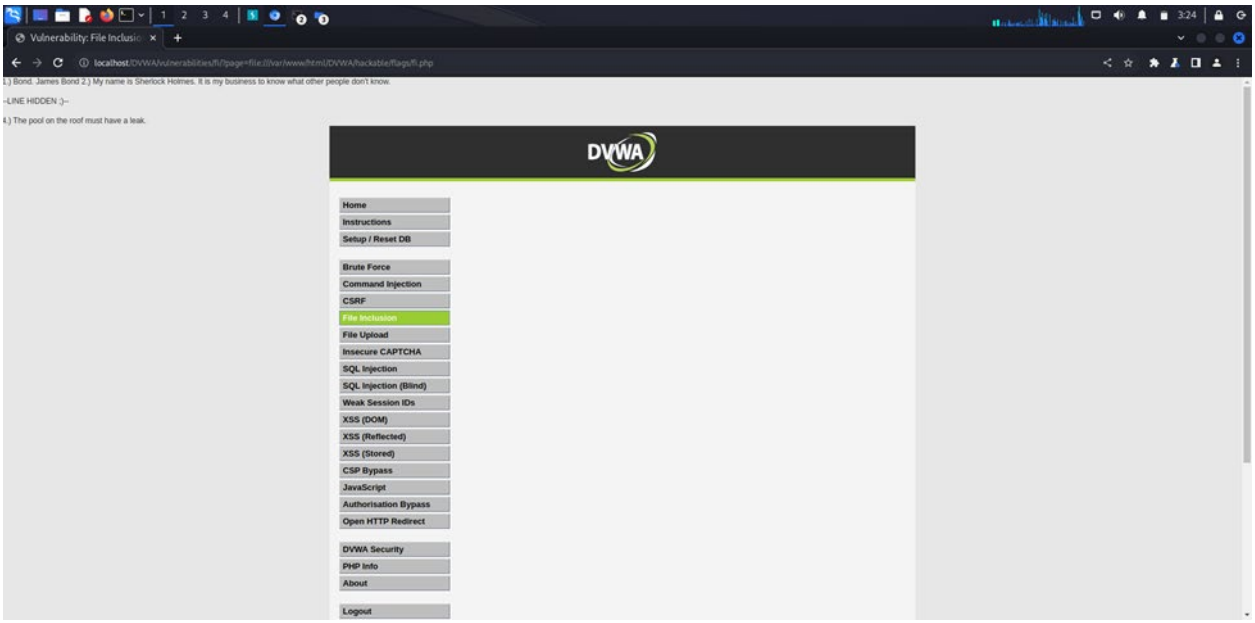
c. High

At the high security level version of this code, the programmer is checking and only allowing files which contain the word “file” which requires us to change the tactic used in the previous part.

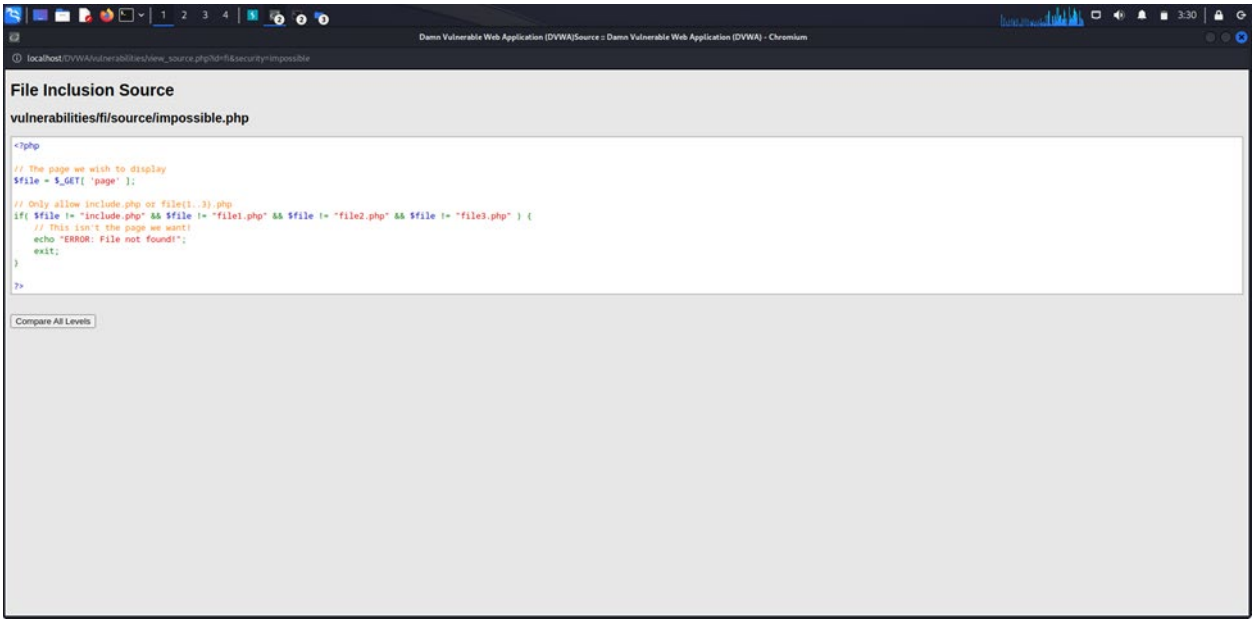


If we use the previous solution, we get a “file not found” error.

In order to complete this level, we managed to use the file:/// header which is used to open files in browsers so, we used the path file:///var/www/html/DVWA/hackable/flags/fi.php in order to open the file containing the quotes.



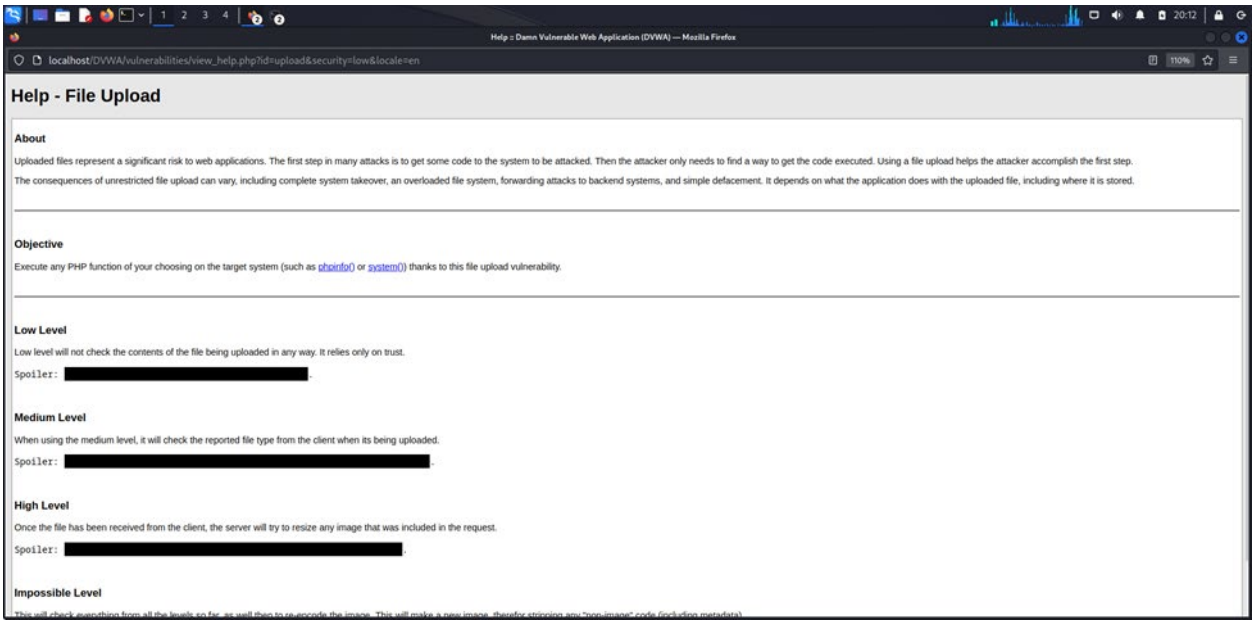
d. Impossible



At this level, the developer checks only for the files which are needed within the page and rejects any other attempt to maneuver through the file system.

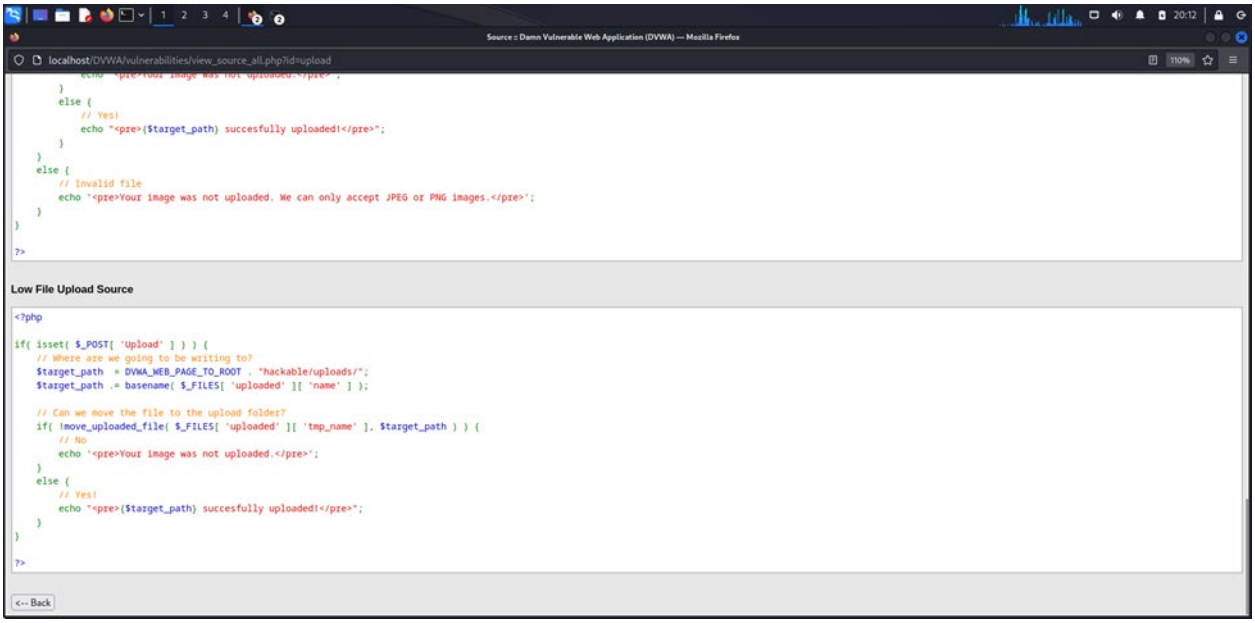
3. File Upload

File uploads present a major risk when it comes to the safety of the system. An attacker can upload a malicious code instead of what is being requested. The attacker then only needs find a way in order to execute the file uploaded. In this attack, we are required to upload a php file and execute it.



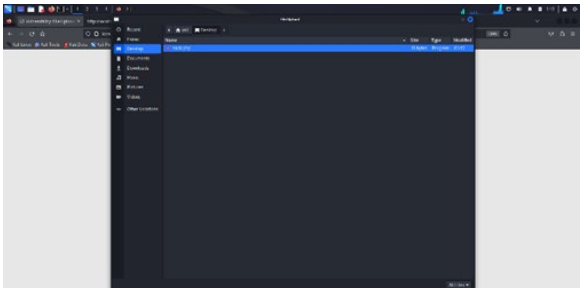
a. Low

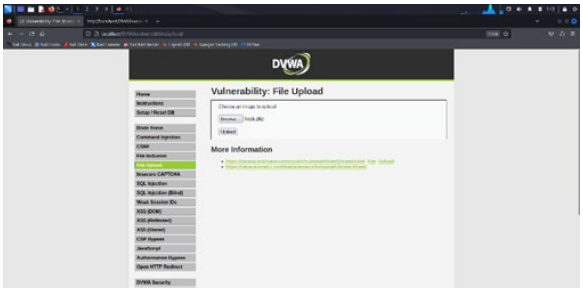
At this level, the system does not perform any check on the file being uploaded.

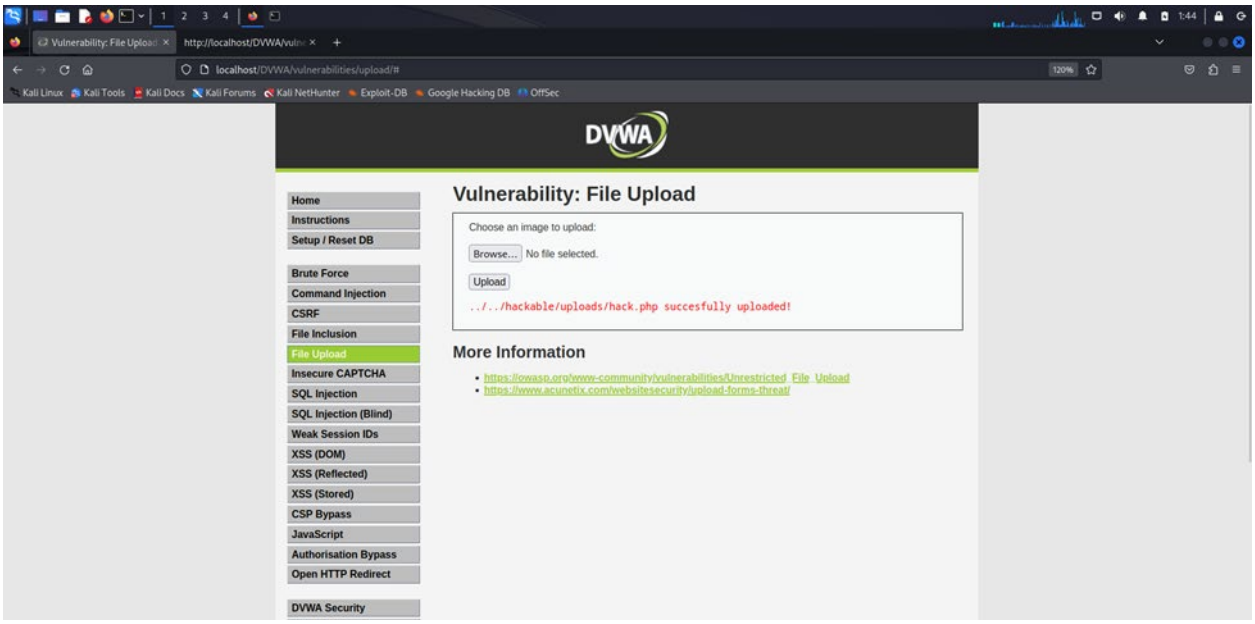


In order to initiate the attack, we created a php file containing the malicious code.

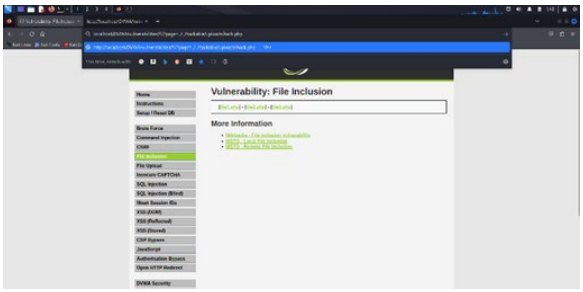
We then uploaded the file which we had just created.

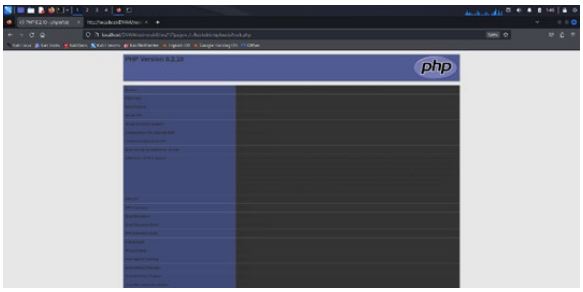






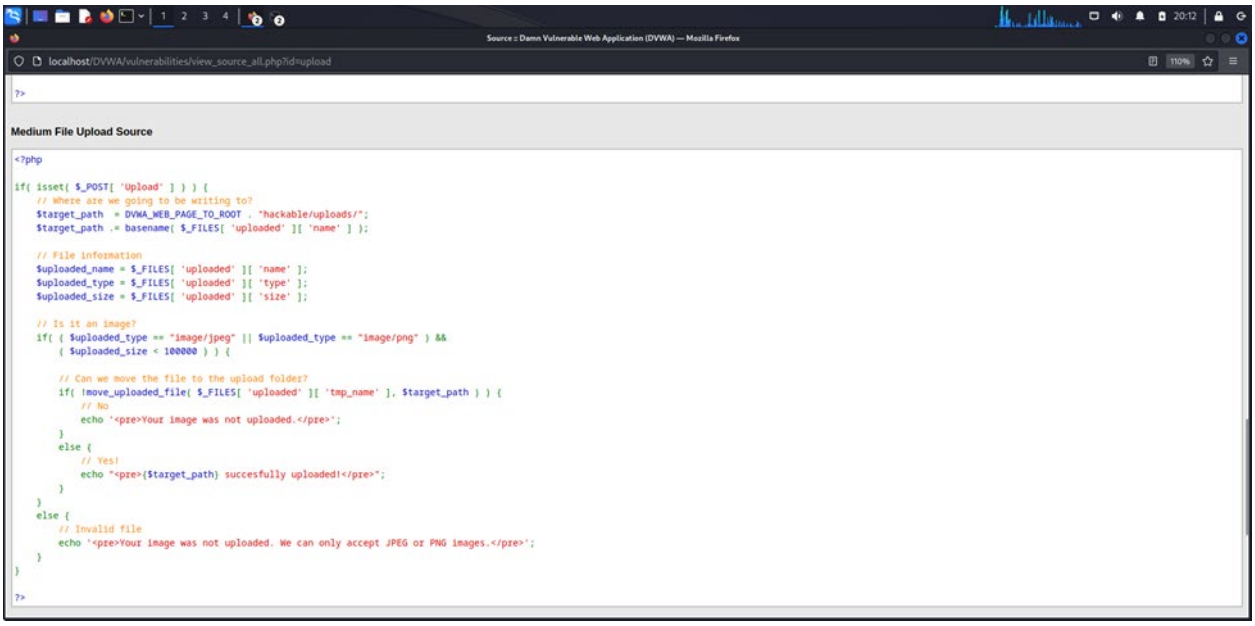
Now, in order to execute the file we had uploaded, we utilized the *File Inclusion* attack which allows us to load the file.



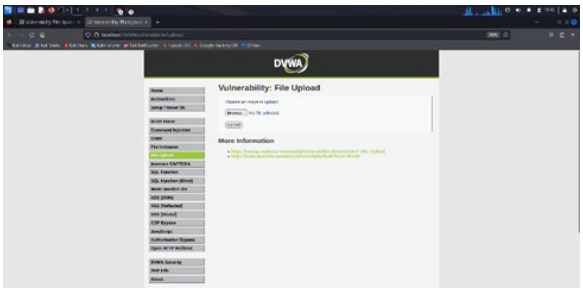


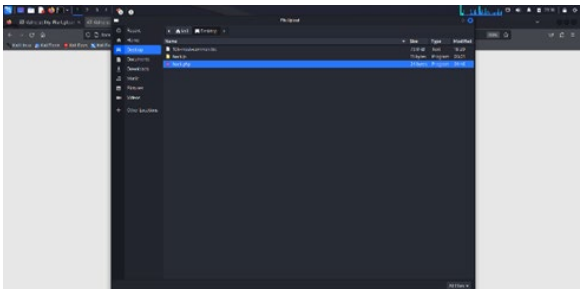
b. Medium

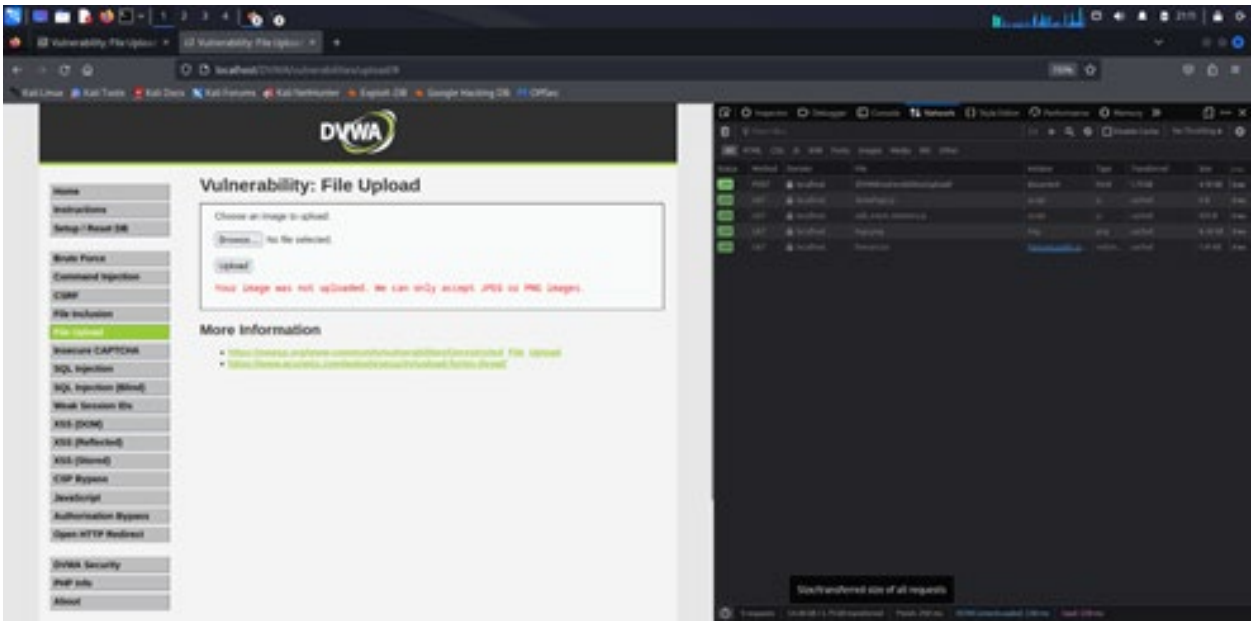
At this level, the system starts to add a bit of restrictions on the type of file being uploaded. In the following code, the developer is checking to see if the type of file uploaded is an image.



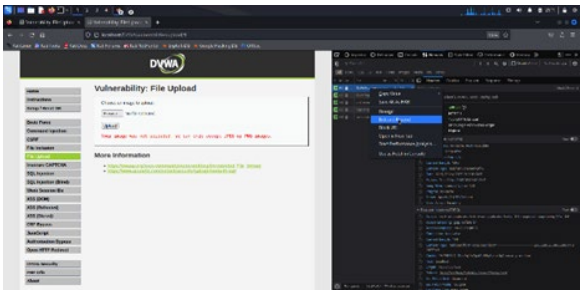
In order to bypass this restriction, we can send the file with the image type while still being a malicious php file. We first try and upload the file we created.

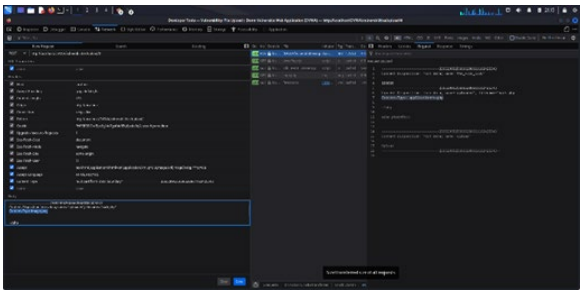


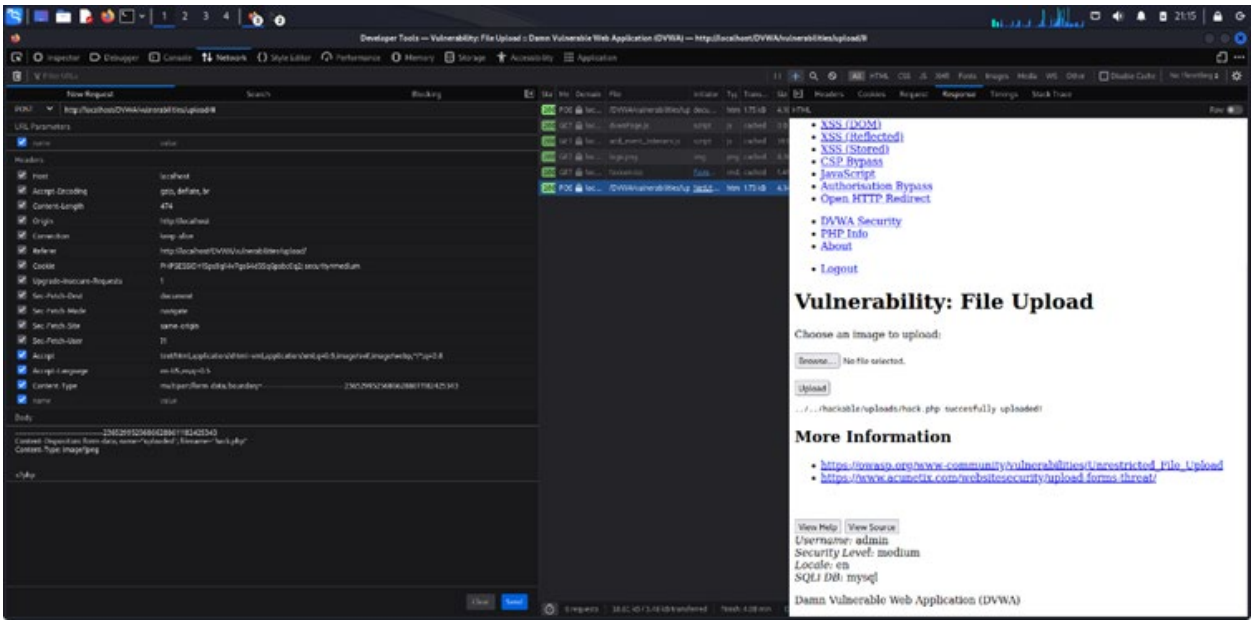




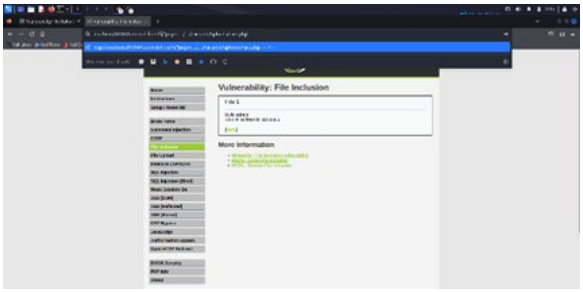
We then open the failed upload request and inspect it in order to modify the file type being sent from php to Image/jpeg. We then send another request which uploads the php file which does so successfully.

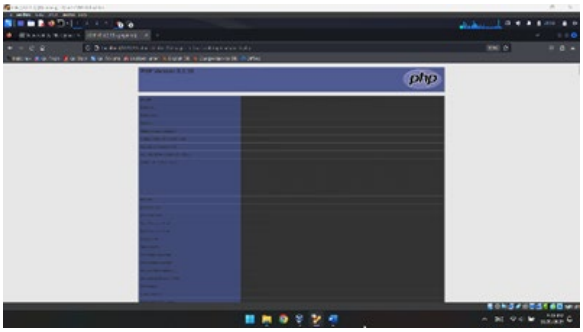






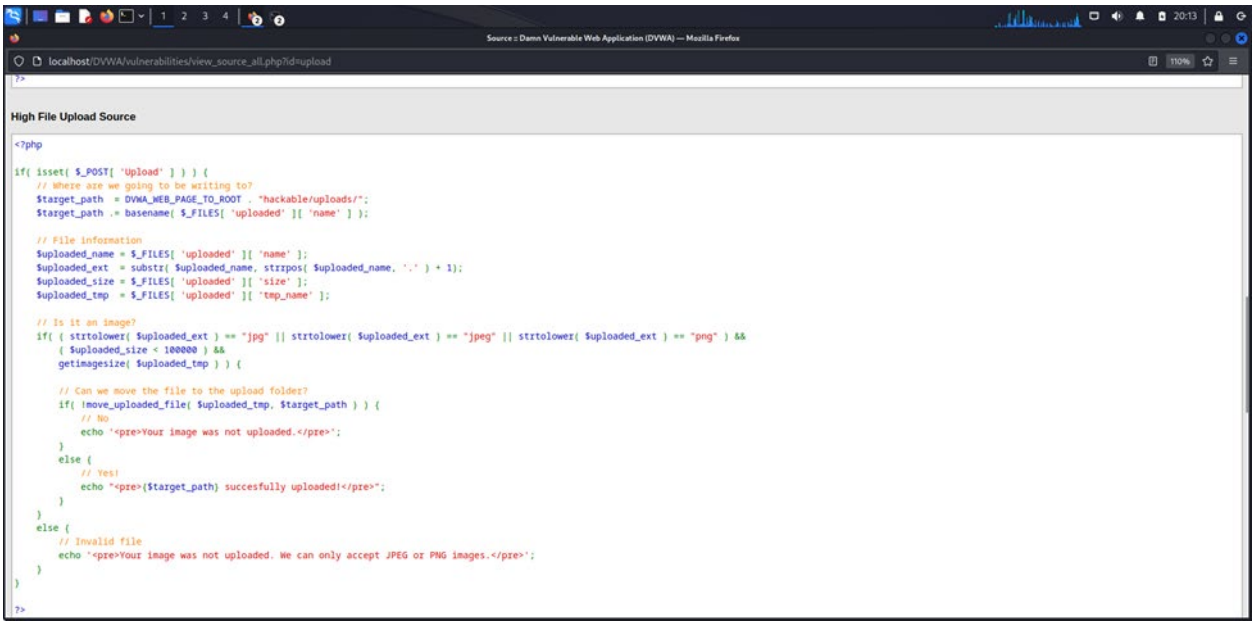
We then execute the file by using the same method as above, the file inclusion vulnerability.



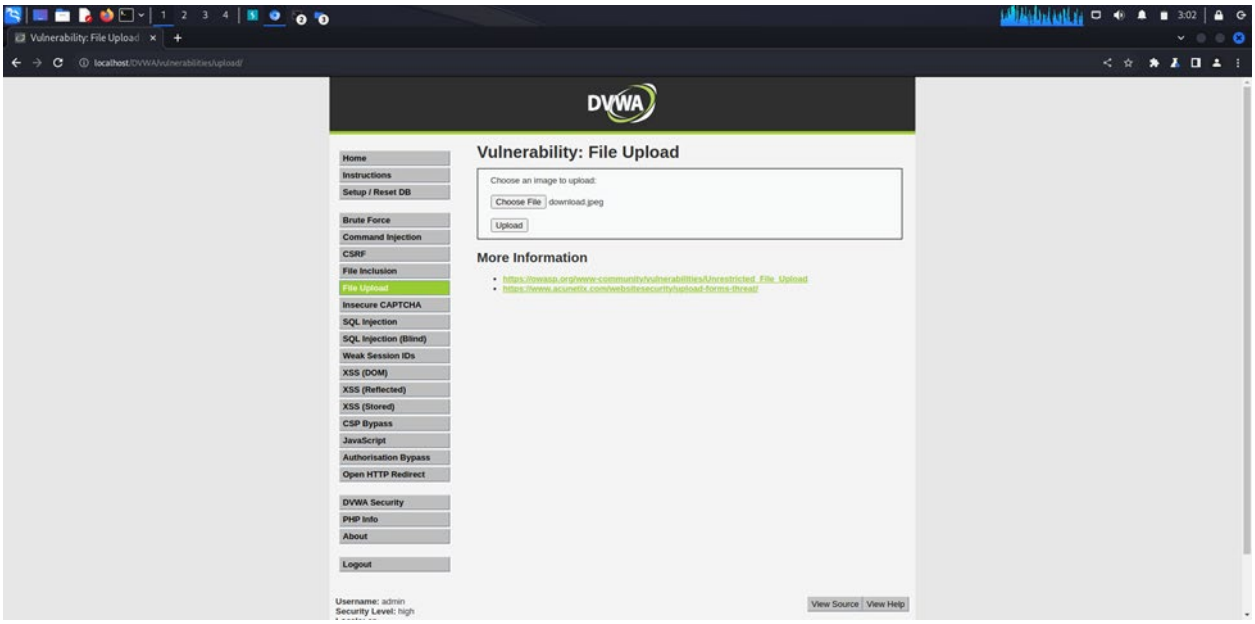


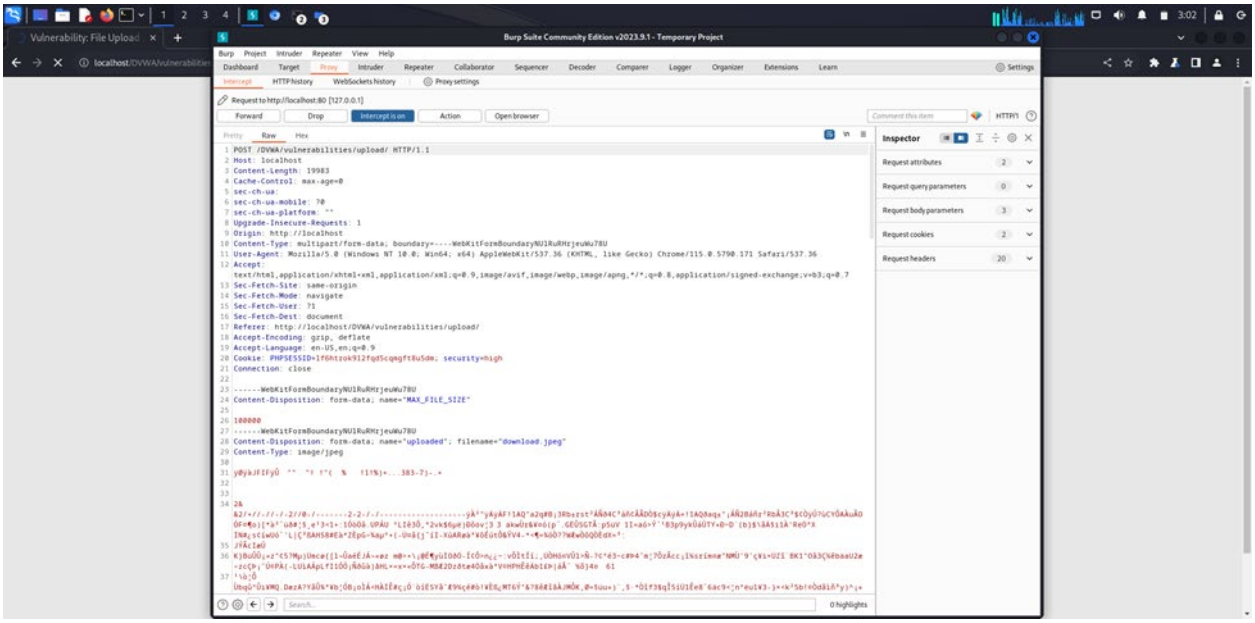
c. High

At this level, instead of checking for the file type of the uploaded file, the developer is looking for particular file extensions.

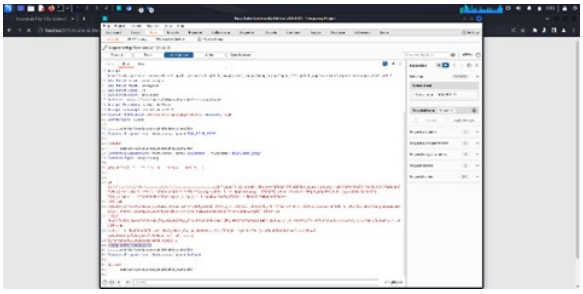


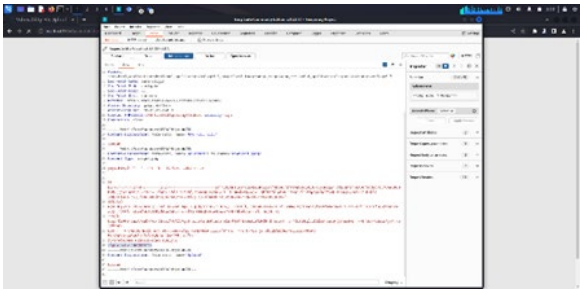
In order to bypass this feature, we decided to intercept the request using burp suit and editing the contents of a normal image being uploaded.

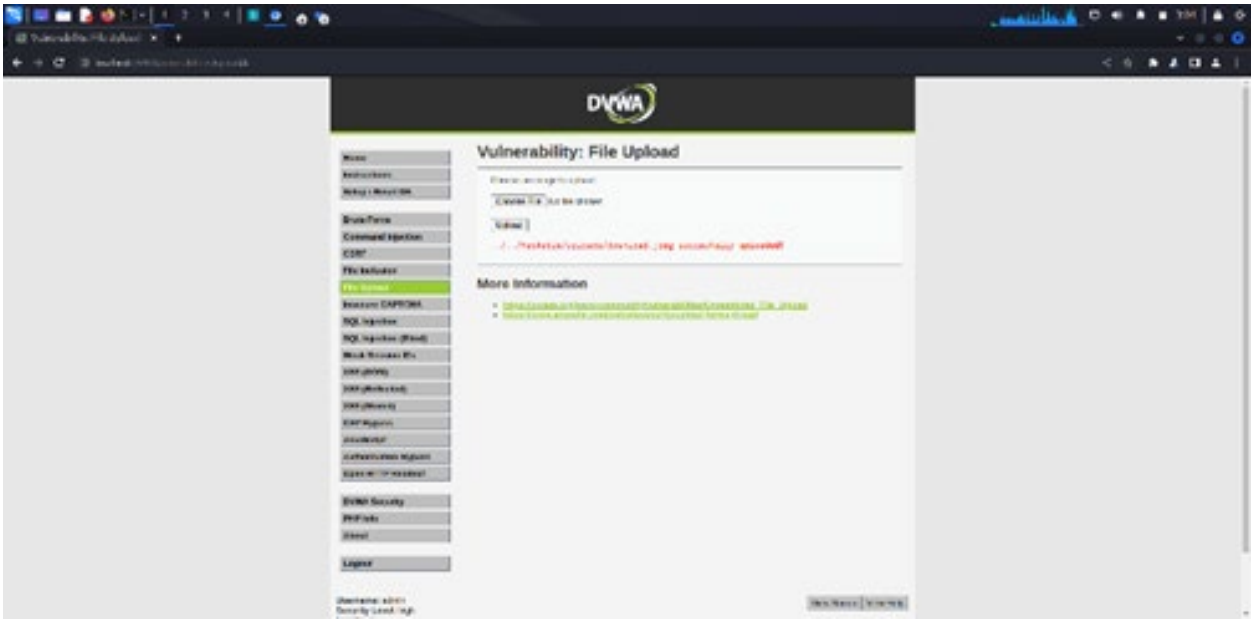




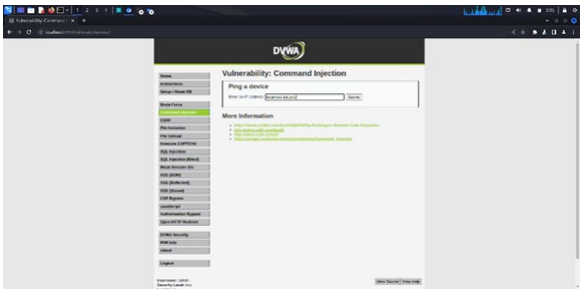
We made sure not to remove the header since the website is trying to resize the image, so we remove the bottom part of the file and put whatever php code we want. Which the server accepts.

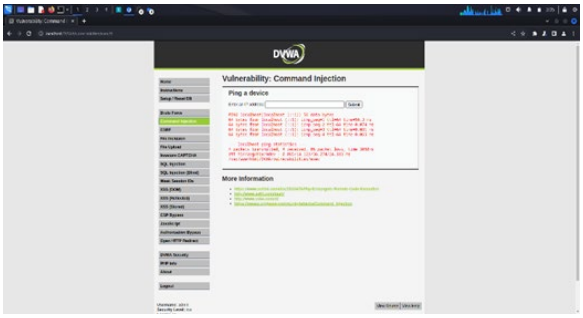


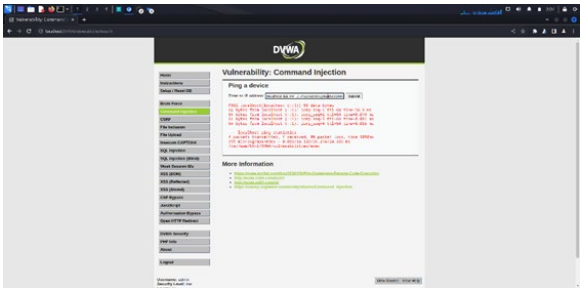


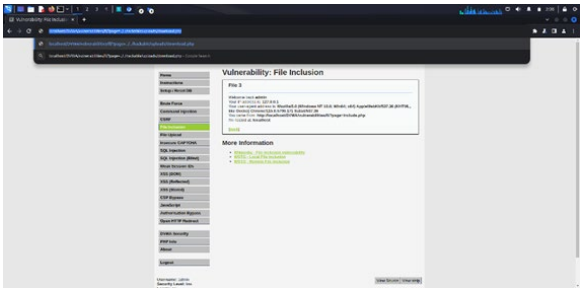


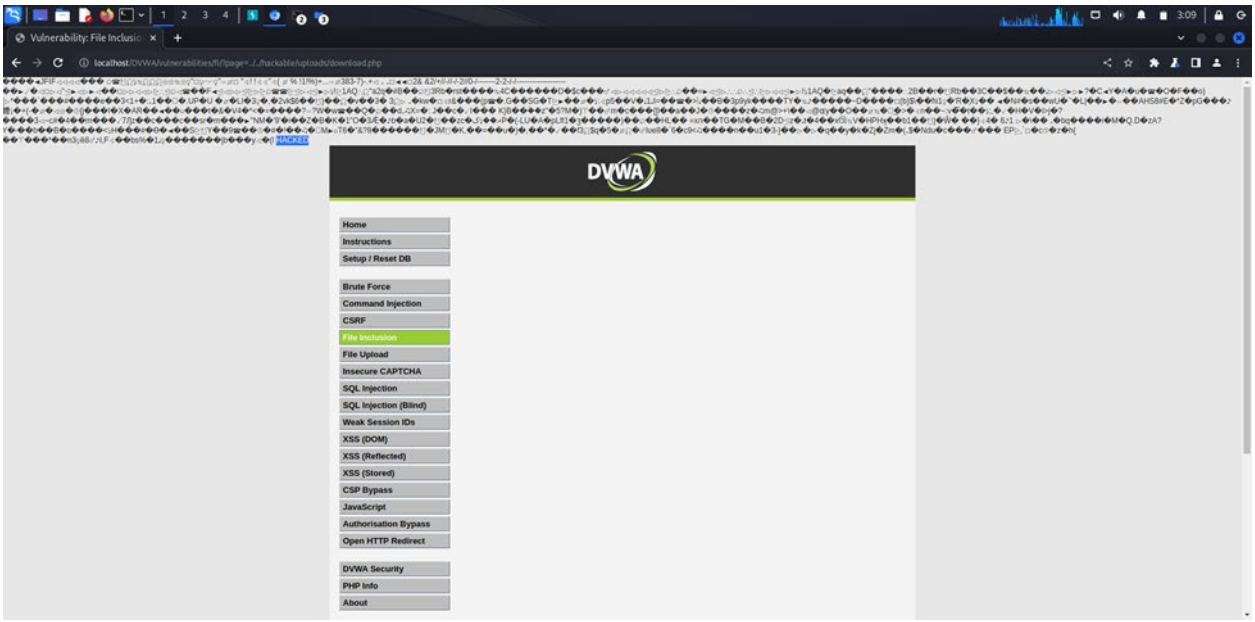
In order to execute it, we changed the extension from .jpeg to .php using the command injection attack, and then used the file inclusion attack to execute the file.



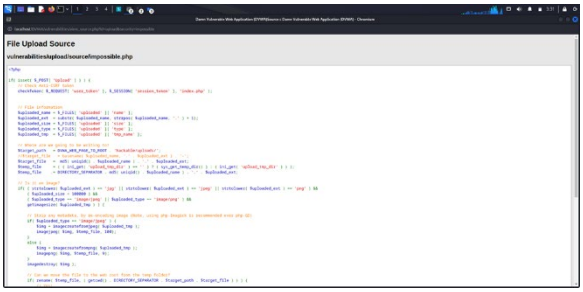


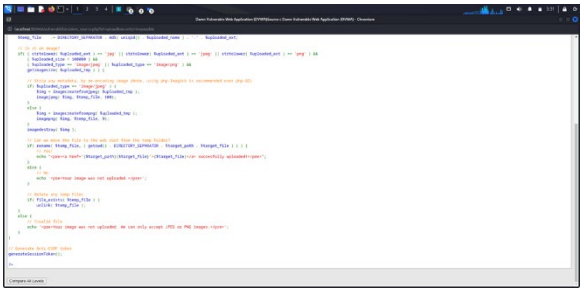






d. Impossible

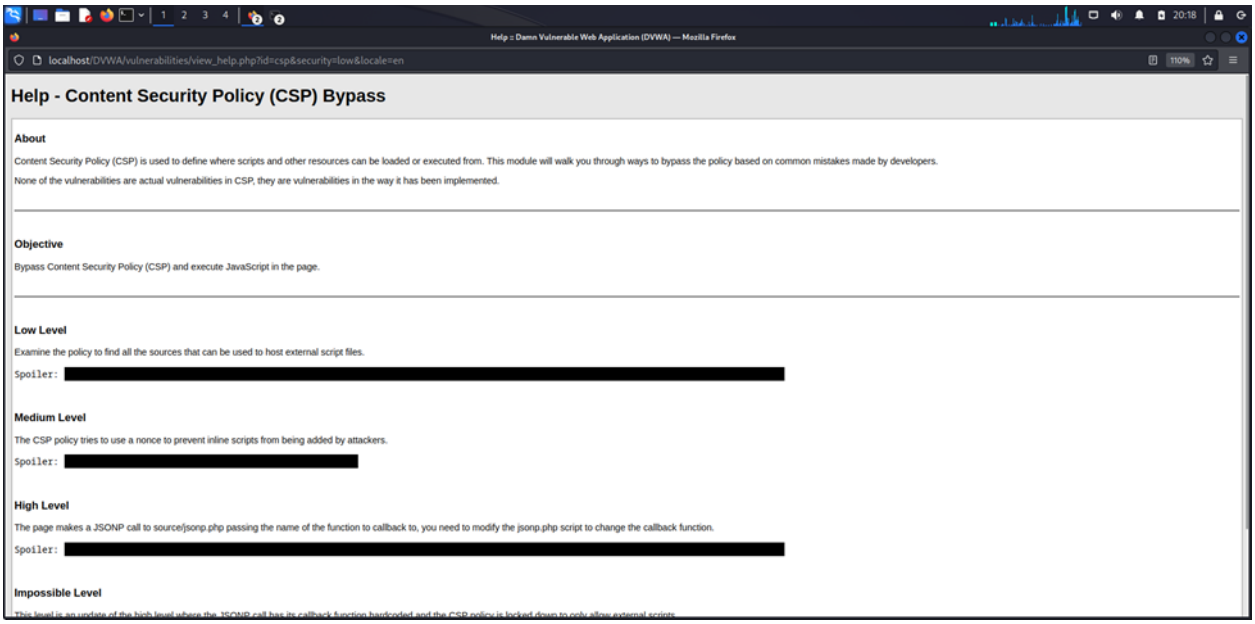




In the impossible level, the developer checks for the type of file being uploaded and makes sure it is an image, it also checks if the file has been altered by if the file size matches the one in the header. The code also indicates the file is being stored in a temporary folder with an altered name.

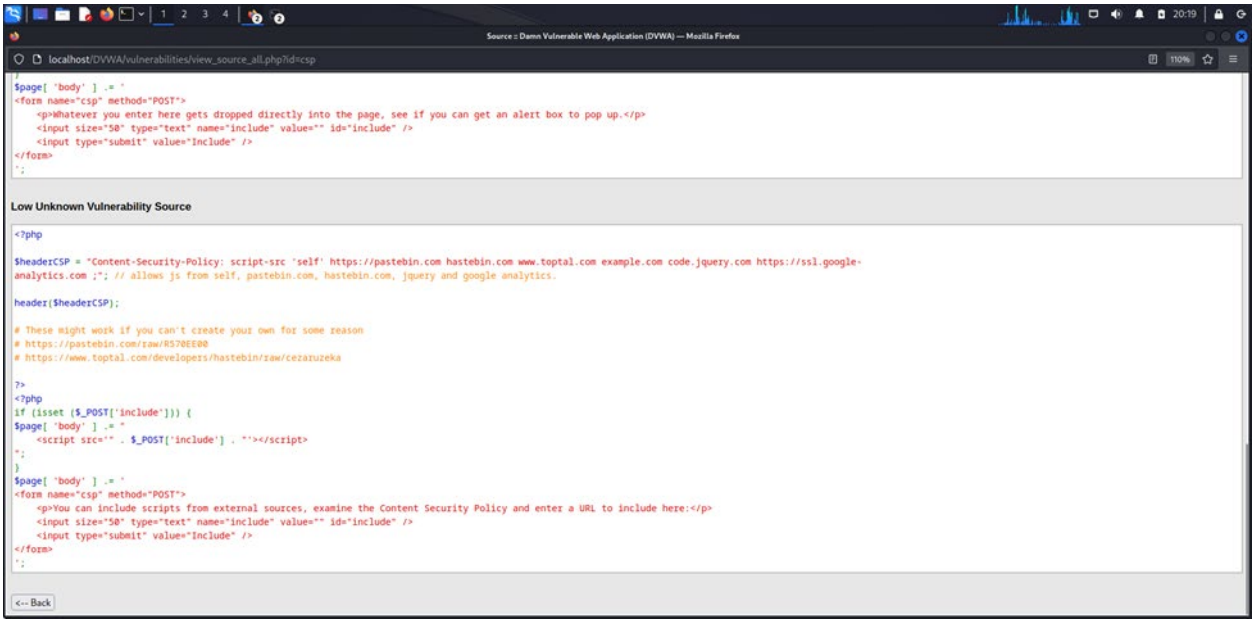
4. CSP Bypass

A Content Security Policy (CSP) is used to limit where the files are loaded from. In this section, our aim is to bypass these policies and execute JavaScript in the page.

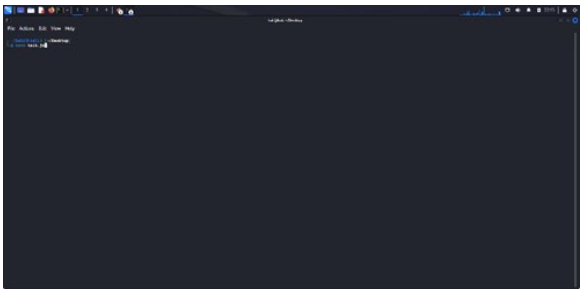


a. Low

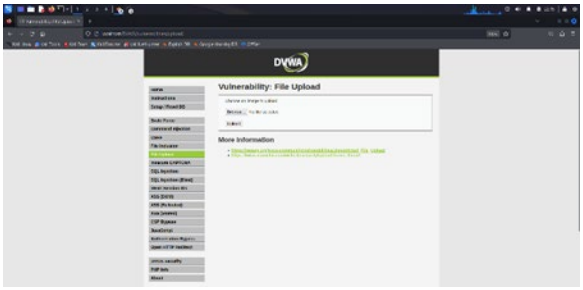
In this level, we can use files from this website, Pastebin, toptal.com, example.com, code.jqery.com, and google analytics.

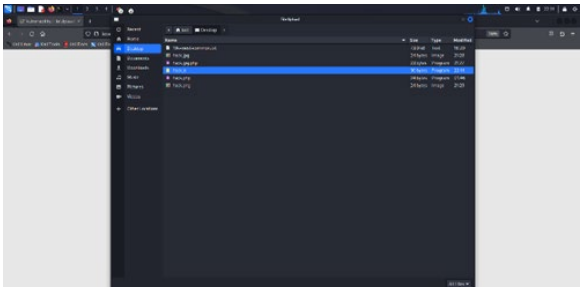


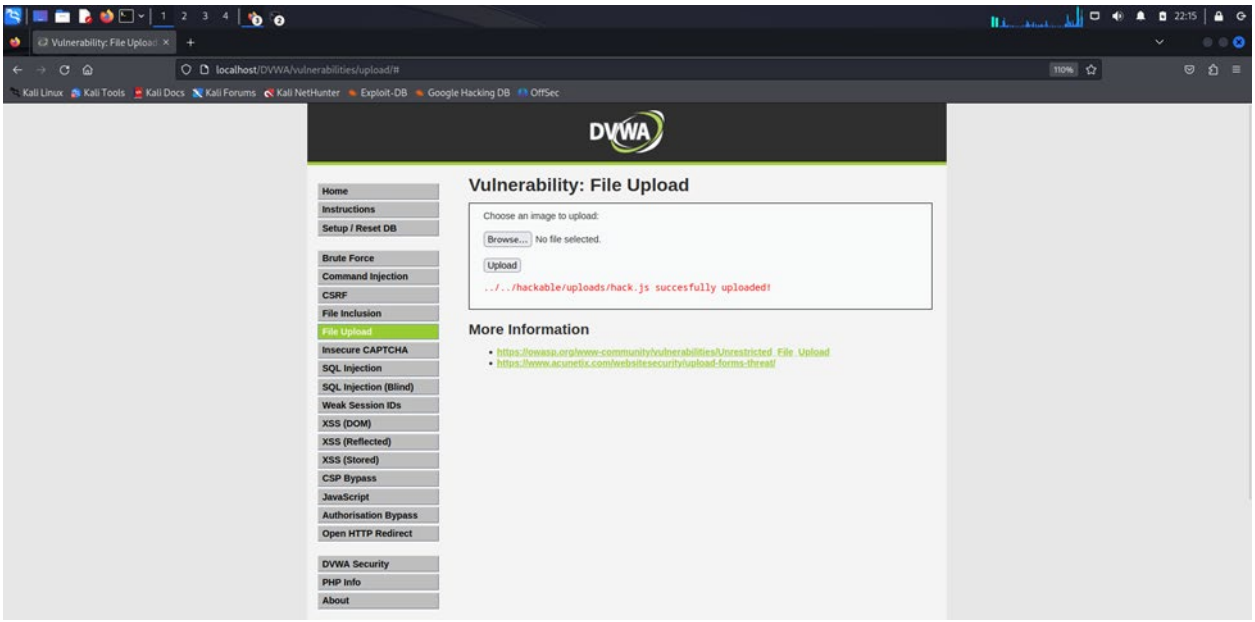
We first create the JavaScript file to be executed.



I then upload the file using the file upload attack.

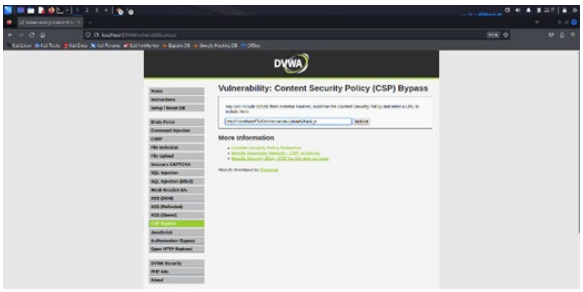


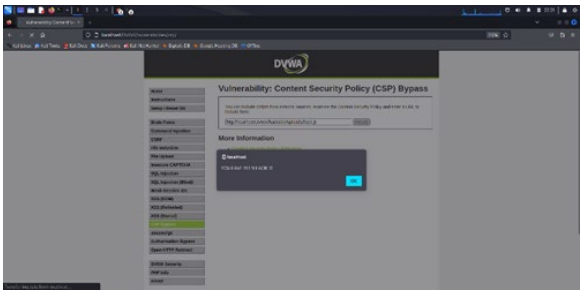




We then send this file to be executed by including its URL

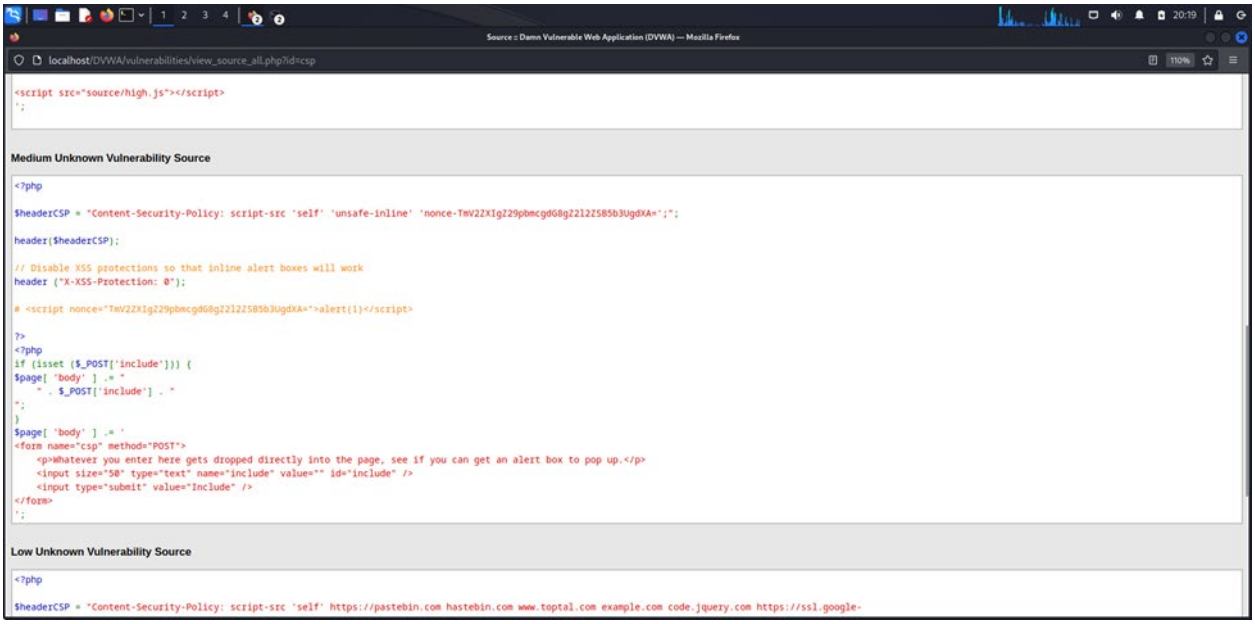
http//:localhost/DVWA/hackable/uploads/hack.js



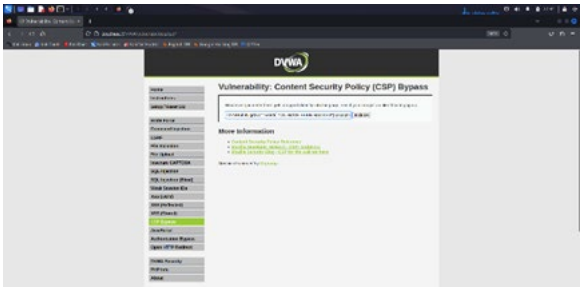


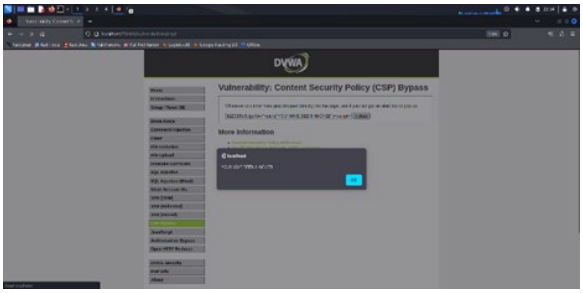
b. Medium

In this level, the CSP only allows scripts with the nonce tag with the value shown below.



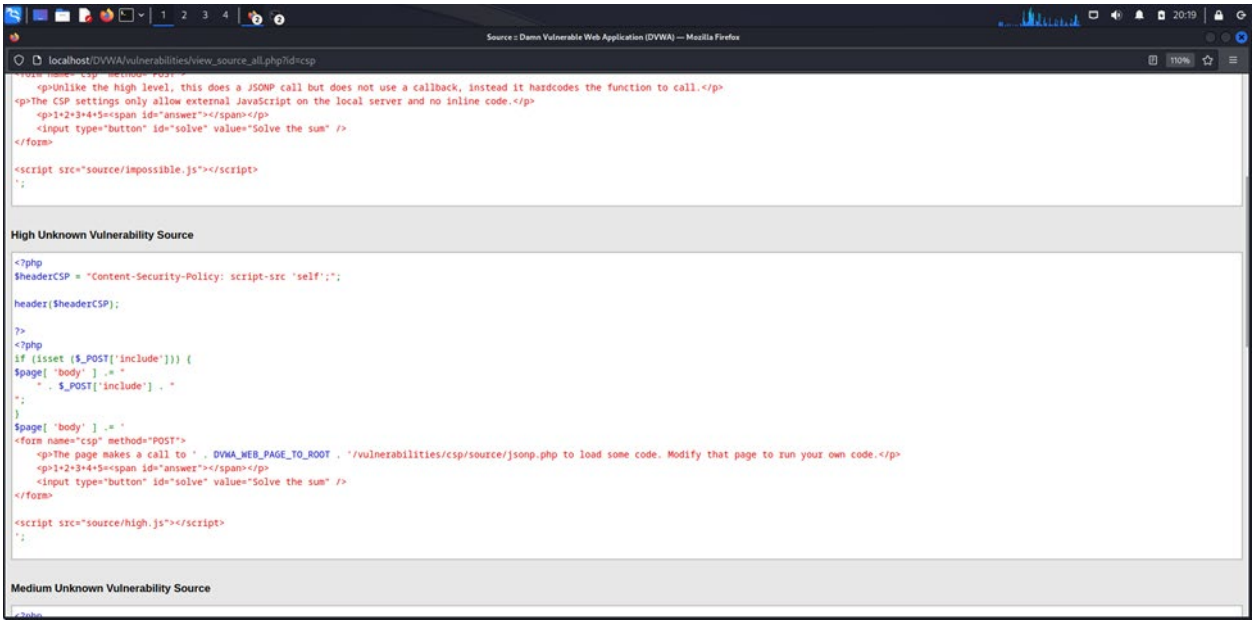
So, in order to run my JavaScript function I can just add my tag to the input box directly with the nonce tag <script nonce="TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA=">alert("YOU HAVE BEEN HACKED")</script>



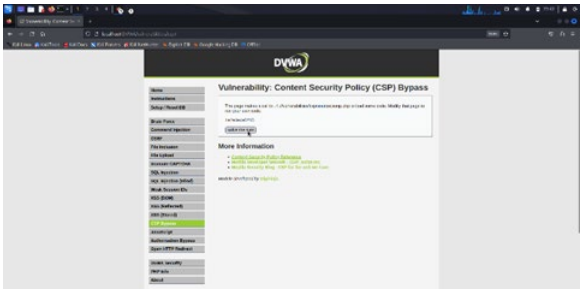


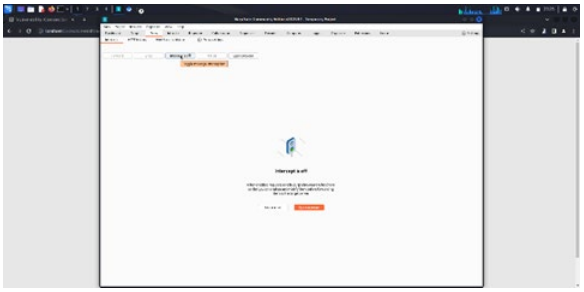
c. High

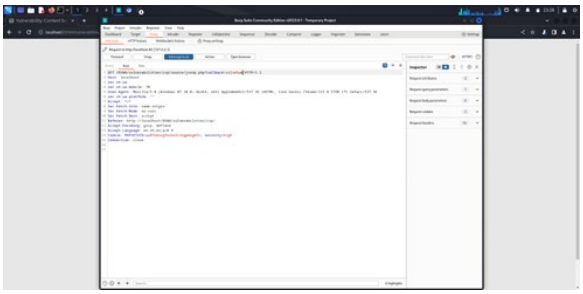
In this level, the developer only allows JavaScript files within this website's environment. Here, a function is returned via callback and being run to display the value of the sum.

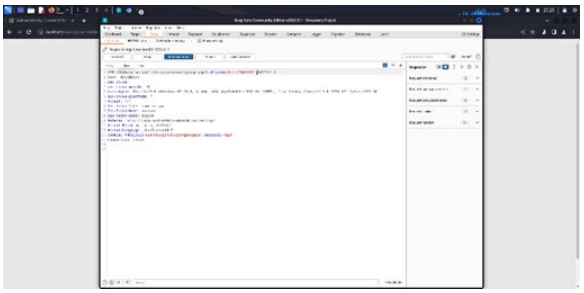


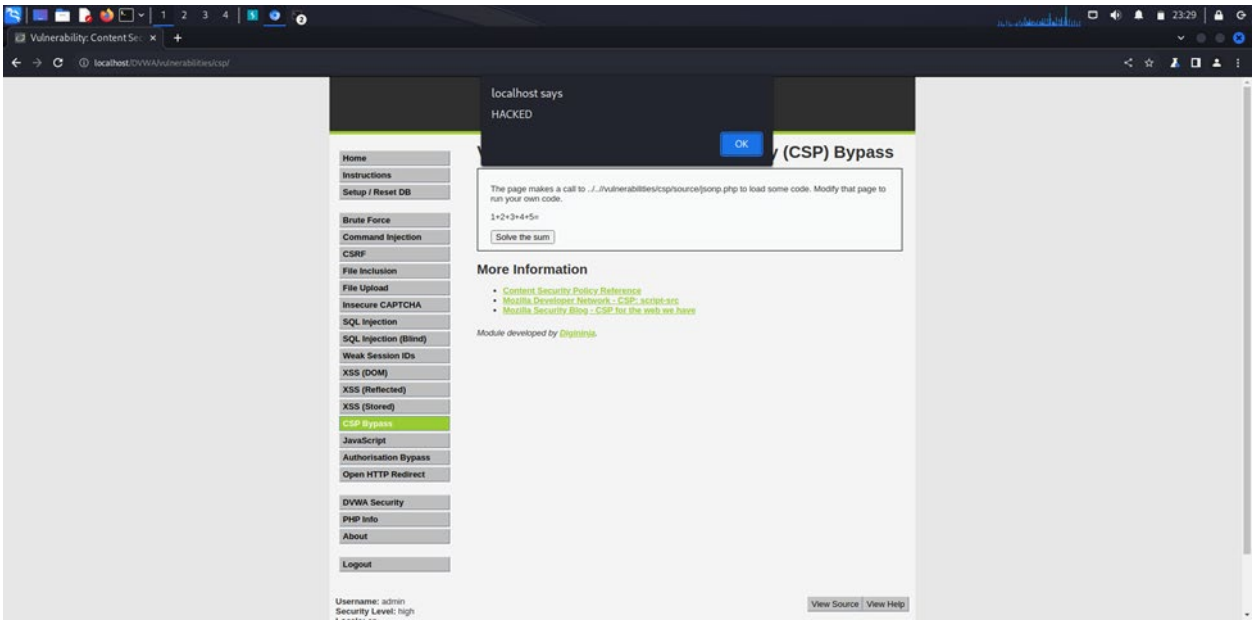
In order to run my own JavaScript code we use burp suit to intercept the request and change the function being called from solve sum, to alert("HACKER").



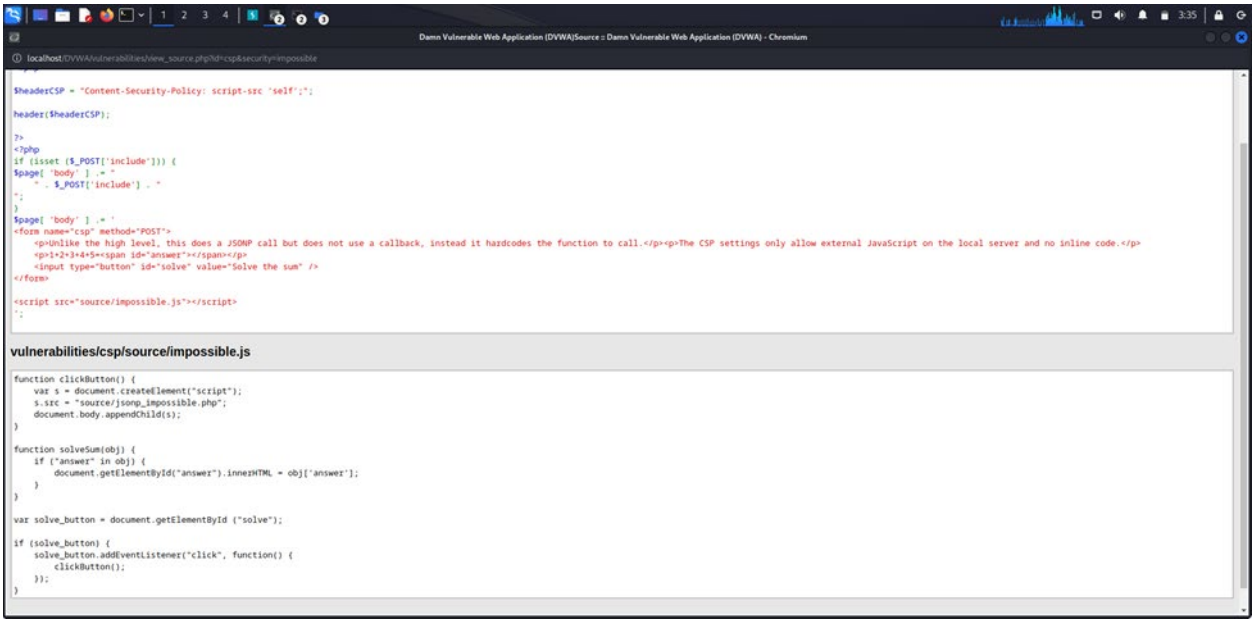








d. Impossible



In this code, the developer does not return the function via callback but by taking the source code directly from the document which does not allow any modification to the function returned.

V. Conclusion

This was generally a very fun lab to work on. It's more hands-on than the previous ones and entailed a more thorough or detailed understanding of the concepts to be able to apply the different techniques and types of attacks.

This was a more demanding lab though, so we had to distribute the work for the sake of time management and meeting the deadline.

Finally, the references list is relatively shorter as we didn't need to research as much this time. DVWA "View Help" and "View Source" interfaces are informative enough, and we only resorted to the spoilers where indicated throughout the report. It also, probably, helped that both of us have completed CSC326 (Operating Systems) and CSC443 (Web Programming) courses, which provided us with enough background information to minimize the need to search different commands and syntax.

VI. References

Kali Linux

https://www.kali.org/

Damn Vulnerable Web Application (DVWA)

https://github.com/digininja/DVWA

https://www.youtube.com/watch?v=PaB17Cc0dUg&t=124s

MariaDB Comment Syntax

https://mariadb.com/kb/en/comment-syntax/

MariaDB Different Releases and Versions

https://mariadb.org/mariadb/all-releases/

MySQLi Real Escape String

https://www.php.net/manual/en/mysqli.real-escape-string.php

Text to ASCII Converter

https://www.browserling.com/tools/text-to-ascii

HTML Special Chars

https://www.php.net/manual/en/function.htmlspecialchars.php

LAU Website

https://www.lau.edu.lb/

PHP Trim

https://www.php.net/manual/en/function.trim.php

Brute Force (High)

https://www.agarri.fr/blog/archives/2020/01/13/intruder_and_csrf-protected_form_without_macros/index.html

Most Common Passwords File

https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10k-most-common.txt

File Upload

https://ethicalhacs.com/dvwa-file-upload/