1. Exploring DVWA : Command Injection Vulnerability Testing

What is Command Injection ?

Command injection is when an attacker can run any commands on a remote system. This usually happens when a web application or server doesn't properly handle user input, giving the attacker the chance to inject and run commands that the system would normally execute.

Steps to Reproduce:

Analyzing Low Security Source Code

Command Injection Source

vulnerabilities/exec/source/low.php

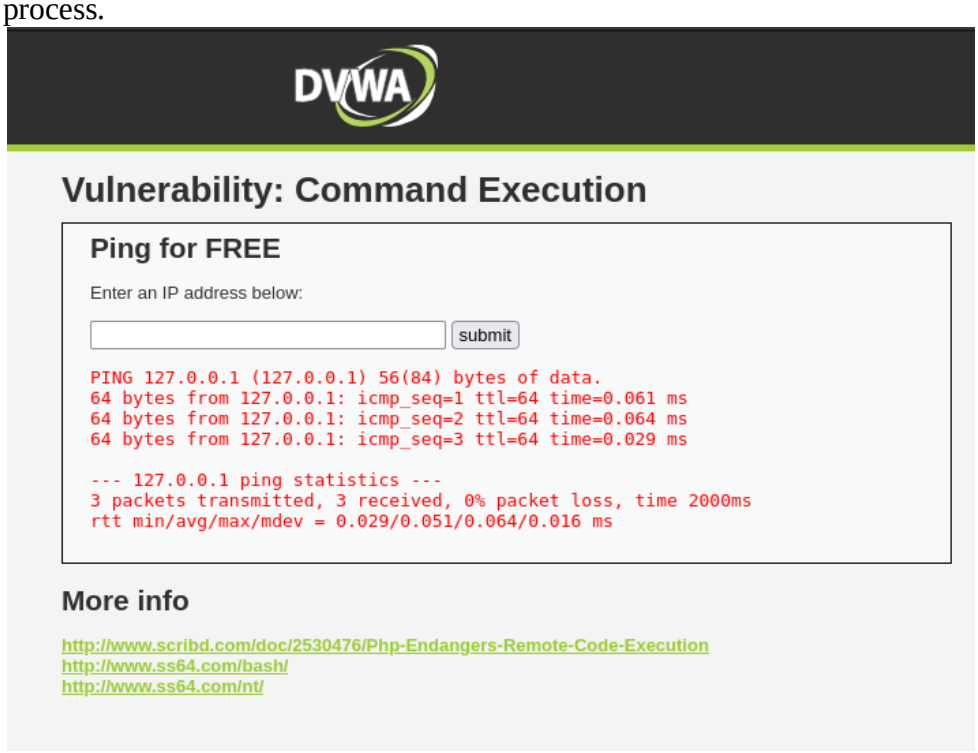
```php
<?php

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

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

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

?>
```

Compare All Levels

Low Security Source Code

The source code shows that the input requires an IP address, triggering a ping command via shell_exec. On Windows, it sends 4 packets, while on other systems, it sends 3 to avoid manual interruption with Ctrl+C.

Here, we input the localhost IP address and trigger the ping command to observe the execution process.



Executing a Simple Ping on the Localhost

Next, let’s modify the command a bit to see what happens. Here’s the updated version :

127.0.0.1 ; cat /etc/passwd

Tampering with the command

The semicolon allows us to run multiple commands in a single line.
Here, the plan is to ping the localhost and then display the content of the /etc/passwd file.

We can also use && in the command :

127.0.0.1 && cat /etc/passwd

Testing Command Injection

What this does is, if the first command works, it will automatically run the second one and show us the contents of the /etc/passwd file.

Mitigation Steps:

- Never directly concatenate user input into shell commands.
- Use **safe APIs** (e.g., execFile() in Node.js or subprocess.run() with arrays in Python).
- Sanitize and strictly validate all inputs (e.g., allow-list approach).
- Run backend processes with **least privileges**.

Why it's dangerous:

- Can lead to full system compromise (e.g., reading sensitive files, adding users, or taking control of the server).
- Often provides **remote code execution (RCE)**.
- Attackers can pivot into internal networks or escalate privileges.

Impact Level: **High**

2. Exploring DVWA : SQL injection Vulnerability Testing

What is SQL Injection ?

SQL injection is a technique used to manipulate SQL queries, allowing attackers to access, modify, or delete data in a database by exploiting vulnerable input fields .

Steps to Reproduce:

Analyzing Low Security Source Code

SQL Injection Source

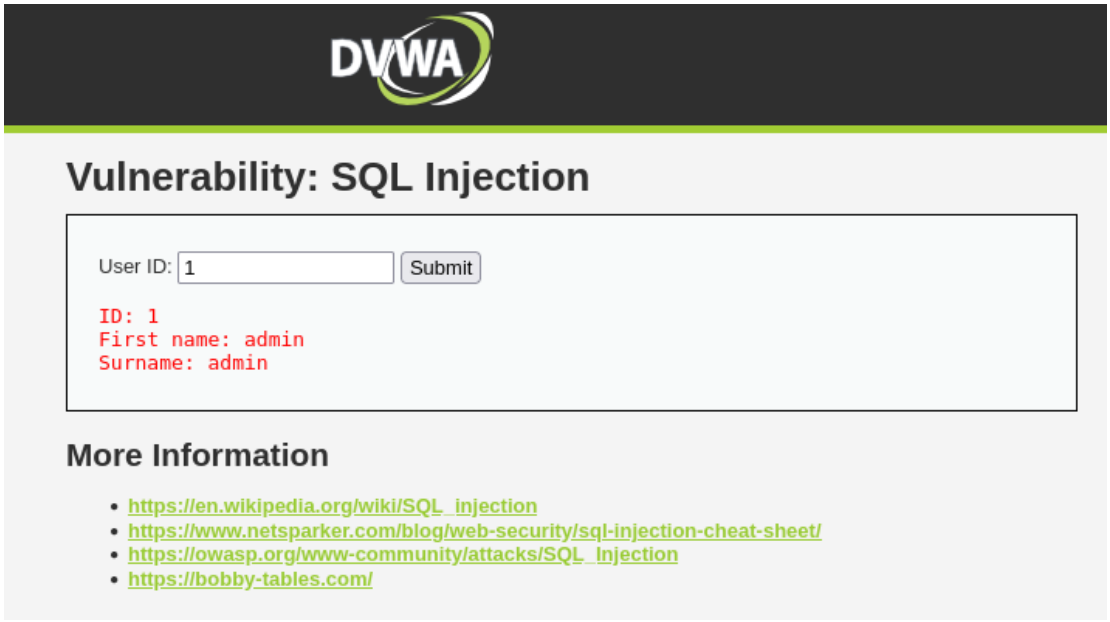
vulnerabilities/sqli/source/low.php

```php
<?php

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

    switch ($_DVWA['SQLI_DB']) {
        case MYSQL:
            // Check database
            $query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
            $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die( '<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? mysqli_error($GLOBALS["___mysqli_ston"]) : (($_
pre>' );

            // Get results
            while( $row = mysqli_fetch_assoc( $result ) ) {
                // Get values
                $first = $row["first_name"];
                $last  = $row["last_name"];

                // Feedback for end user
                echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
            }

            mysqli_close($GLOBALS["___mysqli_ston"]);
            break;
        case SQLITE:
            global $sqlite_db_connection;

            #$sqlite_db_connection = new SQLite3($_DVWA['SQLITE_DB']);
            #$sqlite_db_connection->enableExceptions(true);

            $query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
            #print $query;
            try {
                $results = $sqlite_db_connection->query($query);
            } catch (Exception $e) {
                echo 'Caught exception: ' . $e->getMessage();
                exit();
            }
```

Low Security Source Code

The code takes the ID submitted through the input field and searches the database for the corresponding user. It retrieves the user's first name and last name, then displays them on the web interface.

For example, if we enter the ID 1, the code fetches the first name admin and last name admin, which are associated with that ID.



Testing the Application

Because the code directly uses the ID submitted by the user in the SQL query without checking it properly, it becomes vulnerable to SQL injection .

So, our first attempt will be to enter this payload :

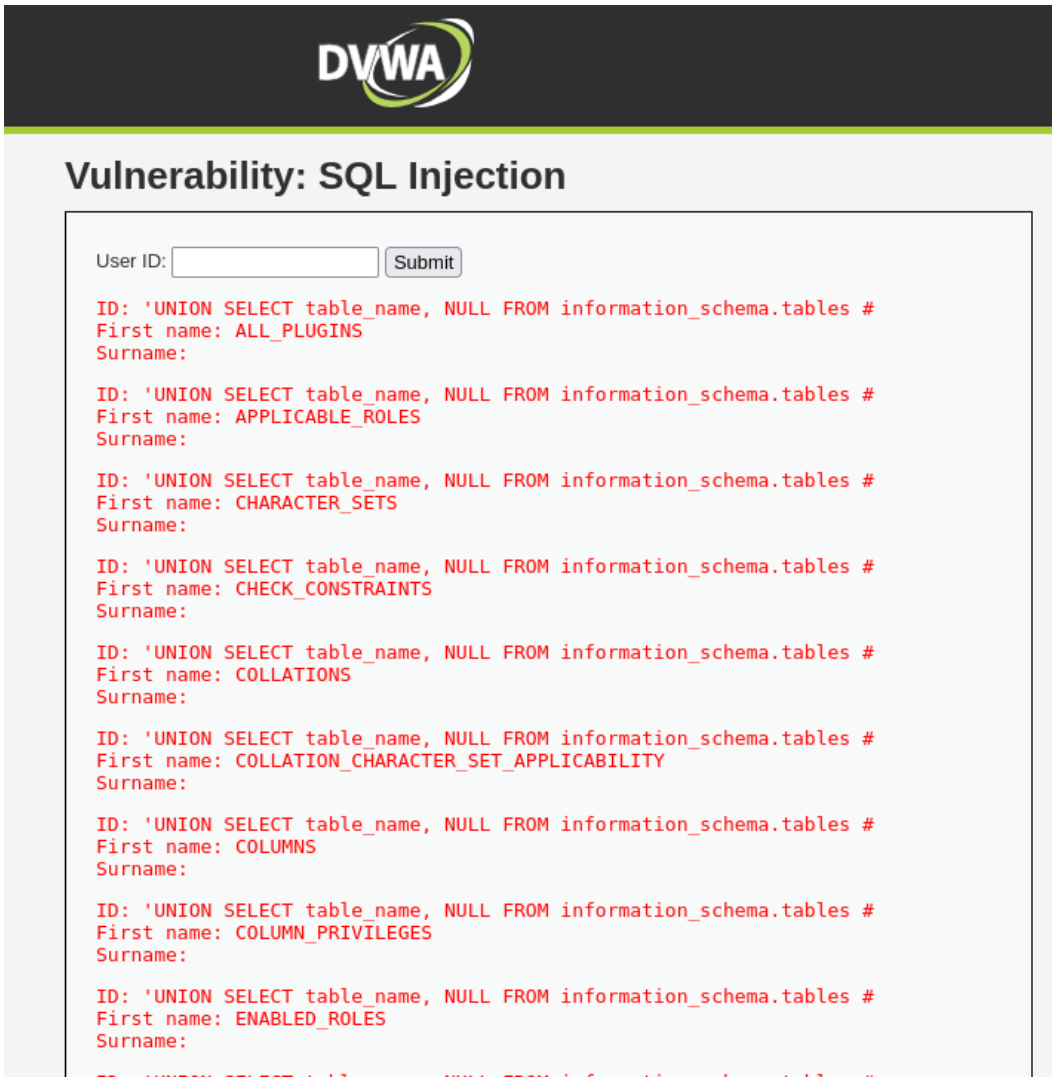
1' OR '1'='1'#

First SQL Injection Payload Attempt

This will alter the SQL query, letting us skip the need for a specific ID and instead display all users in the database, as the condition ‘1’=’1' is always true .

Our second attempt will be to display the existing tables within the database using the payload :

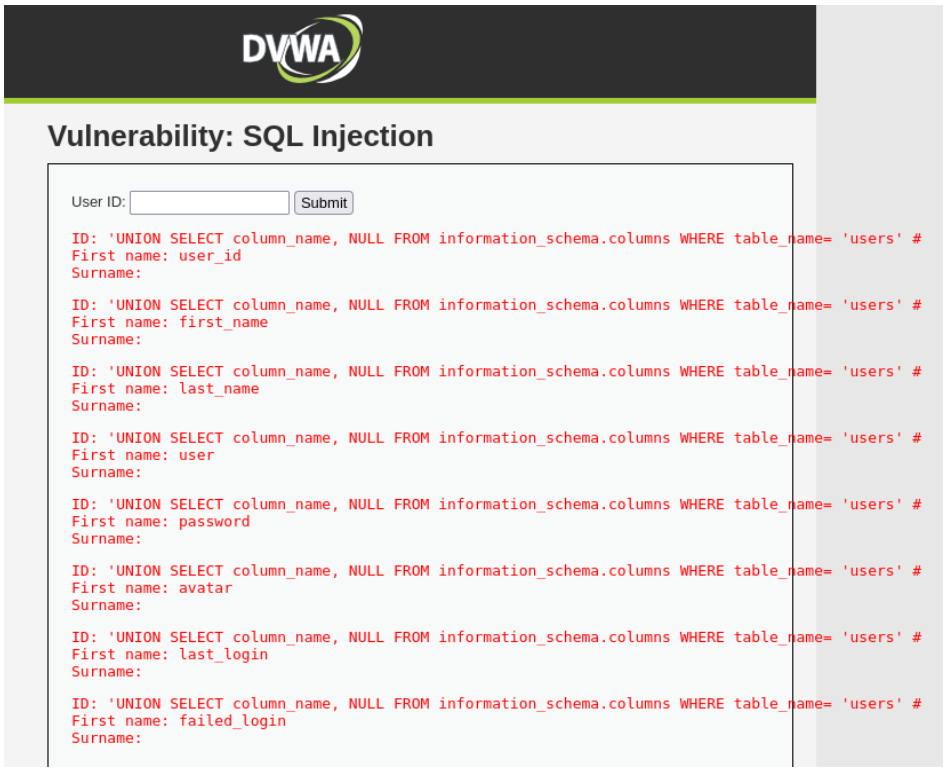
'UNION SELECT table_name, NULL FROM information_schema.tables #



Listing Tables Using SQL Injection

Next, we'll retrieve and display the columns of the users table by using the query :

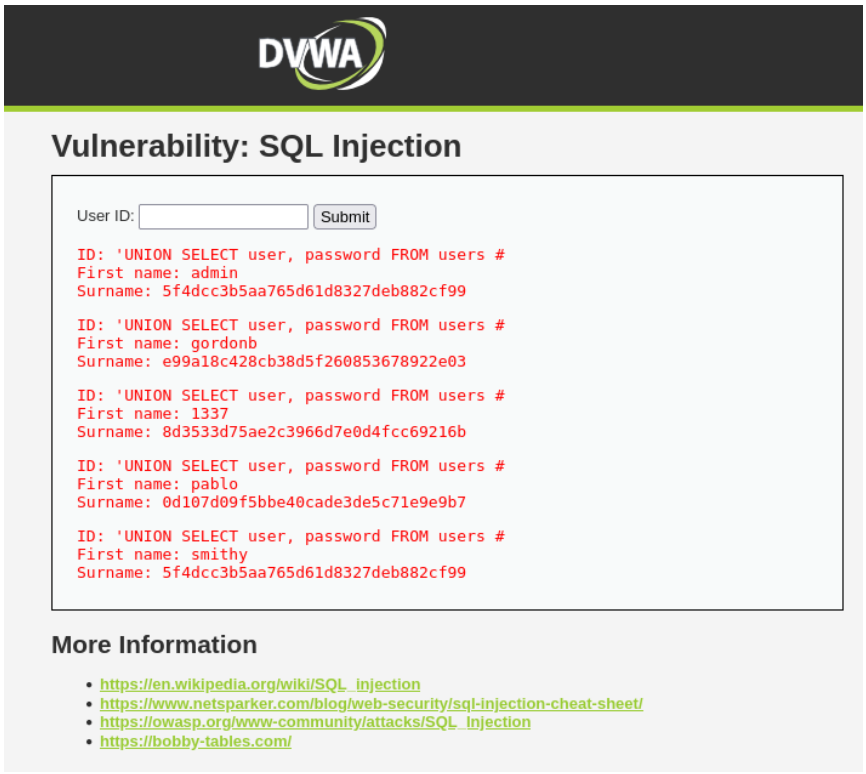
'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' #



Listing Columns in the Users Table Using SQL Injection

We can also access the usernames along with their encrypted passwords .

'UNION SELECT user, password FROM users #



Viewing Users Encrypted Passwords with SQL Injection

Analyzing Medium Security Source Code

SQL Injection Source

vulnerabilities/sqli/source/medium.php

```php
<?php

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

    $id = mysqli_real_escape_string($GLOBALS["___mysqli_ston"], $id);

    switch ($_DVWA['SQLI_DB']) {
        case MYSQL:
            $query  = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
            $result = mysqli_query($GLOBALS["___mysqli_ston"], $query) or die( '<pre>' . mysqli_error($GLOBALS["___mysqli_ston"]) . '</pre>' );

            // Get results
            while( $row = mysqli_fetch_assoc( $result ) ) {
                // Display values
                $first = $row["first_name"];
                $last  = $row["last_name"];

                // Feedback for end user
                echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
            }
            break;
        case SQLITE:
            global $sqlite_db_connection;

            $query  = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
            #print $query;
            try {
                $results = $sqlite_db_connection->query($query);
            } catch (Exception $e) {
                echo 'Caught exception: ' . $e->getMessage();
                exit();
            }

            if ($results) {
                while ($row = $results->fetchArray()) {
                    // Get values
                    $first = $row["first_name"];
```

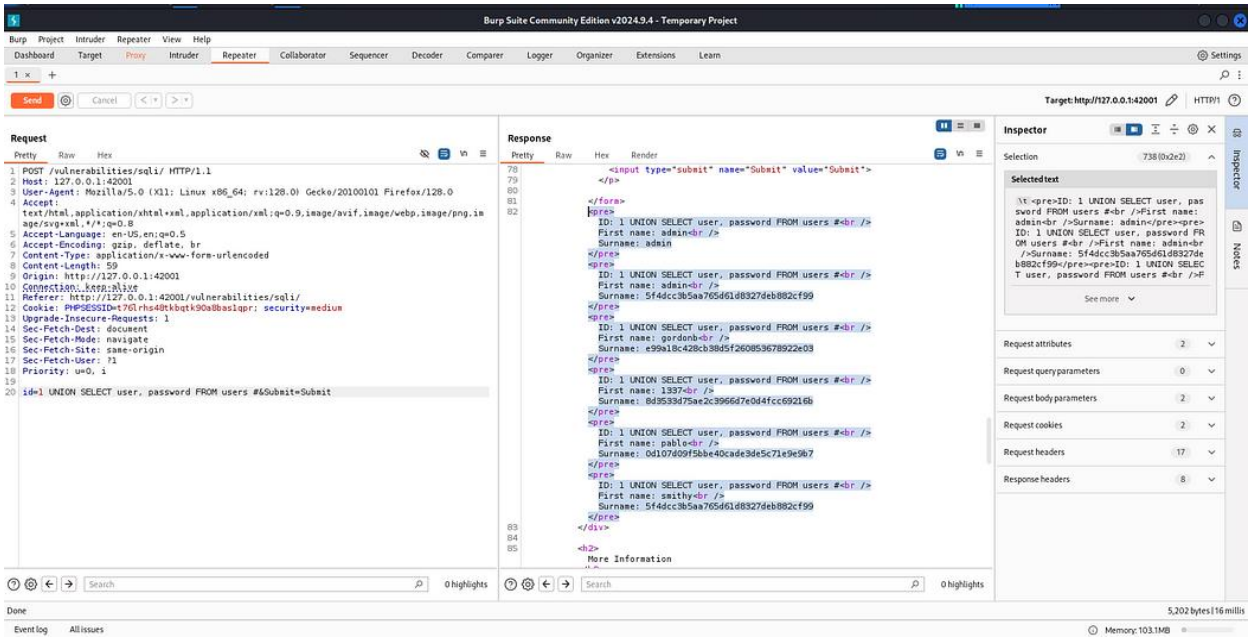
After analyzing the code, I noticed the addition of the mysqli_real_escape_string function to the ID input, which escapes special characters and helps protect against SQL injection .

Additionally, the ID input has been changed to a checkbox, making the input handling more efficient .

Testing the Application on Medium Security Level

If we can’t inject the input directly through the form, we can use Burp Suite to do it .

By intercepting the request, we can modify the input parameter id=1 to 1 UNION SELECT user, password FROM users #, send the request, and then view the results in the response .



Injecting SQL Payload Through Burp Suite

Mitigation Steps:

- Use **parameterized queries** (e.g., prepared statements).
- Sanitize and validate all user inputs.
- Use **ORMs** (e.g., Sequelize, Hibernate) which abstract raw queries.
- Enforce **least privilege** access to databases.
- Implement a **WAF** (Web Application Firewall) to block malicious payloads.

Why it's dangerous:

- Can expose, alter, or delete **entire databases**.
- Attackers can **bypass authentication**, retrieve passwords, and more.
- If combined with other flaws, may lead to full server compromise.

Impact Level: **High**

3. Exploring DVWA : CSRF Vulnerability Testing

What is CSRF ?

CSRF is when an attacker tricks a victim into performing actions on a website they're logged into, like changing their password or making a purchase, without their knowledge.

Steps to Reproduce:

Analyzing Low Security Source Code

CSRF Source

vulnerabilities/csrf/source/low.php

```php
<?php

if( isset( $_GET[ 'Change' ] ) ) {
    // Get input
    $pass_new  = $_GET[ 'password_new' ];
    $pass_conf = $_GET[ 'password_conf' ];

    // Do the passwords match?
    if( $pass_new == $pass_conf ) {
        // They do!
        $pass_new = ((isset($GLOBALS["___mysqli_ston"]) && is_object($GLOBALS["___mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___mysqli_ston"],  $pass_new ) : ((trigger_error("
[MySQLConverterToo] Fix the mysql_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
        $pass_new = md5( $pass_new );

        // Update the database
        $insert = "UPDATE `users` SET password = '$pass_new' WHERE user = '" . dvwaCurrentUser() . "';";
        $result = mysqli_query($GLOBALS["___mysqli_ston"],  $insert ) or die( '<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? mysqli_error($GLOBALS["___mysqli_ston"]) : (($___mysqli_res = mysqli_connect_err

        // Feedback for the user
        echo "<pre>Password Changed.</pre>";
    }
    else {
        // Issue with passwords matching
        echo "<pre>Passwords did not match.</pre>";
    }

    ((is_null($___mysqli_res = mysqli_close($GLOBALS["___mysqli_ston"]))) ? false : $___mysqli_res);
}

?>
```

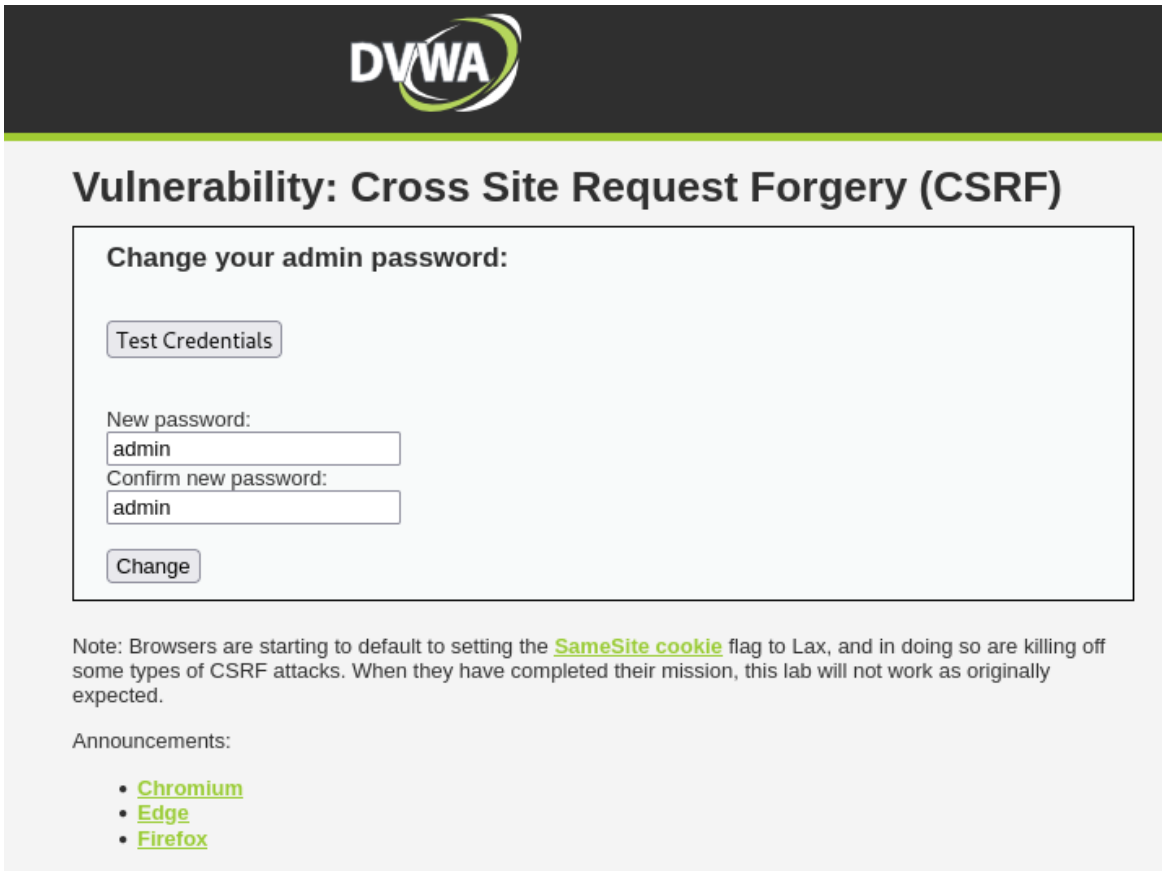
Compare All Levels

Low Security Source Code

While reviewing the code, I noticed that the New Password and Confirm New Password fields are processed using the GET method. If the passwords match, the new password is hashed with MD5, updated in the database, and a Password Changed message appears. If they don't match, a message saying Passwords did not match is displayed instead.

The __mysqli_ston is just an internal variable used by PHP to keep track of which database connection to use, especially when there are multiple connections involved.

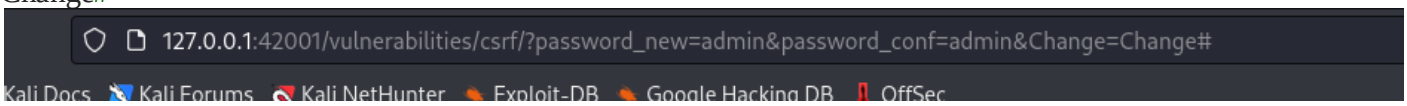
We will enter admin as the new password in both inputs and click the Change button to update it.



Setting a New Admin Password

After clicking the Change button, we can see that the request is handled using the GET method, as we saw earlier in the source code, with the parameters included directly in the URL .

http://127.0.0.1:42001/vulnerabilities/csrf/?password_new=admin&password_conf=admin&Change=Change#

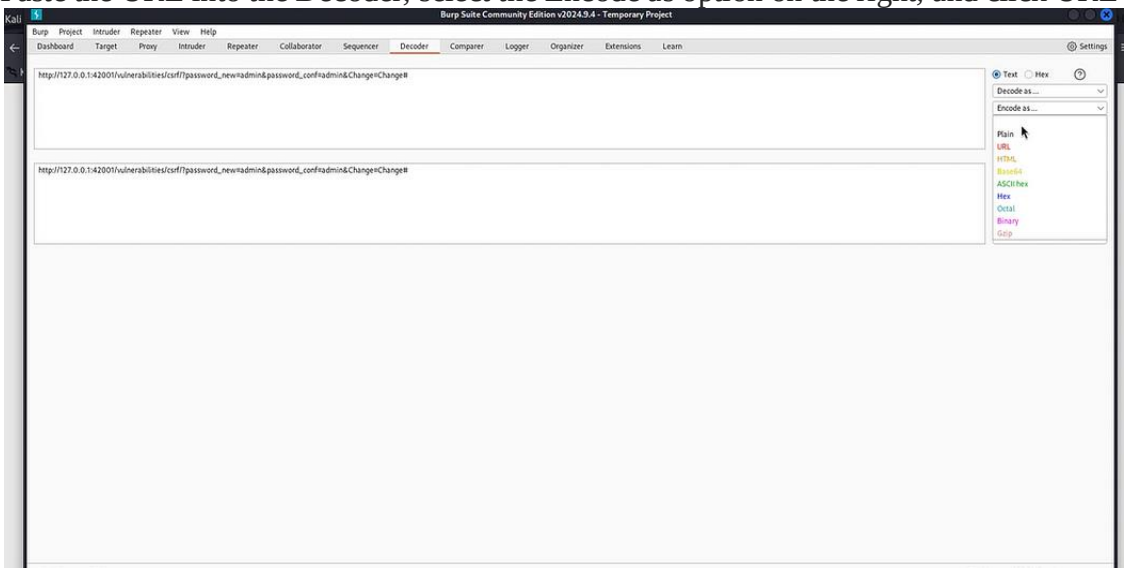


Password Modification URL

This lets us craft a URL with the parameters and send it to the victim. When they click on it, their password will be changed automatically, as long as they're logged in .

We can use Burp Suite's decoder option to further encode the URL, making it harder for the victim to recognize its true purpose.

Paste the URL into the Decoder, select the Encode as option on the right, and click URL .



Encoding URL with Burp Suite

It should look like this

%68%74%74%70%3a%2f%2f%31%32%37%2e%30%2e%30%2e%31%3a%34%32%30%30%31%2f%76%75%6c%6e%65%72%61%62%69%6c%69%74%69%65%73%2f%63%73%72%66%2f%3f%70%61%73%73%77%6f%72%64%5f%6e%65%77%3d%61%64%6d%69%6e%26%70%61%73%73%77%6f%72%64%5f%63%6f%6e%66%3d%61%64%6d%69%6e%26%43%68%61%6e%67%65%3d%43%68%61%6e%67%65%23

Mitigation Steps:

- Use **anti-CSRF tokens** (synchronizer token pattern or double-submit cookies).
- Set **SameSite=Lax or Strict** on cookies.
- Verify the **Origin or Referer** headers on sensitive requests.
- Use **JavaScript frameworks** that enforce CSRF protections (e.g., Django, Laravel, Rails).

Why it's dangerous:

- Exploits the **trust a web app has in the user**.
- Actions are performed **without the user's consent**.
- Can lead to unauthorized actions like changing account email, making purchases, etc.

Impact Level: **Medium**

- **Why?**: Attacker can trick users into changing account email/password.

4. Document Object Model (DOM) Cross Site Scripting (XSS) Vulnerability Testing

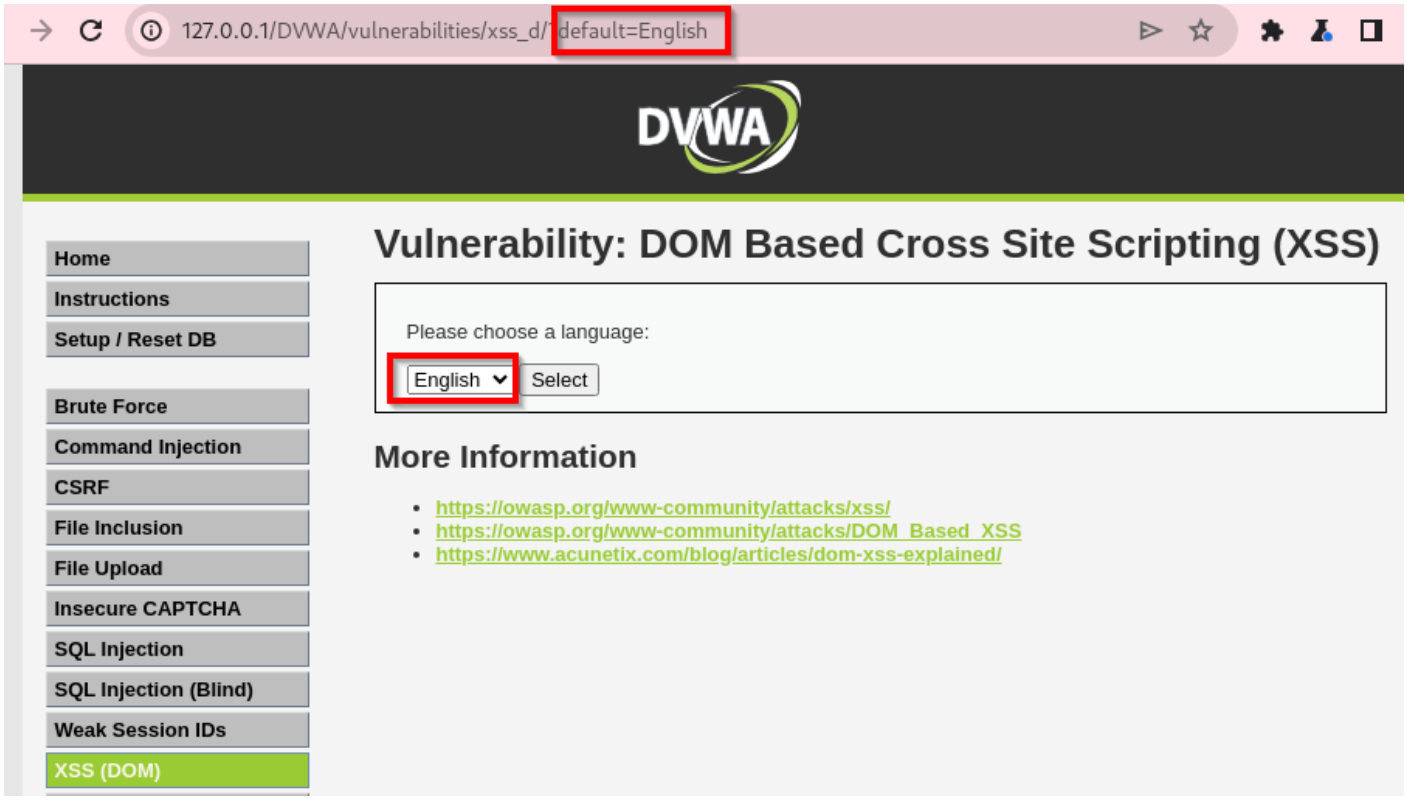
What is DOM XSS?

DOM-Based XSS is a type of Cross-Site Scripting vulnerability that occurs **entirely on the client side** (in the browser), rather than in the server's response. It happens when JavaScript on the web page **reads data from an untrusted source (like the URL or user input)** and writes it back into the page **without proper sanitization or encoding**.

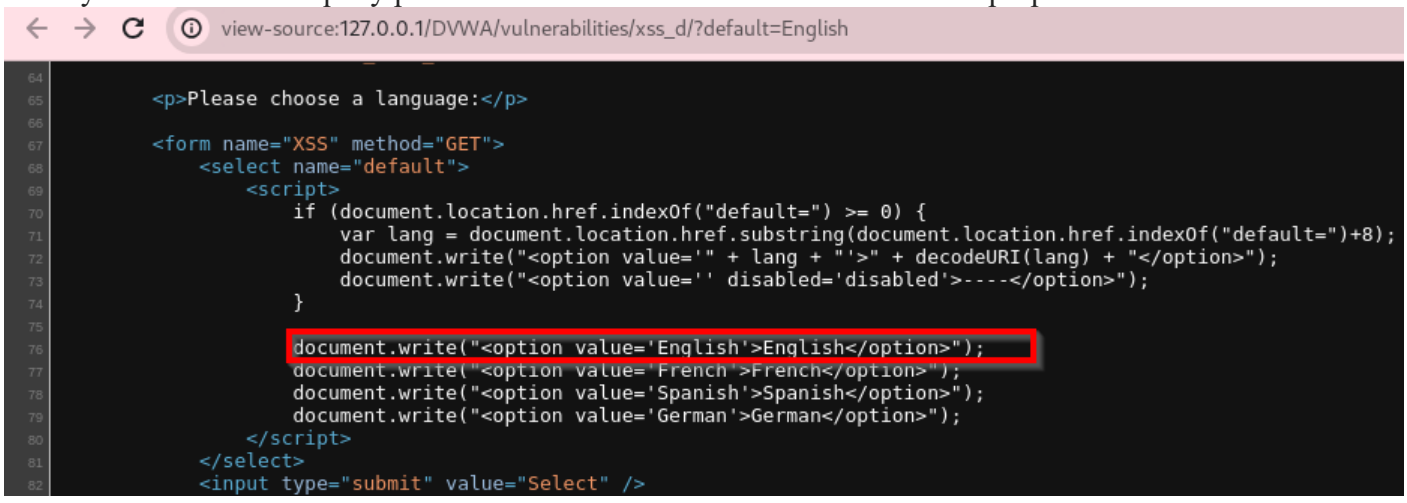
In DOM XSS, the **vulnerable code is in the JavaScript** executed by the browser, not the HTML returned by the server.

Steps to Reproduce:

Go to DOM XSS challenge
We can notice there is no input field and application is asking to select **Language** from dropdown
Lets choose any language and click on the **Select** button.
Selected language appeared in the URL parameter as **default=English.**



In DOM-based XSS, the vulnerability is caused by the client-side JavaScript code, which uses unsafe values from the URL or other DOM elements. In this case, vulnerable script reads input directly from the URL's query parameter and inserts it into the DOM without proper sanitization.



Change the URL parameter to a malicious payload such as default=**<script>alert('DOM XSS')</script>** and click on enter button.

Challenge Solved.

Mitigation Steps:

- Avoid using innerHTML, document.write, or other unsafe sinks.
- Use **context-aware output encoding** (e.g., escaping HTML, JS, or URL components).
- Apply client-side sanitization libraries like **DOMPurify**.
- Implement a strict **Content Security Policy (CSP)** header.

Why it's dangerous:

- Bypasses traditional server-side filters.
- Can lead to **theft of cookies**, **session hijacking**, and **phishing**.
- Happens on the client-side, making it **harder to detect**.

Impact Level: **Medium**

- **Why?**: If exploitable, attacker can hijack sessions of authenticated users or inject malicious JavaScript.

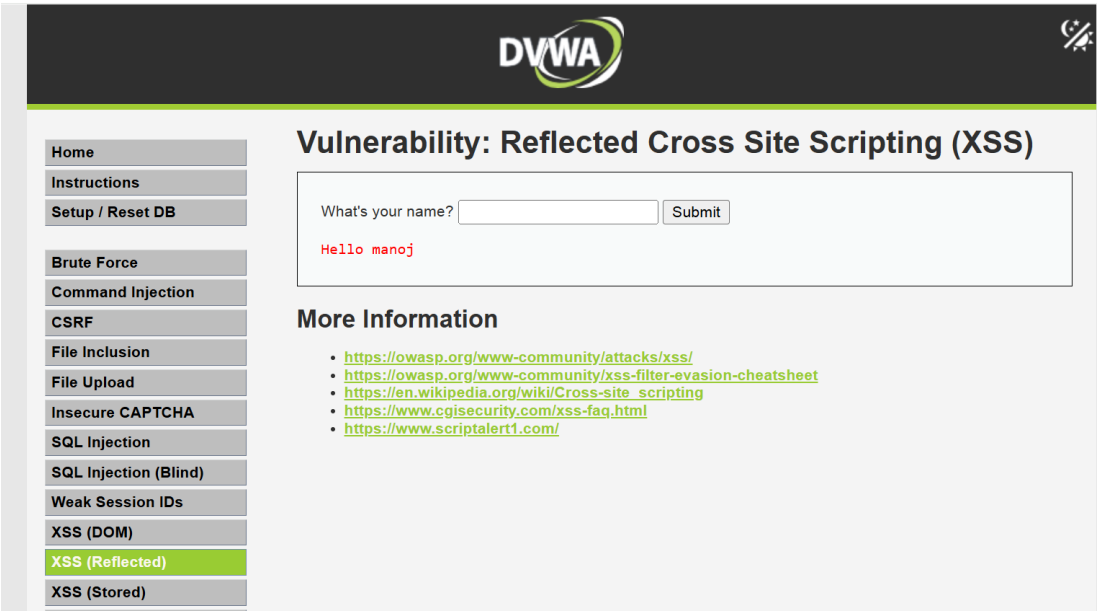
5. Reflected Cross Site Scripting (XSS)

Description:

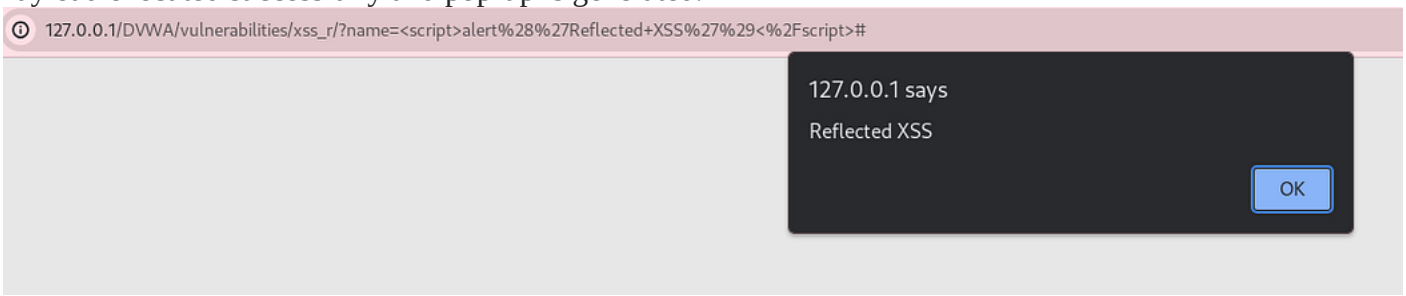
Reflected XSS occurs when user-supplied input (e.g., via a URL parameter, form, or query string) is immediately **reflected back in the server's response** without proper validation or encoding. The malicious script is executed in the victim's browser when they click on a specially crafted link.

Steps to Reproduce:

Login to DVWA application and go to **Reflected Cross Site Scripting (XSS)** challenge. Provide any input and notice that provided input reflected in the same page.



Now try XSS payload in the input field — **<script>alert('Reflected XSS')</script>**
Payload executed successfully and pop-up is generated.



We can use the generated URL to exploit this vulnerability by sharing it to victim user and the vulnerable URL for this scenario is -
http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=%3Cscript%3Ealert%28%27Reflected+XSS%27%29%3C%2Fscript%3E#
Open the URL in new tab and it is possible to exploit Reflected XSS
Challenge Solved.

Mitigation Techniques:

- **Input validation & sanitization**
  - Disallow or escape special characters like <, >, ", ', etc.
- **Output encoding**

    - Use context-aware output encoding (e.g., HTML, JavaScript, URL).
- **Use CSP (Content Security Policy)**
    - Restrict sources of scripts with headers like `Content-Security-Policy`.
- **Avoid reflecting user input in the HTML response**
    - Minimize direct usage of user input in page output.
- **Use security libraries/frameworks**
    - For example, React, Angular automatically handle much of XSS prevention.

Why it's dangerous:

- Can steal session tokens, cookies, or login credentials.
- Often used in phishing attacks.
- Requires user interaction, but can be very effective via crafted links.

Impact Level:

High (if session cookies or sensitive data are stolen)

6. Bruteforce Web Login

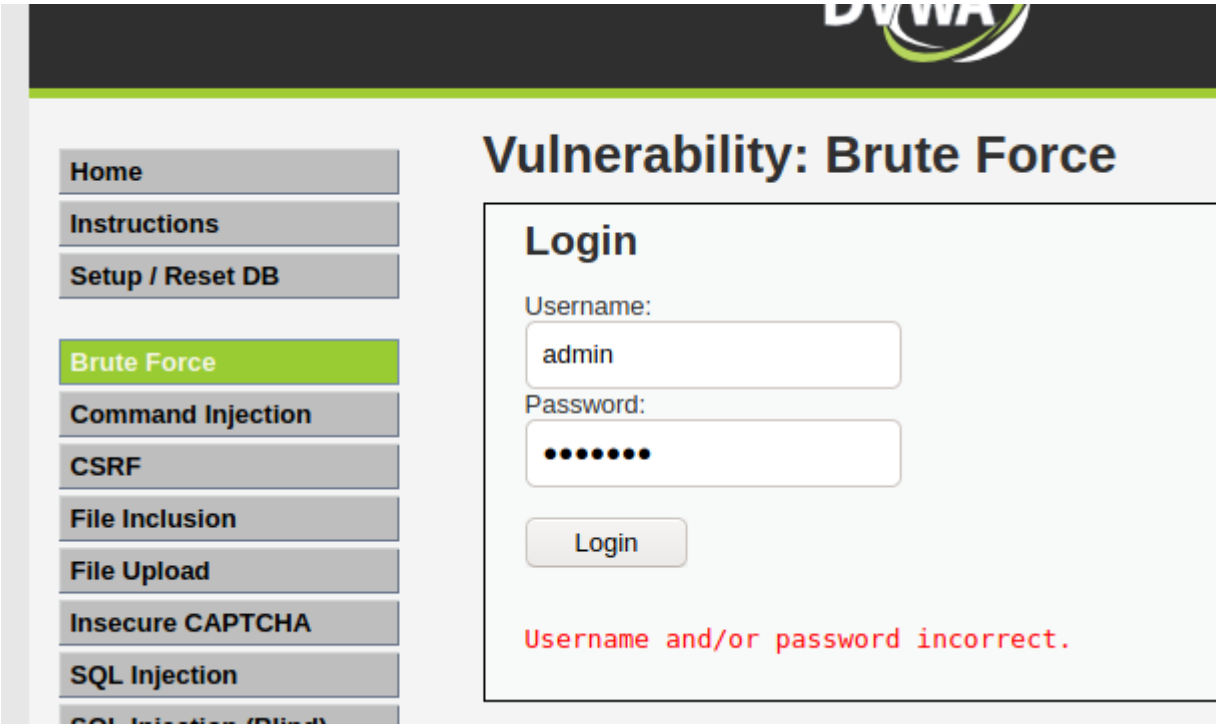
Description:

A brute force attack involves an attacker trying many **username-password combinations** repeatedly until they find valid credentials. This attack is usually automated using tools or scripts that can perform thousands of login attempts in a short time.

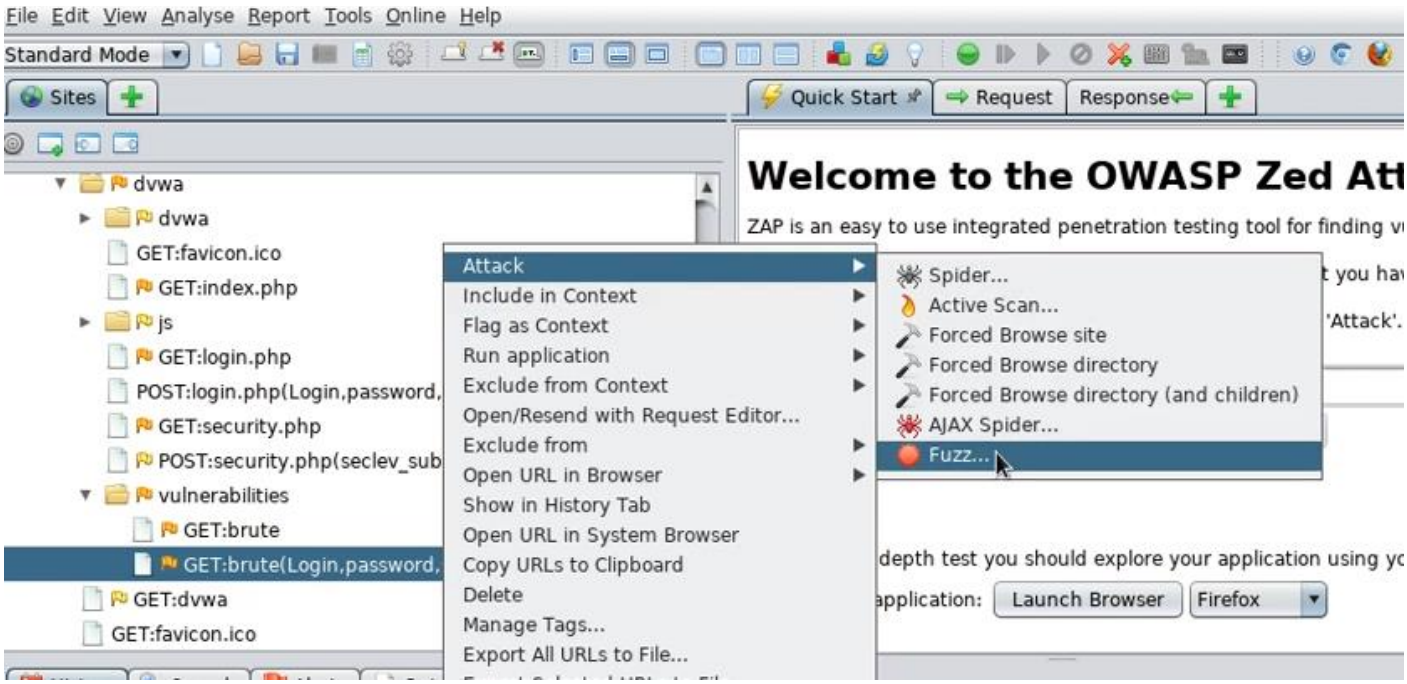
Lets brute-force a form to get credentials. Although we already know the credentials, lets see if we can use Zap to obtain credentials through a Brute-Force attack.

Steps to Reproduce:

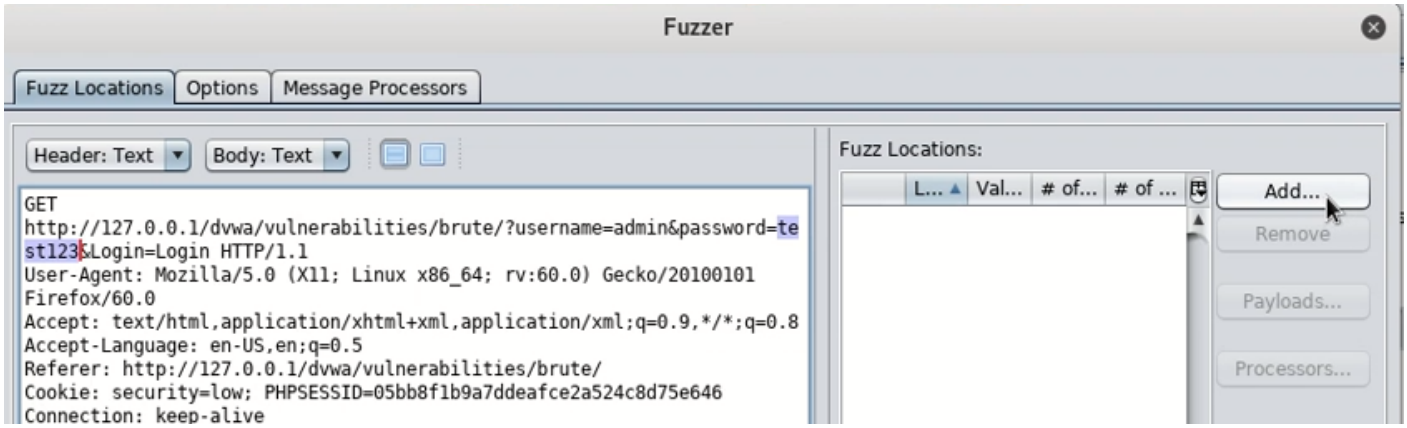
However, this process is much easier with ZAP!



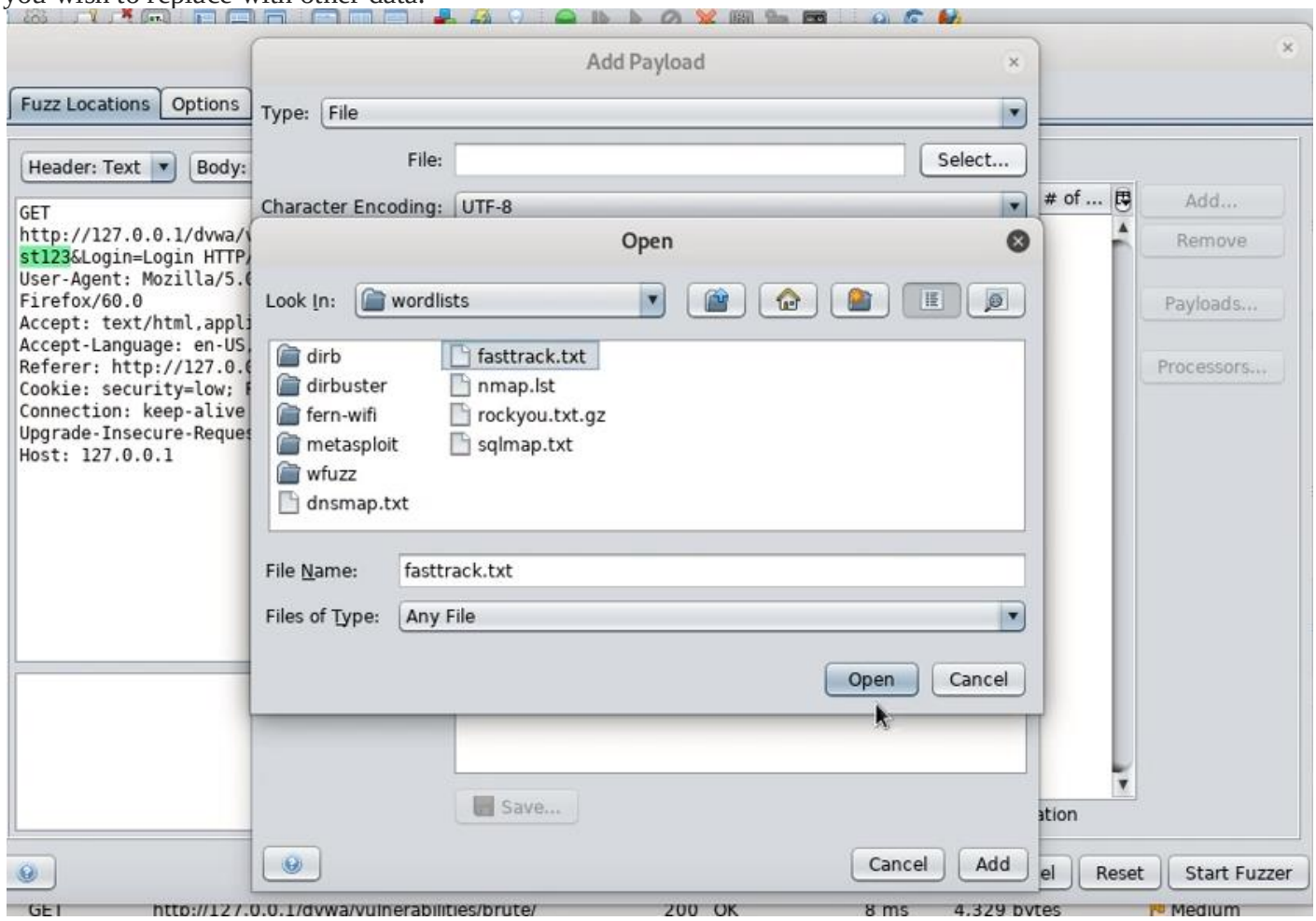
Navigate to the Brute Force page on DVWA and attempt login as “admin” with the password “test123”



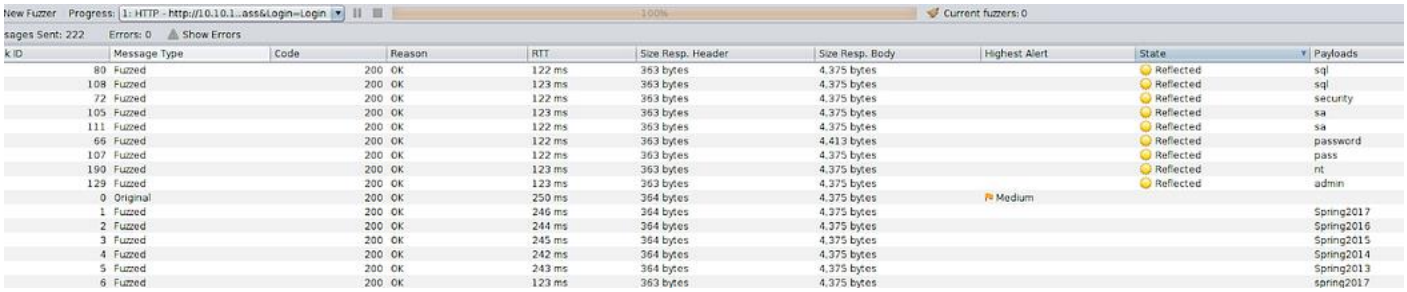
Then, find the GET request and open the Fuzz menu.



Then highlight the password you attempted and add a wordlist. This selects the area of the request you wish to replace with other data.



For speed we can use fasttrack.txt which is located in your /usr/share/wordlists if you're using Kali Linux.



k ID	Message Type	Code	Reason	RTT	Size Resp. Header	Size Resp. Body	Highest Alert	State	Payloads
80	Fuzzed	200	OK	122 ms	363 bytes	4,375 bytes		Reflected	sql
108	Fuzzed	200	OK	123 ms	363 bytes	4,375 bytes		Reflected	sql
72	Fuzzed	200	OK	122 ms	363 bytes	4,375 bytes		Reflected	security
105	Fuzzed	200	OK	123 ms	363 bytes	4,375 bytes		Reflected	sa
111	Fuzzed	200	OK	122 ms	363 bytes	4,375 bytes		Reflected	sa
66	Fuzzed	200	OK	122 ms	363 bytes	4,413 bytes		Reflected	password
107	Fuzzed	200	OK	122 ms	363 bytes	4,375 bytes		Reflected	pass
190	Fuzzed	200	OK	123 ms	363 bytes	4,375 bytes		Reflected	nt
129	Fuzzed	200	OK	123 ms	363 bytes	4,375 bytes		Reflected	admin
0	Original	200	OK	250 ms	364 bytes	4,375 bytes	Medium		
1	Fuzzed	200	OK	246 ms	364 bytes	4,375 bytes			Spring2017
2	Fuzzed	200	OK	244 ms	364 bytes	4,375 bytes			Spring2016
3	Fuzzed	200	OK	245 ms	364 bytes	4,375 bytes			Spring2015
4	Fuzzed	200	OK	242 ms	364 bytes	4,375 bytes			Spring2014
5	Fuzzed	200	OK	243 ms	364 bytes	4,375 bytes			Spring2013
6	Fuzzed	200	OK	123 ms	363 bytes	4,375 bytes			spring2017

After running the fuzzer, sort the state tab to show Reflected results first. Sometimes you will get false-positives, but you can ignore the passwords that are less than 8 characters in length.

Use ZAP to bruteforce the DVWA '*brute-force'* page. What's the password?

password

Mitigation Techniques:

- **Rate limiting / Throttling**
  - Limit the number of login attempts per IP/user per time frame.
- **Account lockout mechanisms**
  - Temporarily disable account after `n` failed attempts.
- **CAPTCHA integration**
  - Prevent automated tools from sending login requests.
- **2FA (Two-Factor Authentication)**
  - Adds an extra verification step even if credentials are stolen.
- **Strong password policies**
  - Enforce complex passwords to reduce guessability.
- **Login attempt logging and alerts**
  - Monitor and alert on suspicious login patterns.

Why it's dangerous:

- If successful, it leads to **account takeover**.
- Can be automated and targeted at **admin or user accounts**.
- May be used to discover weak passwords or gain initial access to escalate further.

Impact Level:

Medium to High
(Depends on the sensitivity of the compromised account)