Application Penetration Assessment Report of

Client Portal WordPress Plugin

Findings, Attack Narrative, and Recommendations

DATE: 03.01.24

Table of Contents

Executive Summary

This report presents the results of the White Box & Black Box penetration testing for the new **Client Portal - WordPress Plugin**. The recommendations provided in this report are structured to facilitate remediation of the identified security risks. This document serves as a formal letter of attestation for the recent authentication Service penetration testing. Evaluation ratings compare information gathered during the engagement to “best in class” criteria for security standards. We believe that the statements made in this document provide an accurate assessment of the current security of the Authentication.

We highly recommend reviewing the Summary section of business risks and High-Level Recommendations to better understand risks and discovered security issues

The intent of an application assessment is to dynamically identify and assess the impact of potential security vulnerabilities within the application. During this assessment, manual testing tools and techniques were employed to discover and exploit possible vulnerabilities.

All testing activities were conducted against the stage environment to limit the impact of any service disruptions.

Testing was conducted from both an unauthenticated and authenticated context. Unauthenticated testing examines the exterior security posture of an application and looks for vulnerabilities that do not require authentication to exploit, while authenticated tests focus on discovering and exploiting vulnerabilities on portions of the internal application that are only accessible after successful authentication. Assessors were provided both a regular user and an administrative user account to assess the internal security controls of the application.

1.1 Project Objectives

Our primary goal within this project was to provide the TE with an understanding of the current level of security in the web application authentication. We completed the following objectives to accomplish this goal:

- Identifying authentication-based threats and vulnerabilities in the application
- Comparing current security measures with industry best practices.
- Providing recommendations that can be implemented to mitigate threats and vulnerabilities and meet industry best practices

The Common Vulnerability Scoring System (CVSS) version 3.0 was used to calculate the scores of the vulnerabilities found.

1.2 Scope & Timeframe

We conducted the tests using a staging (non-production) environment with installed Client Portal WordPress Plugin. All other applications and servers were out of scope. The following hosts were considered to be in scope for testing.

The main approach was to test authentication and authorization.

1.2.1 Hostnames & IP-addresses

Stage environment with plugin installed:

- {REDACTED}

1.2.2 User Accounts provided

User provided for application itself

- {REDACTED}

User provided for WP admin panel

- {REDACTED}

1.3 Summary of Findings

Our assessment of the Client Portal WP Plugin revealed the following vulnerabilities

Security experts performed manual security testing according to the OWASP Web Application Testing Methodology, which demonstrates the following results.

Severity	Critical	High	Medium	Low	Informational
Number of issues	0	0	2	0	0

Severity scoring:
- Critical – Immediate threat to key business processes.
- High – Direct threat to key business processes.
- Medium – Indirect threat to key business processes or partial threat to business processes.
- Low – No direct threat exists. The vulnerability may be exploited using other vulnerabilities.
- Informational – This finding does not indicate vulnerability, but states a comment that notifies about design flaws and improper implementation that might cause a problem in the long run.

The exploitation of found vulnerabilities may cause full compromise of some services, stealing users' accounts, and gaining organization's and users' sensitive information.

1.4 Summary of Business Risks

In the case of **WP Plugin** and related infrastructure

Critical severity issues can lead to:

- Disruption and unavailability of main services, which company provide to their users
- Prolonged recovery from backups phase as a result of a focused attack against internal infrastructure
- Company-wide ransomware attack with the following unavailability of certain parts of the infrastructure and possible financial loss due to insufficient security insurance

High severity issues can lead to:

- Usage of CRM infrastructure for illegitimate activity (scanning of the internal network, Denial of Service attacks)
- Disclosure of confidential and Personally Identifiable Information
- Theft or exploitation of the credentials of a higher-level account

Medium severity issues can lead to:

- Disclosure of system components versions, logs and additional information about systems that might allow disgruntled employees or external malicious actors to misuse or download sensitive information outside of the company perimeter.
- Disclosure of confidential, sensitive and proprietary information related to users and companies which use CRM services

Low and **Informational** severity issues can lead to:

- Abusing business logic of main services to gain competitive advantage
- Unauthorised access to user or company confidential, private, or sensitive data
- Repudiation attacks against other users of services which allow maintaining plausible deniability

1.5 High-Level Recommendations

Taking into consideration all issues that have been discovered, we highly recommend to:

- You should validate all user input for data that users can add/edit on the server-side.
- Requests that modify data should be validated through the CSRF token to avoid
- possible Cross-Site Request Forgery attacks.
- Encode data on output
- Content Security Policy

Technical Details

2.1 Methodology

Our Penetration Testing Methodology is grounded on the following guides and standards:

- OWASP Top 10 API Security Risks
- OWASP Web Security Testing Guide
- OWASP Application Security Verification Standard

Open Web Application Security Project (OWASP) is an industry initiative for web application security. OWASP has identified the 10 most common attacks that succeed against APIs. Besides, OWASP has created Application Security Verification Standard (ASVS) which helps to identify threats, provides a basis for testing web application technical security controls, and can be used to establish a level of confidence in the security of Web applications.

The **OWASP Web Security Testing Guide (WSTG)** is a comprehensive guide to testing the security of web applications and web services. Created by the collaborative efforts of security professionals and dedicated volunteers, the WSTG provides a framework of best practices used by penetration testers and organisations all over the world.

The primary aim of the **OWASP Application Security Verification Standard (ASVS)** Project is to provide an open application security standard for web apps and web services of all types. The standard provides a basis for designing, building, and testing technical application security controls, including architectural concerns, secure development lifecycle, threat modelling, agile security including continuous integration / deployment, serverless, and configuration concerns.

2.2 Security tools used

- Manual testing: Burp Suite Pro
- Vulnerability Scan:
- Directory enumeration:
- Injection testing tools: XSSHunter, SQLmap
- Secure Code Review: IntelliJ IDEA CE

2.3 Project limitations

The Assessment was conducted against a testing environment with all limitations it provides.

Findings Details

3.1 Medium severity findings

3.1.1 Reflected Cross-Site Scripting (XSS) via Locale Parameter in JavaScript Code

Vulnerability ID: 1
Threat level: Medium

CWE	CWE-79

CVSS 3	5.4 (AV:N/AC:L/PR:N/UI:R/S:U/C:L/I:L/A:N)
Description	The vulnerability arises from the application's failure to properly sanitize the 'locale' parameter that is dynamically inserted into the webpage. This allows an attacker to inject malicious scripts into the page, which are then executed in the context of the victim's browser.
Security Impact	An attacker can exploit this vulnerability to perform actions on behalf of users, access sensitive information, redirect users to malicious websites, or perform other malicious activities.
Vulnerable Code	/view/templates/AuthorizationStep.php
Remediation	- **Input Validation**: Ensure that all input parameters, especially those used in the DOM, are properly validated against a strict specification. - **Output Encoding**: Apply context-appropriate output encoding when data is inserted into the DOM. - **Use Frameworks that Automatically Escape XSS**: Frameworks like React or Angular have built-in XSS protection. - **Content Security Policy (CSP)**: Implement a robust CSP to mitigate the impact of XSS vulnerabilities.
External References	https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Finding Evidence

```
<script type="text/javascript">

    let hash = window.location.hash;
    const search = hash.split('?');
    const token = search[0].replace('#/', '');
    const locale = new URLSearchParams(search[1]).get('locale');

    let urlTemplate = '<?= SupportFunc::getClientPortalAuthUrl(); ?>';

    if (token) {

        let redirectUrl = '';
        let form = new FormData();
        form.append('action', 'set_cookies')
        form.append('token', token)
        form.append('locale', locale)
        fetch('/wp-admin/admin-ajax.php', {
            method: 'POST',
            body: form
        })
            .then(response => response.json())
            .then(result => {
                redirectUrl = result.data.redirectUrl;
                setTimeout(() => {
                    console.log(redirectUrl.replace('__TOKEN__', token).replace('__LANG__', locale ? locale : 'en'));
                    window.location = redirectUrl.replace('__TOKEN__', token).replace('__LANG__', locale ? locale : 'en');
                }, 2000);
                document.cookie = '<?= COOKIE_LANG; ?>=' + locale + '; Max-Age=' + 24 * 60 * 60 + '; path=/; SameSite=None; Secure';
            })

    `
```

The key part of the code that introduces the vulnerability is:

```
const locale = new URLSearchParams(search[1]).get('locale');
```

This line extracts the 'locale' parameter directly from the URL without any sanitization or validation.

Later in the code, this 'locale' parameter is directly used in the DOM:
javascript

```
document.cookie = '<?= COOKIE_LANG; ?>=' + locale + '; Max-Age=' + 24 * 60 * 60 + '; path=/; SameSite=None;
Secure';
window.location = redirectUrl.replace('__TOKEN__', token).replace('__LANG__', locale ? locale : 'en');
```

Here, the 'locale' parameter is directly used in a cookie and in constructing the 'window.location', without encoding or escaping the input. This practice is susceptible to XSS as it allows for the injection of malicious scripts.

The goal is to close the existing context and start a script context. However, due to the nature of the code, the payload needs to be crafted carefully, especially considering it's being used in both a cookie and a URL context.

A sample payload might look like this:

```
en'; alert('XSS'); //
```

Injected with XSS Payload:

```
// Assuming the payload is: en'; alert('XSS'); //

document.cookie = '<?= COOKIE_LANG; ?>=' + 'en\'; alert(\'XSS\'); //' + '; Max-Age=' + 24 * 60 * 60 + '; path=/;
SameSite=None; Secure';
window.location = redirectUrl.replace('__TOKEN__', token).replace('__LANG__', 'en\'; alert(\'XSS\'); //' ? 'en\';
alert(\'XSS\'); //' : 'en');
```

How the Browser Interprets This:

1. **`document.cookie` line**:
    - The browser will set the cookie named `<?= COOKIE_LANG; ?>` to `en';`.
    - It will then execute `alert('XSS');`, which will show an alert box

with 'XSS' as its content.
    - The rest of the line (`//; Max-Age=86400; path=/; SameSite=None; Secure`) is treated as a comment and thus ignored.
2. **`window.location` line**:
    - The `replace` function is called on `redirectUrl`, replacing `__TOKEN__` with the value in `token`.
    - Next, it attempts to replace `__LANG__` with the XSS payload (`en'; alert('XSS'); //`).
    - The browser will evaluate the `locale` ternary operation, but since the injected script (`en'; alert('XSS'); //`) ends the string and comments out the rest of the line, the expected behavior of the `replace` function is disrupted, and the script `alert('XSS')` is executed.

Remediation

1. **Input Validation**:
    - Implement strict validation on the `locale` parameter. Ensure it only accepts values that match expected locale formats (like 'en', 'fr', etc.). This can often be done using a regular expression that matches known safe patterns.
    - Reject any input that does not conform to these expected formats.
2. **Output Encoding**:
    - For JavaScript, you can use functions to escape user input before inserting it into the script context.
3. **Use Context-Specific Escaping**:
    - Since the `locale` parameter is used in different contexts (HTML, JavaScript, URL, etc.), apply context-specific escaping. For instance, use JavaScript escaping when the data is used in a JavaScript context.
4. **Content Security Policy (CSP)**:
    - Implement a Content Security Policy as an additional layer of protection. This can help mitigate the impact of XSS vulnerabilities by restricting where resources can be loaded from and what actions scripts can perform.
    - For instance, CSP can be configured to disallow the execution of inline scripts, which would prevent most XSS attacks.

3.1.2 Reflected Cross-Site Scripting (XSS) in Email Parameter of JavaScript Cookie Handling

Vulnerability ID: 2
Threat level: Medium

CWE	CWE-79
CVSS 3	5.4 (AV:N/AC:L/PR:N/UI:R/S:U/C:L/I:L/A:N)
Description	• The vulnerability is due to the lack of proper sanitization or encoding of the `email` parameter. This parameter is directly incorporated into a cookie in a JavaScript context, without any sanitization or encoding, leading to a reflected XSS vulnerability. • An attacker can exploit this by crafting a malicious `email` parameter that injects JavaScript code into the webpage.
Security Impact	An attacker can exploit this vulnerability to perform actions on behalf of users, access sensitive information, redirect users to malicious websites, or perform other malicious activities.
Vulnerable Code	/core/includes/HiddenField.php
Remediation	- **Input Validation**: Ensure that all input parameters, especially those used in the DOM, are properly validated against a strict specification. - **Output Encoding**: Apply context-appropriate output encoding when data is inserted into the DOM. - **Use Frameworks that Automatically Escape XSS**: Frameworks like React or Angular have built-in XSS protection. - **Content Security Policy (CSP)**: Implement a robust CSP to mitigate the impact of XSS vulnerabilities.
External References	https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Vulnerable component

`renderHiddenUUIDFormField` function

```
public function renderHiddenUUIDFormField(): string
{
    $emailClear = str_replace(' ', '+', $_REQUEST['email']);
    $uuid = $_GET['uuid'] ?? null;
    $uuid = htmlspecialchars($uuid);
    return $uuid
        ? "<input size=\"1\" type=\"hidden\" value=\"$uuid\" name=\"form_fields[uuid]\" id=\"form-field-uuid\">
          <script type='text/javascript'>document.cookie = '" . COOKIE_EMAIL . "=" . $emailClear . " ; Max-Age=' + 15*60 + '; path=/; SameSite=None; Secure';</script>"
        : '';
}
```

Finding Evidence

In the provided code snippet:

```
"; alert('XSS'); var ignore="= '" . COOKIE_EMAIL . "=" . $emailClear . " ; Max-Age=' + 15*60 + '; path=/;
SameSite=None; Secure';
```

The `emailClear` variable is directly concatenated into the script. To exploit this, the payload must effectively break out of the current script context. A potentially effective payload could be:

```
"; alert('XSS'); var ignore="
```

Breaking this payload down:

1. `";` - This part ends the string that `emailClear` is being concatenated into.

2. `alert('XSS');` - This is the JavaScript code intended for execution. An `alert` is often used in PoCs due to its immediate visual feedback.

3. `var ignore="` - This portion of the payload is intended to neutralize the rest of the original line of code to avoid JavaScript errors that could prevent the payload from executing.