Fortify Audit Workbench

Developer Workbook

wordpress-scan_audited

Table of Contents

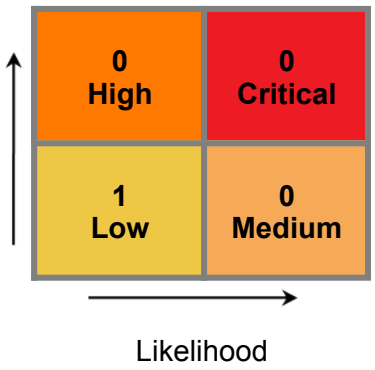
Executive Summary

This workbook is intended to provide all necessary details and information for a developer to understand and remediate the different issues discovered during the wordpress-scan_audited project audit. The information contained in this workbook is targeted at project managers and developers.

This section provides an overview of the issues uncovered during analysis.

Project Name:	wordpress-scan_audited
Project Version:	
SCA:	Results Present
WebInspect:	Results Not Present
WebInspect Agent:	Results Not Present
Other:	Results Not Present

Issues by Priority



Top Ten Critical Categories

This project does not contain any critical issues

Project Description

This section provides an overview of the Fortify scan engines used for this project, as well as the project meta-information.

SCA

Date of Last Analysis:	May 16, 2022, 2:35 PM	**Engine Version:**	21.2.3.0005
Host Name:	sp-scan02	**Certification:**	VALID
Number of Files:	41	**Lines of Code:**	2,833

Rulepack Name	Rulepack Version
Fortify Secure Coding Rules, Community, Cloud	2022.1.0.0007
Fortify Secure Coding Rules, Community, PHP	2022.1.0.0007
Fortify Secure Coding Rules, Community, Universal	2022.1.0.0007
Fortify Secure Coding Rules, Core, JavaScript	2022.1.0.0007
Fortify Secure Coding Rules, Core, PHP	2022.1.0.0007
Fortify Secure Coding Rules, Core, Universal	2022.1.0.0007
Fortify Secure Coding Rules, Extended, Configuration	2022.1.0.0007
Fortify Secure Coding Rules, Extended, Content	2022.1.0.0007
Fortify Secure Coding Rules, Extended, JavaScript	2022.1.0.0007

Issue Breakdown by Fortify Categories

The following table depicts a summary of all issues grouped vertically by Fortify Category. For each category, the total number of issues is shown by Fortify Priority Order, including information about the number of audited issues.

Category	Fortify Priority (audited/total)				Total Issues
	Critical	High	Medium	Low	
Cross-Site Scripting: Self	0	0	0	1 / 1	1 / 1

Results Outline

Cross-Site Scripting: Self (1 issue)

Abstract

Sending unvalidated data to a web browser can result in the browser executing malicious code.

Explanation

Cross-site scripting (XSS) vulnerabilities occur when: 1. Data enters a web application through an untrusted source. In the case of self-XSS, data is read from a text box or other value that can be controlled from the DOM and written back into the page using client-side code. 2. The data is included in dynamic content that is sent to a web user without validation. In the case of self-XSS, malicious content is executed as part of DOM (Document Object Model) modification. The malicious content in the case of self-XSS takes the form of a JavaScript segment, or any other type of code that the browser executes. As self-XSS is primarily an attack on oneself, it is often considered unimportant, but should be treated the same as a standard XSS weakness if one of the following can occur: - A Cross-Site Request Forgery vulnerability is identified on your website. - A social engineering attack can convince a user to attack their own account, compromising their session. **Example 1:** Consider the HTML form:

```
<div id="myDiv">
  Employee ID: <input type="text" id="eid"><br>
  ...
  <button>Show results</button>
</div>
<div id="resultsDiv">
  ...
</div>
```

The following jQuery code segment reads an employee ID from the text box, and displays it to the user.

```
$(document).ready(function(){
  $("#myDiv").on("click", "button", function(){
    var eid = $("#eid").val();
    $("resultsDiv").append(eid);
    ...
  });
});
```

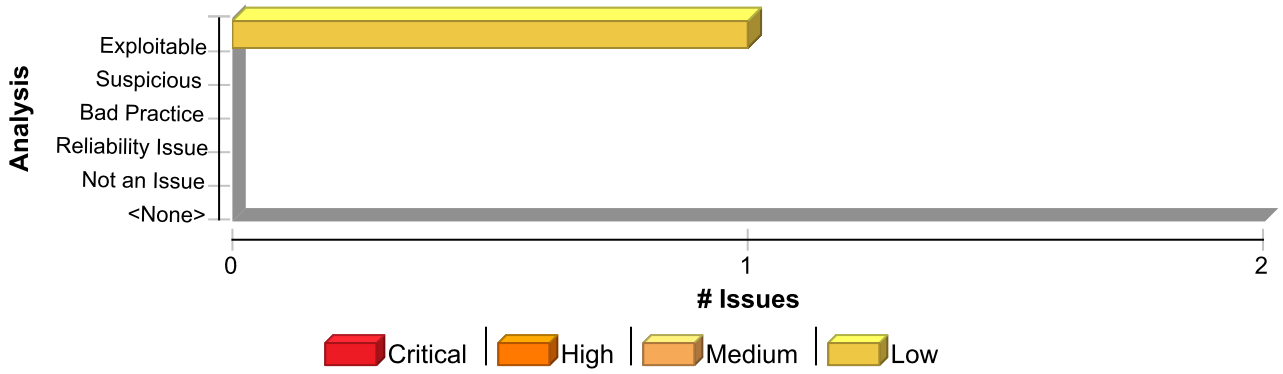
These code examples operate correctly if the employee ID from the text input with ID `eid` contains only standard alphanumeric text. If `eid` has a value that includes metacharacters or source code, then after the user clicks the button, the code is added to the DOM for the browser to execute. If an attacker can convince a user to input malicious input into the text input, then this is simply a DOM-based XSS.

Recommendation

The solution to XSS is to ensure that validation occurs in the correct places and checks are made for the correct properties. Because XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application (or just before rendered, if DOM-based). However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS. Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application might accept input through a shared data store or other trusted source, and that data store might accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means that the best way to prevent XSS vulnerabilities is to validate

everything that enters the application and leaves the application destined for the user. The most secure approach to validation for XSS is to create an allow list of safe characters that are permitted to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alphanumeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser must be considered valid input after they are encoded, such as a web design bulletin board that must accept HTML fragments from its users. A more flexible, but less secure approach is to implement a deny list, which selectively rejects or escapes potentially dangerous characters before using the input. To form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines which characters have special meaning, many web browsers try to correct common mistakes in HTML and might treat other characters as special in certain contexts. This is why we do not recommend the use of deny lists as a means to prevent XSS. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]: In the content of a block-level element (in the middle of a paragraph of text): - "<" is special because it introduces a tag. - "&" is special because it introduces a character entity. - ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error. The following principles apply to attribute values: - In attribute values enclosed in double quotes, the double quotes are special because they mark the end of the attribute value. - In attribute values enclosed in single quote, the single quotes are special because they mark the end of the attribute value. - In attribute values without any quotes, white-space characters, such as space and tab, are special. - "&" is special when used with certain attributes, because it introduces a character entity. In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters: - Space, tab, and new line are special because they mark the end of the URL. - "&" is special because it either introduces a character entity or separates CGI parameters. - Non-ASCII characters (that is, everything greater than 127 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context. - The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page. Within the body of a : - Semicolons, parentheses, curly braces, and new line characters must be filtered out in situations where text could be inserted directly into a pre-existing script tag. Server-side scripts: - Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering. Other possibilities: - If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and might bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7). After you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and might be unacceptable in circumstances where the integrity of the input must be preserved for display. If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2]. Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. For any developed application, there are no guarantees about which application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will continue to stay in sync.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Cross-Site Scripting: Self	1	0	0	1
Total	**1**	**0**	**0**	**1**

Cross-Site Scripting: Self — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 286 (Cross-Site Scripting: Self) — Low

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Audit Details

Analysis Exploitable

Audit Comments

aelchlepp: Fri May 20 2022 11:05:43 GMT+0200 (CEST)
a user can attack himself during the wizard, no reason to use innerHTML here

Source Details

Source: Read value
From: lambda
File: admin/js/eidlogin-admin.js:285

```
282  // Maybe we need to switch panel.
283  if (switchPanel) {
284  // Display the sp_entity_id.
285  let value = document.getElementById('eidlogin-settings-form-wizard-sp_entity_id').value;
286  document.getElementById(
287  'eidlogin-settings-wizard-display-sp_entity_id'
288  ).innerHTML = value;
```

Sink Details

Sink: Assignment to innerHTML
Enclosing Method: lambda()
File: admin/js/eidlogin-admin.js:286
Taint Flags: SELF_XSS, WEB

Cross-Site Scripting: Self — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 286 (Cross-Site Scripting: Self) — Low

```
283  if (switchPanel) {
284  // Display the sp_entity_id.
285  let value = document.getElementById('eidlogin-settings-form-wizard-sp_entity_id').value;
286  document.getElementById(
287  'eidlogin-settings-wizard-display-sp_entity_id'
288  ).innerHTML = value;
289
```