Simplify work life.
Achieve more.

25Q3 MANUAL AUDIT
PeopleHub Test

Tested by
Pranesh Sanathanagopalakrishnan

Reviewed by
Mohammed Sharoz

Table of Contents

List of Figures

1 Document Control

1.1 Team

Contact	Details	Role
		Pentester

1.2 List of Changes

Version	Description	Date

2 Executive Summary

2.1 Overview

TODO: Executive Summary

The manual audit uncovered a combination of exploitable vulnerabilities and structural weaknesses across Zalaris's test environment applications. These findings highlight both immediate risks and underlying security gaps that require attention.

Critical and high-severity issues should be addressed promptly to prevent potential exploitation, while broader improvements should be made to harden the environment. This report includes detailed remediation steps to support both tactical fixes and strategic enhancements.

2.2 Identified Vulnerabilities

#	CVSS	Description	Page
C1	9.8	SQL Injection (SQLi)	7
H1	7.2	Stored Cross-Site Scripting (XSS)	9

Vulnerability Overview

In the course of this penetration test **1 Critical** and **1 High** vulnerabilities were identified:

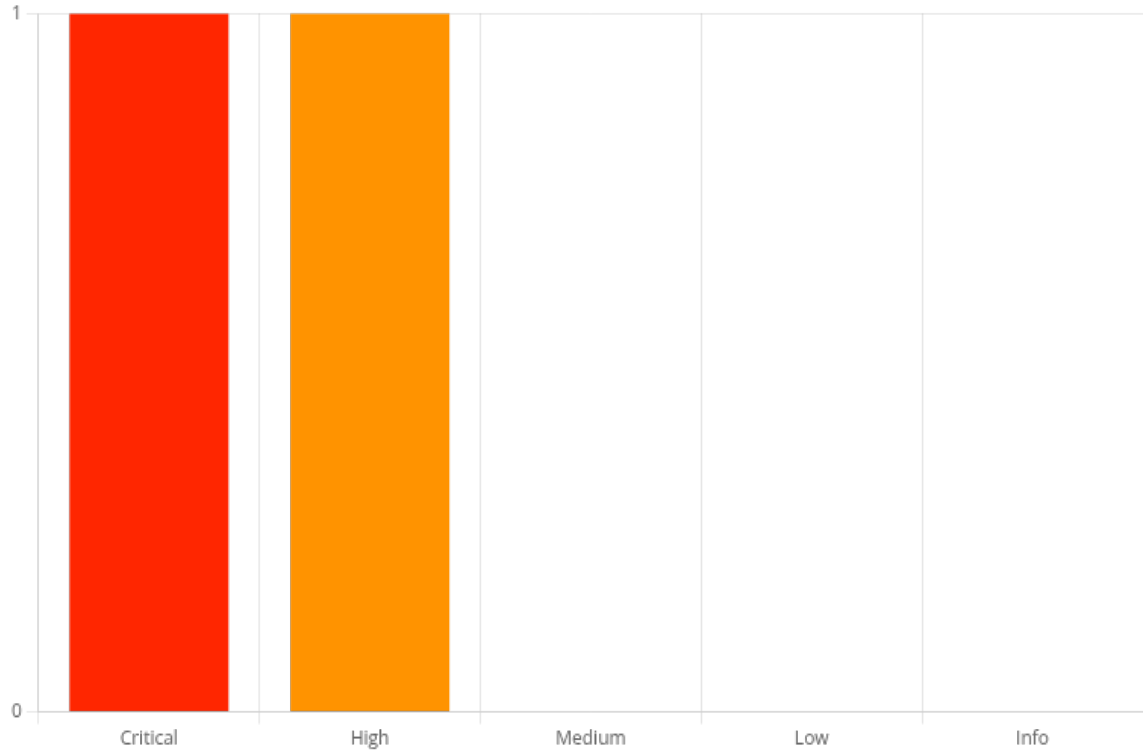

Figure 1 - Distribution of identified vulnerabilities

3 Methodology

This is a static text built into the design template. If this text changes from report to report, you can easily make it dynamic by adding a new report field and replacing the text by the used variable, e.g.:

3.1 Objective

The objective of this manual penetration test is to evaluate the security posture of applications hosted in the Zalaris test environment by identifying vulnerabilities, misconfigurations, and weaknesses that could be exploited by malicious actors. The assessment aims to simulate real-world attack scenarios using manual techniques to uncover flaws in authentication, access control, input validation, session management, and overall application logic. Findings from this engagement will help Zalaris strengthen its defenses, reduce risk exposure, and enhance secure development practices

Scope

This manual penetration test was conducted against applications hosted in the **Zalaris test environment**. The assessment focused on identifying security vulnerabilities and weaknesses across web interfaces, APIs, authentication mechanisms, and access control implementations. Testing was performed using a risk-based approach aligned with OWASP and industry best practices. Production systems and third-party components were excluded from scope.

System	Description
testportal.zalaris.com	PeopleHub Test Environment

3.2 User Accounts and Permissions

Provided Users

- 510-04000116
- 510-04000117
- 510-04000118

4 Findings

C1: SQL Injection (SQLi)	
Score	9.8 (Critical)
Vector string	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Target	TODO: affected component
References	https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Overview

The web application processed user input in an insecure manner and was thus vulnerable to SQL injection. In an SQL injection attack, special input values in the web application are used to influence the application's SQL statements to its database. Depending on the database used and the design of the application, this may make it possible to read and modify the data stored in the database, perform administrative actions (e.g., shut down the DBMS), or in some cases even gain code execution and the accompanying complete control over the vulnerable server.

Details

We identified an SQL injection vulnerability in the web application and were able to access stored data in the database as a result.

TODO: technical description

SQL Injection is a common server-side vulnerability in web applications. It occurs when software developers create dynamic database queries that contain user input. In an attack, user input is crafted in such a way that the originally intended action of an SQL statement is changed. SQL injection vulnerabilities result from an application's failure to dynamically create database queries insecurely and to properly validate user input. They are based on the fact that the SQL language basically does not distinguish between control characters and data characters. In order to use a control character in the data part of an SQL statement, it must be encoded or escaped appropriately beforehand.

An SQL injection attack is therefore essentially carried out by inserting a control character such as `'` (single apostrophe) into the user input to place new commands that were not present in the original SQL statement. A simple example will demonstrate this process. The following SELECT statement contains a variable userId. The purpose of this statement is to get data of a user with a specific user id from the Users table.

```
sqlStmnt = 'SELECT * FROM Users WHERE UserId = ' + userId;
```

An attacker could now use special user input to change the original intent of the SQL statement. For example, he could use the string `' or 1=1` as user input. In this case, the application would construct the following SQL statement:

```
sqlStmnt = 'SELECT * FROM Users WHERE UserId = ' + ' or 1=1;
```

Instead of the data of a user with a specific user ID, the data of all users in the table is now returned to the attacker after executing the statement. This gives an attacker the ability to control the SQL statement in his own favor.

There are a number of variants of SQL injection vulnerabilities, attacks and techniques that occur in different situations and depending on the database system used. However, what they all have in common is that, as in the example above, user input is always used to dynamically construct SQL statements. Successful SQL injection attacks can have far-reaching consequences. One would be the loss of confidentiality and integrity of the stored data. Attackers could gain read and possibly write access to sensitive data in the database. SQL injection could also compromise the authentication and authorization of the web application, allowing attackers to bypass existing access controls. In some cases, SQL injection can also be used to gain code execution, allowing an attacker to gain complete control over the vulnerable server.

Recommendation

- Use prepared statements throughout the application to effectively avoid SQL injection vulnerabilities. Prepared statements are parameterized statements and ensure that even if input values are manipulated, an attacker is unable to change the original intent of an SQL statement.
- Use existing stored procedures by default where possible. Typically, stored procedures are implemented as secure parameterized queries and thus protect against SQL injections.
- Always validate all user input. Ensure that only input that is expected and valid for the application is accepted. You should not sanitize potentially malicious input.
- To reduce the potential damage of a successful SQL Injection attack, you should minimize the assigned privileges of the database user used according to the principle of least privilege.
- For detailed information and assistance on how to prevent SQL Injection vulnerabilities, see OWASP's linked SQL Injection Prevention Cheat Sheet.

H1: Stored Cross-Site Scripting (XSS)	
Score	7.2 (High)
Vector string	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:L/I:L/A:N
Target	-
References	https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Overview

At the time of testing, the web application stored user input unchecked and later included it in HTTP responses in an insecure manner. It was thus vulnerable to stored cross-site scripting (XSS) attacks.
Exploitation of Stored XSS vulnerabilities does not require user interaction, making them more dangerous than Reflected XSS vulnerabilities.

Details

We were able to identify a stored XSS vulnerability in the web application during testing. Due to incorrect validation and encoding of data, we were able to inject malicious scripts into the web application and store them persistently.

TODO: technical description

Cross-site scripting (XSS) is a common web security vulnerability where malicious scripts can be injected into web applications due to insufficient validation or encoding of data. In XSS attacks, attackers embed JavaScript code in the content delivered by the vulnerable web application.

The goal in stored XSS attacks is to place script code on pages visited by other users. Simply visiting the affected subpage is enough for the script code to be executed in the victim's web browser.

For an attack, malicious scripts are injected into the web application by the attacker and stored and included in subsequent HTTP responses of the application. The malicious script is ultimately executed in the victim's web browser and can potentially access cookies, session tokens or other sensitive information.

If the attack is successful, an attacker gains control over web application functions and data in the victim's context. If the affected user has privileged access, an attacker may be able to gain complete control over the web application.

Recommendation

- Ensure that all processed data is filtered as rigorously as possible. Filtering and validation should be done based on expected and valid inputs.
- Data should be encoded before the web application includes it in HTTP responses. Encoding should be done contextually, that is, depending on where the web application inserts data in the HTML document, the appropriate encoding syntax must be considered.

- The HTTP headers `Content-Type` (e.g. `text/plain`) and `X-Content-Type-Options: nosniff` can be set for HTTP responses that do not contain HTML and JavaScript.
- We recommend to additionally use a Content Security Policy (CSP) to control which client-side scripts are allowed and which are forbidden.
- Detailed information and help on preventing XSS can be found in the linked Cross-Site Scripting Prevention Cheat Sheet from OWASP.

5 Disclaimer

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

A Appendix

A.1 Static Appendix Section

TODO: Appendix section content