Portswigger: Blind SQL injection with conditional errors

Sam Newman

April 2025

1 Website in Question

This website allows users to shop for different items on a webpage. It uses a tracking cookie (trackingID) for analytics, and sends this to an SQL server to see whether the user is familiar or not.

2 Business Impact

This vulnerability allows users to silently grab information from the sql server, potentially exposing any and all user passwords and any other information stored in the database. In this case, it allows the attacker to log in as an administrator.

3 How It Works

This vulnerability is less straightforward than a normal or even blind SQL injection. First I had to figure out what I could do to cause the server to error. i began with a simple attack, injecting ' OR 1=1 – into the tracking cookie. This resulted in a 200 OK response, so I knew it was vulnerable to injection. Next I tried many variations of selecting administrator password, with cases and other SQL commands. Everything besides the most simple 1=1 would return an error, so I retraced my steps back to the SQL version. I thought it was MySQL, but after investigating more with SUBSTRING, I realized that using SUBSTR instead would result in a 200 OK response, so I knew the database was using Oracle. From here, I constructed a payload looking like

' OR (CASE WHEN SUBSTR((SELECT password FROM users WHERE username='administrator') ,(position),1) = 'guess' THEN tochar(1/0) ELSE 'safe' END) = 'safe'–

which would return a 200 OK when the guessed character didn't match, and a 500 internal server error when it did, as it would attempt to divide by 0 and create a SQL error. I then used Burp Intruder with a sniper attack of 'a-z0-9' in the payload, and had it auto-pause when it encountered 'Internal Server Error' in the response. By doing this and manually doing an attack for all 20 characters of the password, I was able to see which character cause the error and therefore was at each position in the password.

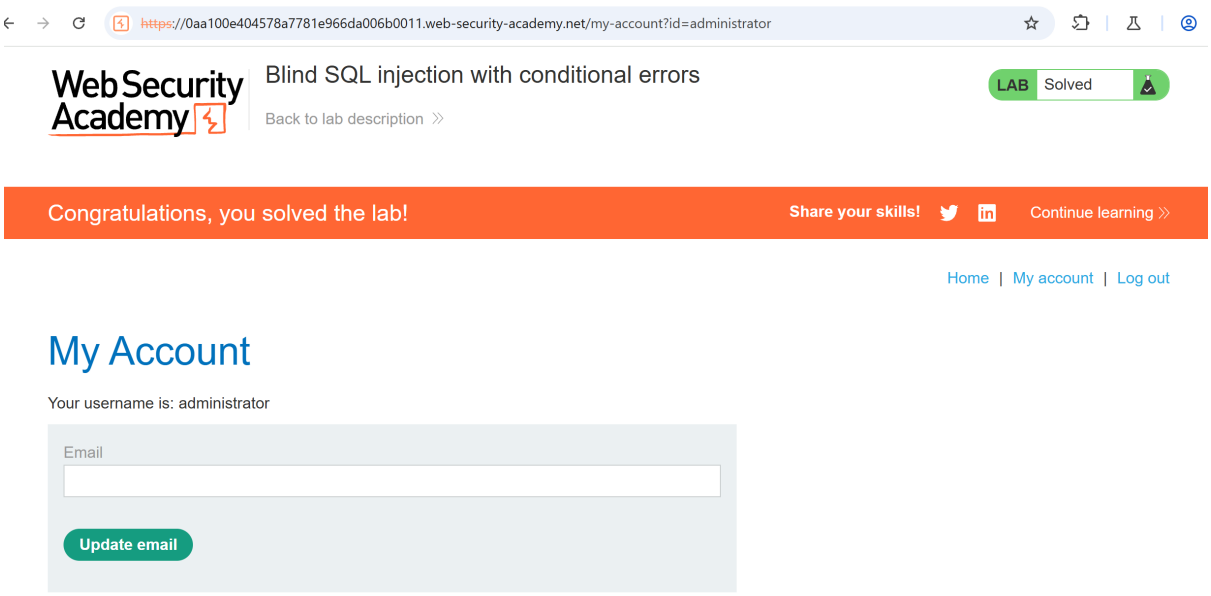

Figure 1: Solved Lab

4 Why It Works + Mitigation

The injection works because user input is not correctly and safely parametrized, allowing a user to execute SQL commands on the server. To mitigate this, the developer should simply use 'trackingID = $1' style queries, which automatically makes user input safe from SQL injection.