Lab 4: Advanced SQL Injection

Shivali Mate

I. Introduction

THIS lab explores SQL injection vulnerabilities using two different WebGoat instances (v7 avd v8). The exercises focus on advanced SQL injection techniques and blind SQL injection numeric and string. Through hands-on exploitation and controlled testing, students practiced crafting injection payloads, inferred protected data via boolean and timing channels, and reflected on mitigation strategies such as parameterized queries and prepared statements.

II. Methodology

SQL injection is a class of vulnerability that occurs when an application incorporates user-supplied input into SQL queries without sufficient validation or sanitization. An attacker can manipulate these inputs to alter query logic, exfiltrate data, or perform unauthorized actions.

Special characters and behaviors used in SQL injection

- **/* */** inline block comments
- **-, #** line comments
- **;** allows query chaining i.e. terminates a statement
- **’, +, —** commonly used in string construction or concatenation
- **CHAR()** function form to inject string bytes without quotes

Special statements and operators in SQL Injection

- **UNION** combines results of two or more SELECT statements. Each SELECT in the UNION must return the same number of columns. Corresponding columns across SELECTs should have compatible datatypes.
- **UNION ALL** includes duplicate rows i.e. does not remove duplicates.
- **JOIN** combines rows from two or more tables based on related columns. It is used for correlating data across tables.

A. SQL Injection (Advanced)

The advanced exercises required extracting passwords stored in a separate table. In the lab scenario, Table A (users) contained user metadata while Table B (user_system_data) stored user passwords.

One primary approach used the UNION operator to combine the query’s result set with a SELECT that read from the password table. Because the original query and the injected UNIONed query returned different numbers of columns, the tester aligned the column count by inserting NULL placeholders for missing columns and ensured datatype compatibility for corresponding columns

’ UNION SELECT id, username, null, null, null, password FROM user_system_data –

A successful UNION extraction is shown in the query Fig. 1, which demonstrates password data results after payload adjustments.

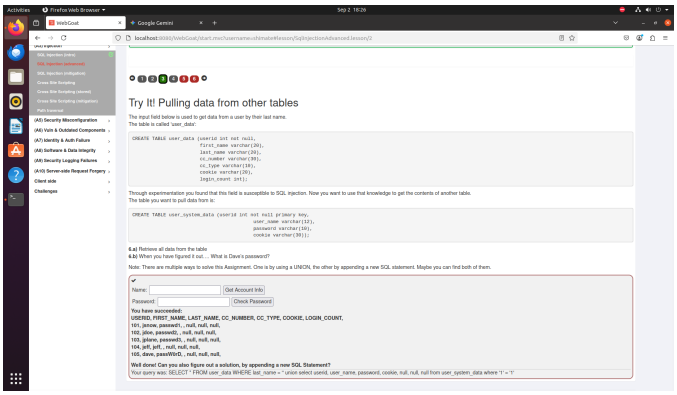

Fig. 1. Advanced SQL Injection

A second approach terminated the original query and injected a separate standalone SELECT to fetch password data. An illustrative stacked-query payload (DBMS dependent) is shown below:

’; SELECT password FROM user_system_data WHERE user_id = 1; –

B. Blind Numeric SQL Injection

The numeric blind exercise aimed to discover a pin in the pins table for a given cc_number by injecting boolean comparisons and inferring truth from the application’s responses. Binary-search style comparisons were used to cut the number of requests, then automated the validated queries with Burp Suite Intruder’s Sniper mode using a numeric payload set. Fig. 2 shows Intruder payload in Burpsuite.

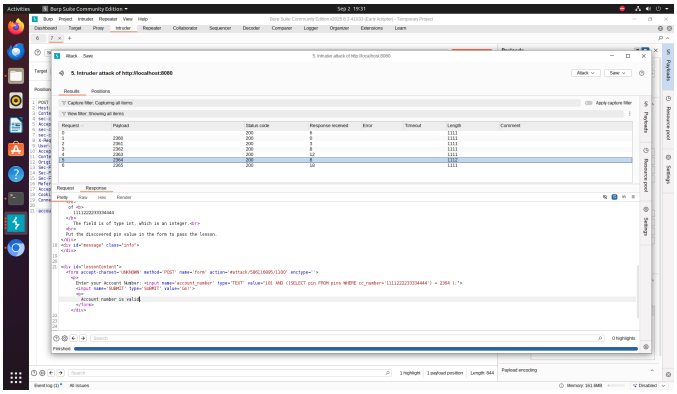

Fig. 2. Payload of BurpSuite Intruder

An illustrative numeric blind payload used in the lab took the form of a conditional comparison injected into the existing parameter, for example:

101 AND (SELECT pin FROM pins WHERE cc_number='4321432143214321') = 1234

When the condition evaluated to true the application returned behavior consistent with a "valid" account; when false it returned the "invalid" response. By iteratively adjusting the guessed number and automating the process with Burp Intruder, the correct pin value was inferred without ever directly viewing database output as shown in Fig. 3.

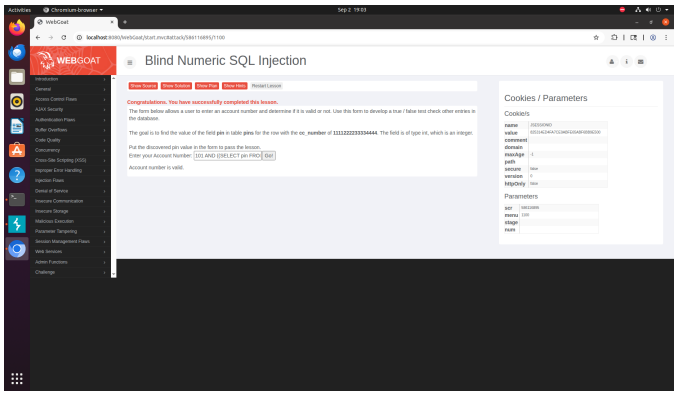

Fig. 3. Blind Numerical SQL Injection

C. Blind String SQL Injection

The string blind exercise recovered a text name from the pins table for a given cc_number by inferring characters one-by-one. The tester used SUBSTRING to isolate each character and injected boolean comparisons that bracketed the character against pivot letters, repeating the process and switching case when needed until the full name was reconstructed. This per-character method is slower than numeric blind testing but reliable when direct output is unavailable; screenshots of the queries and responses are attached in Fig. 4.

101 AND (SUBSTRING((SELECT name FROM pins WHERE cc_number='4321432143214321'), 1, 1) < 'H')

If true, the first character is before 'H'; if false, it is not. Repeating with indices 2, 3, ... and adjusted pivots reconstructs the entire name.

III. Tools and Learning Experience

WebGoat v2023.8 was used for advanced SQLi practice and realistic payload adaptation, while WebGoat 7.1 provided canonical blind-injection exercises for learning boolean/timing inference. Burp Suite acted as an intercepting proxy for request manipulation and its Intruder module automated repetitive payloads for blind testing. Browser developer tools and simple HTTP clients supported ad-hoc inspection and replay.

Limitations included WebGoat's simplified environment no real WAFs or diverse DBMS dialects, the noisiness and detectability of Intruder automation, and the limited timing precision available from browse r-level observation.

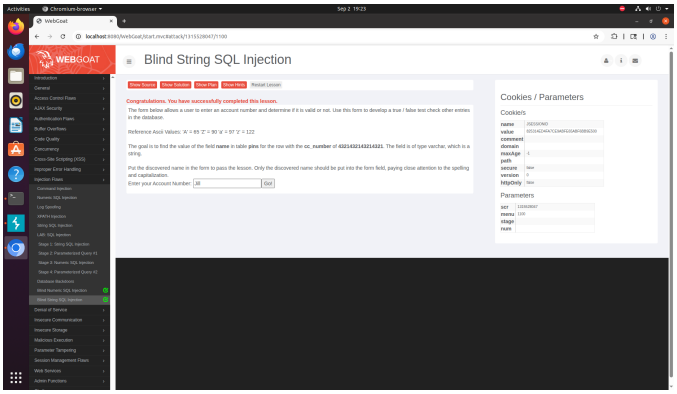

Fig. 4. Blind String SQL Injection

IV. Findings and Discussion

Successful exploitation depended heavily on schema and query awareness: UNION attacks required matching column counts and datatypes which can be often solved with NULL placeholders, while stacked queries depended on whether the DBMS accepted multi-statement inputs. Blind numeric and string methods reliably recovered secrets when direct output was suppressed but were far more query-intensive; binary or bracketing search and automation reduced effort at the cost of greater traffic and detectability. Overall, automation improved speed but highlighted the trade-off between efficiency and stealth.

V. Conclusion

The lab reinforced that SQL injection both direct and blind—remains an effective attack class when input is not properly handled. The two WebGoat versions offered complementary lessons: modern attack patterns and classic inference strategies. Practical, effective defenses are straightforward: use parameterized queries or prepared statements, validate inputs, limit DB privileges, and avoid verbose error messages to thwart both direct extraction and blind inference.

References

[1] W3Schools SQL UNION. https://www.w3schools.com/sql/sql_union.asp
[2] OWASP Query Parameterization Cheat Sheet. https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet
[3] Wikipedia SQL injection (section: Parameterized statements). https://en.wikipedia.org/wiki/SQL_injection#Parameterized_statements
[4] How and why to use parameterized queries (MSDN blog). https://blogs.msdn.microsoft.com/sqlphp/2008/09/30/how-and-why-to-use-parameterized-queries/
[5] MSDN SQL documentation (prepared statements). https://msdn.microsoft.com/en-us/library/cc296201(SQL.90).aspx
[6] Wikipedia Prepared statement. https://en.wikipedia.org/wiki/Prepared_statement
[7] WebGoat 7.1 Docker Hub https://hub.docker.com/r/webgoat/webgoat-7.1/