|  |
| --- |
| Developer’s Hub Corporation |
| Cybersecurity Internship Report |
| Made by: Tayyab Khurram  Week: 1  Student-ID: DHC-3381 |

|  |
| --- |
|  |

# Week 1: Security Assessment

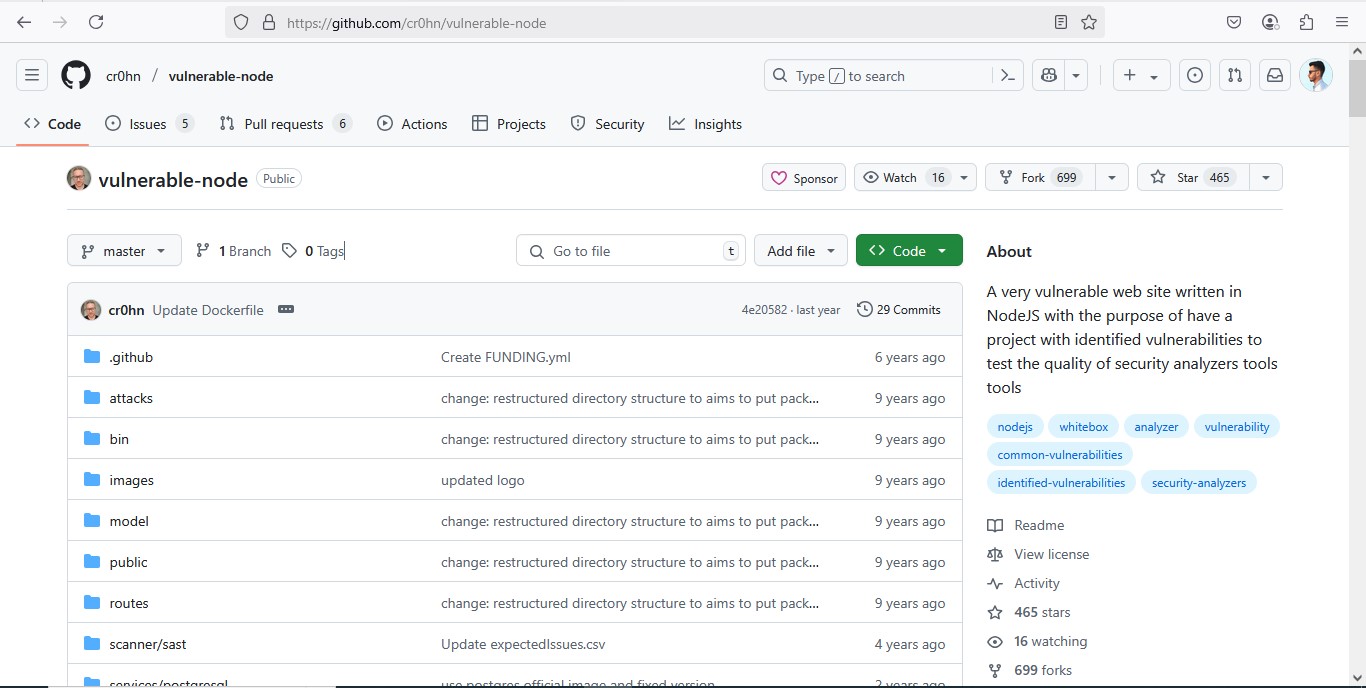
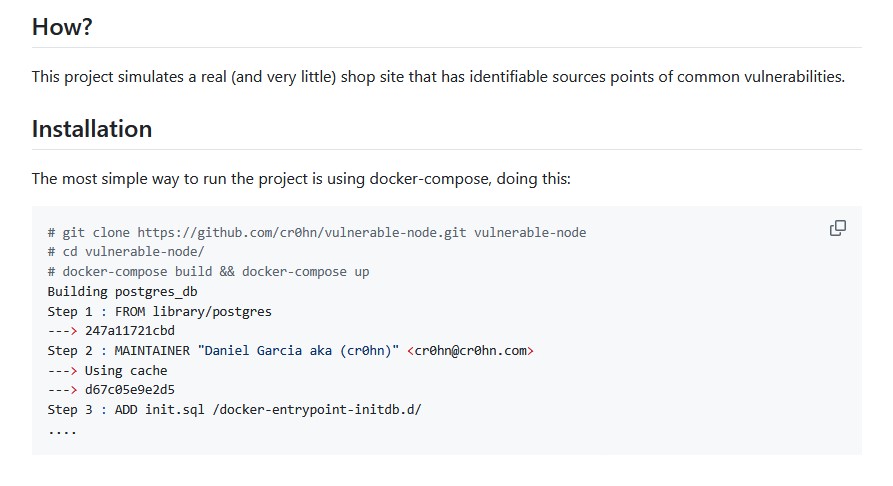
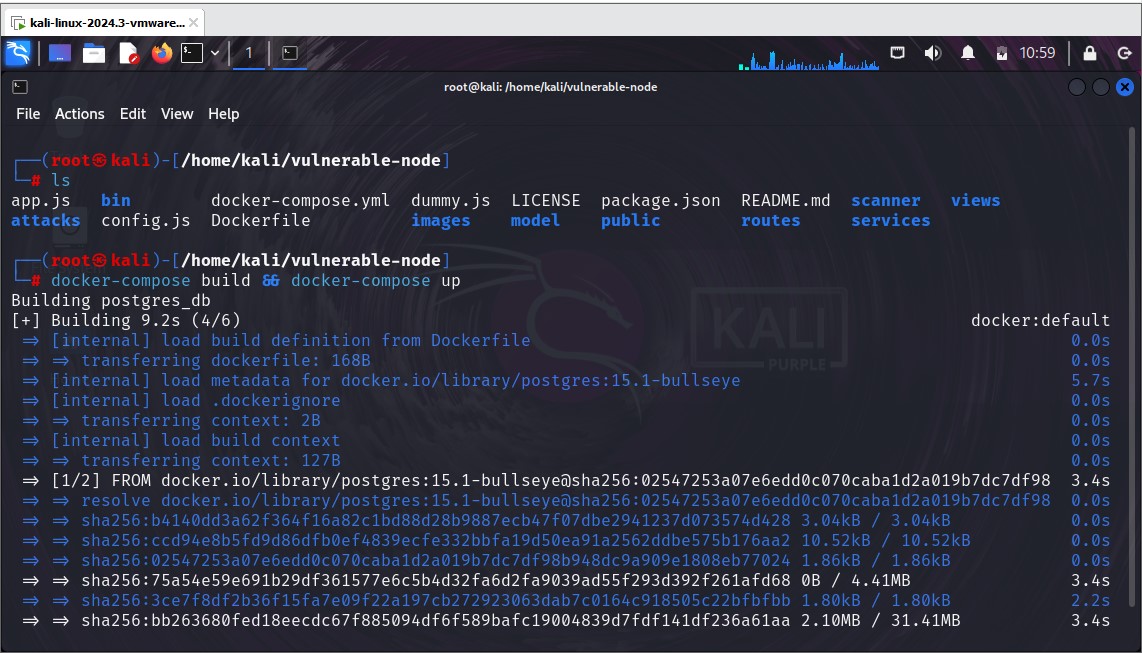
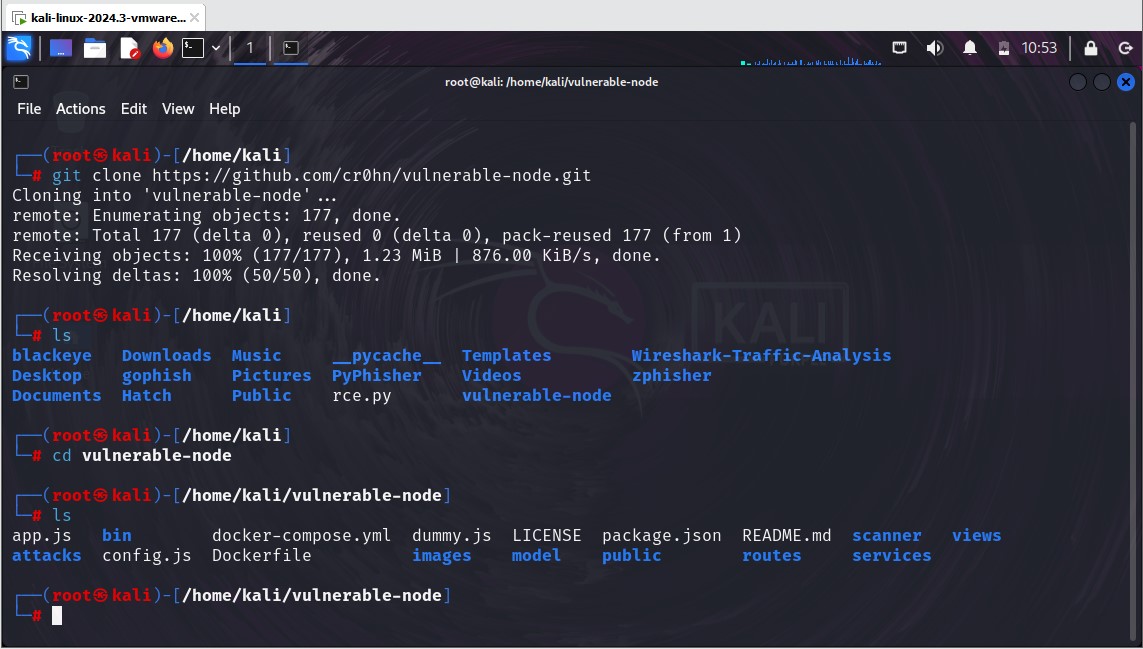
## 1. Understand the Application

Set up the application:

Take any mock web base application from github for cyber security testing

* npm install
* npm start

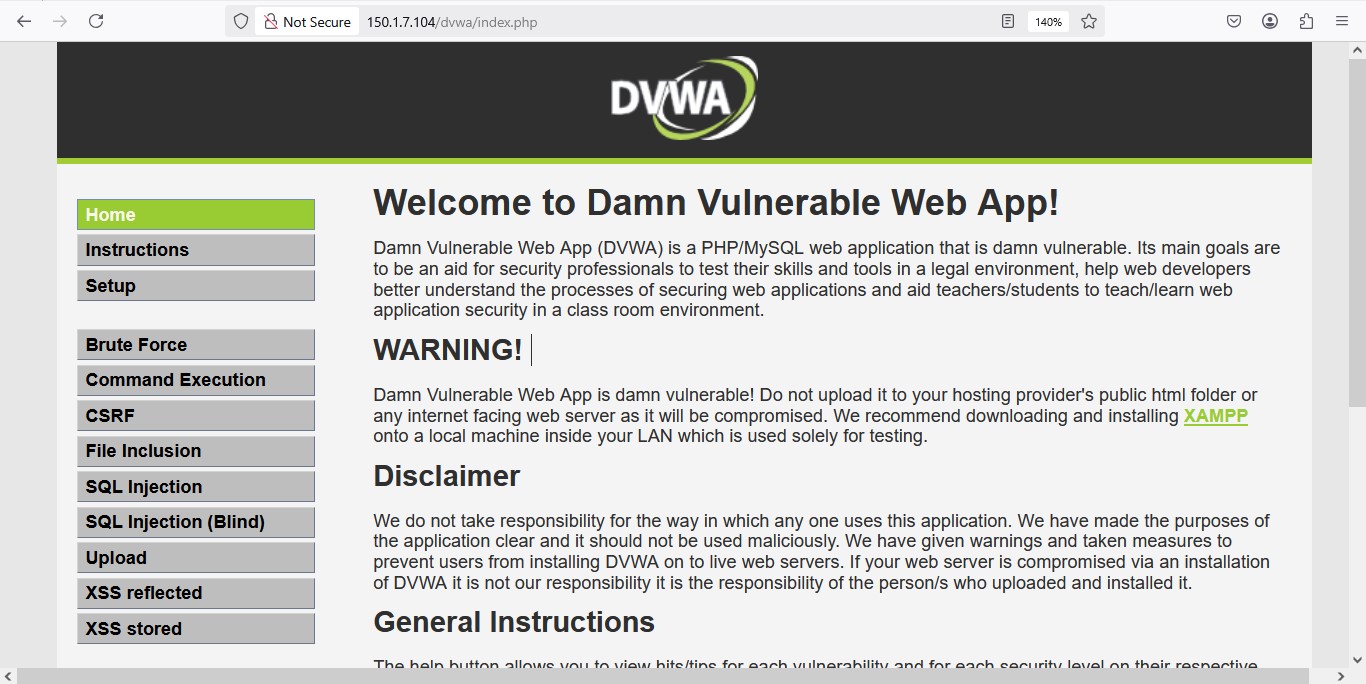
● Explore the app at <http://localhost:3000>

.

**Focus Areas**:

* Check for:
  1. Cross-Site Scripting (XSS).

○ Weak password storage.

○ Simple security misconfiguration

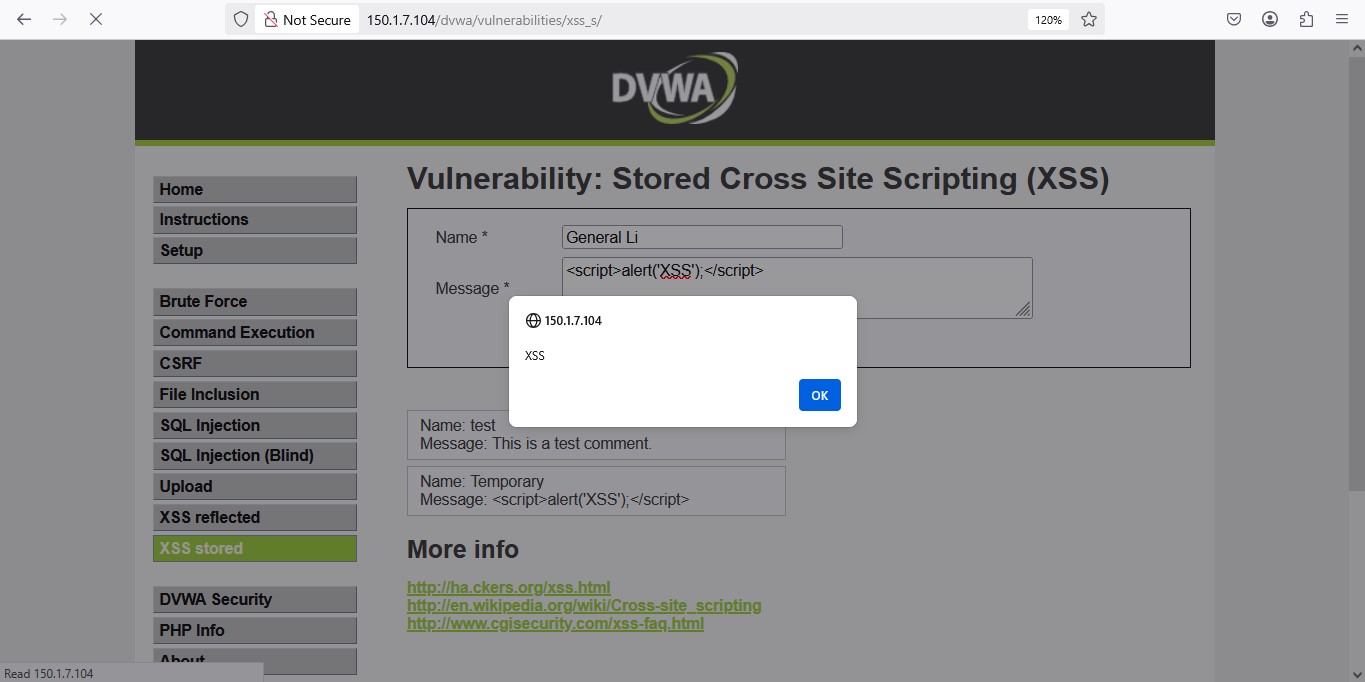
**Cross Site Scripting Test:**  
An XSS payload (<script>alert('XSS');</script>) was injected into the **Message** field during the user signup process.

**Result:**

* The application accepted the malicious input without any client-side or server-side validation.
* The script was not executed immediately, but its acceptance indicates that the input is not being properly sanitized or encoded.

**Risk Level:**  
**High** — If the injected script is reflected or stored and later executed in a user's browser, it can lead to a range of security issues including session hijacking, data theft, or defacement of the application.

**Recommendation:**

* **Input Validation:** Implement strict validation on user inputs. Only allow expected characters for each input field.
* **Output Encoding:** Ensure that all user-supplied data is properly encoded before rendering it in the browser, especially in HTML, JavaScript, or attribute contexts.
* Use security libraries or frameworks that offer built-in XSS protection (e.g., OWASP ESAPI, React’s JSX auto-escaping, etc.).
* Conduct **regular security testing** to detect and remediate such vulnerabilities proactively.
* 

**SQL Injection Test:**  
A classic SQL injection payload (admin' OR '1'='1) was injected into the **Name** field during the signup process to test for improper handling of input within SQL queries.

**Result:**

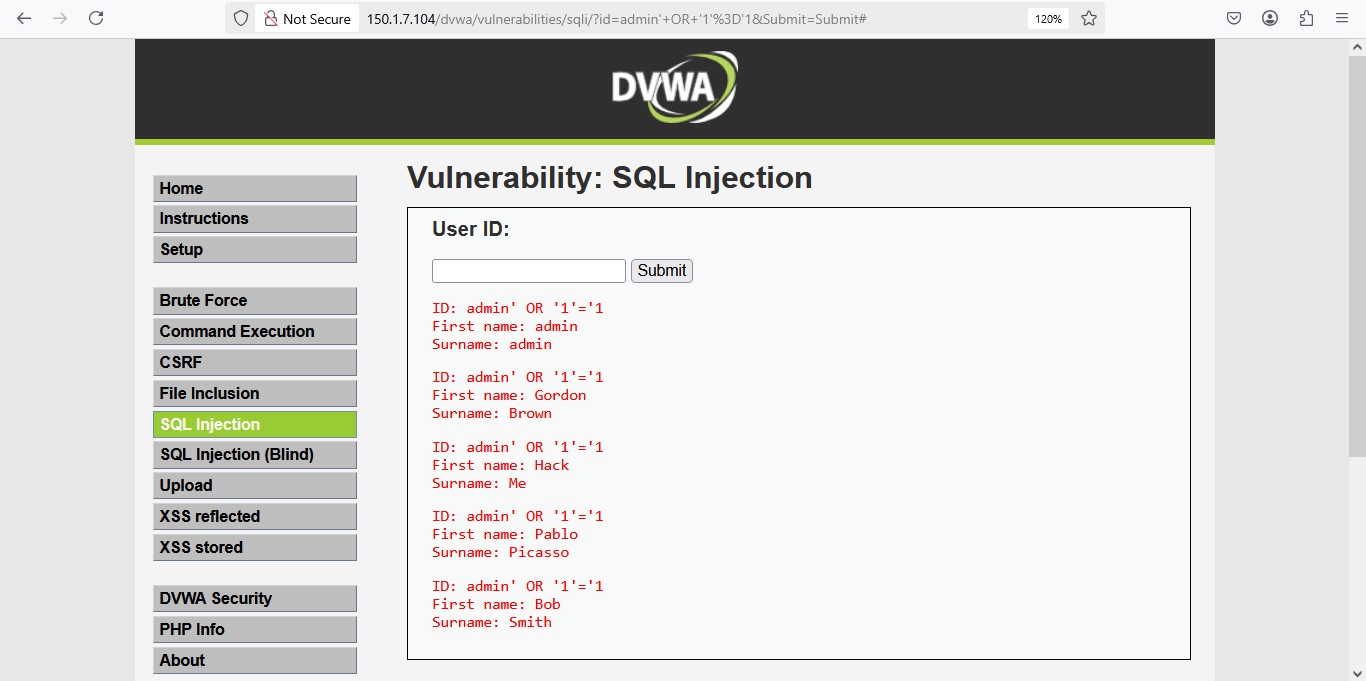
* The application accepted the input and responded abnormally, indicating that the input was directly included in an SQL statement without proper sanitization.
* This behavior confirms the presence of a SQL Injection vulnerability, allowing potential attackers to manipulate backend database queries.

**Risk Level:**  
**High** — SQL Injection can allow attackers to:

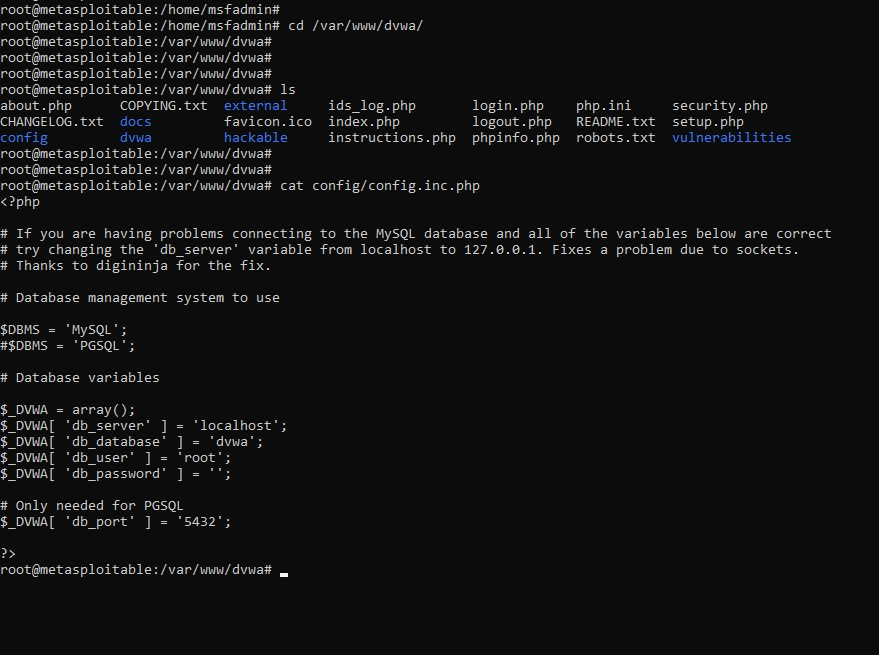
* Bypass authentication mechanisms
* Extract, modify, or delete sensitive data
* Execute administrative operations on the database
* Potentially gain full control of the database server in severe cases

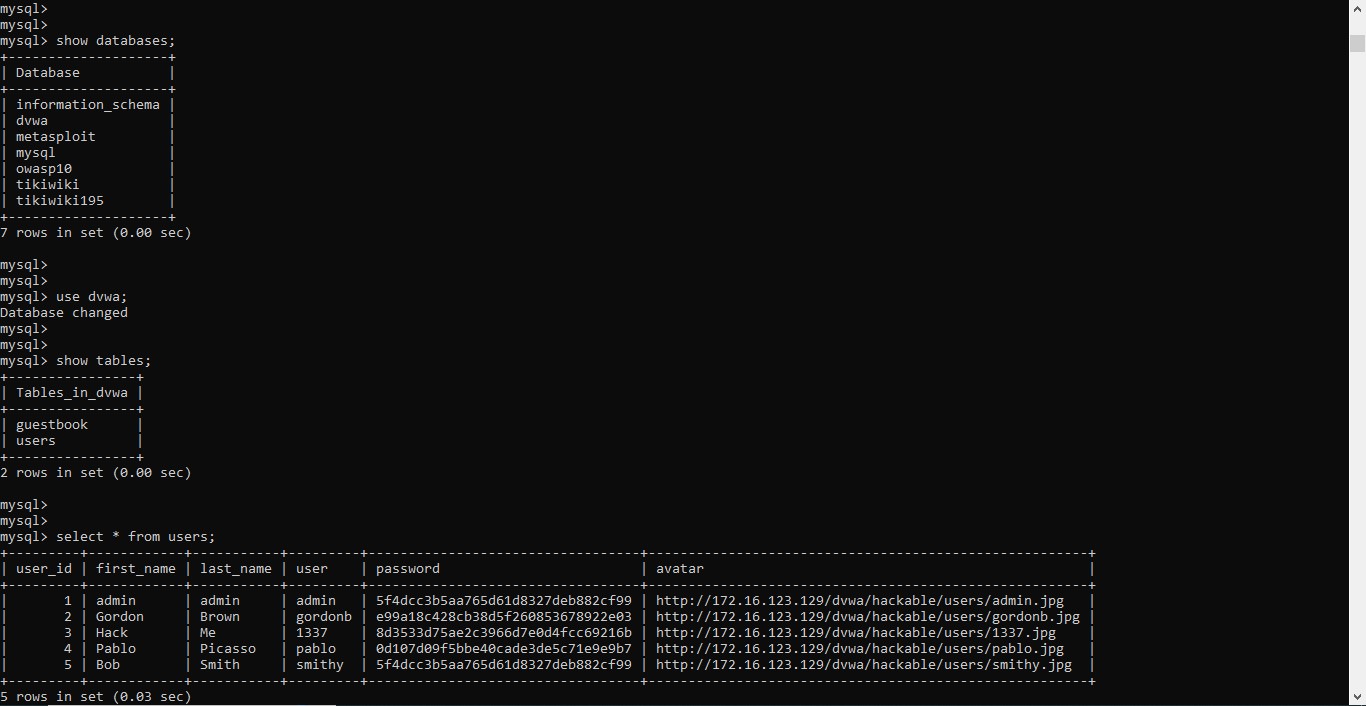
**Recommendation:**

* **Use Parameterized Queries:** All database interactions should use prepared statements or ORM methods that separate code from data.
* **Input Validation:** Validate all input fields based on expected formats and constraints. Reject any unexpected or malicious input.
* **Least Privilege Principle:** Ensure the database user has only the necessary permissions to perform required operations.
* **Error Handling:** Avoid exposing raw database error messages to users, as they can provide valuable information for attackers.



**Weak Password Storage:**





**No salting**: All passwords are stored using just a hash — and if two users have the same password, they have the same hash (e.g., admin and smithy).

**Weak hashing (MD5)**:

* 5f4dcc3b5aa765d61d8327deb882cf99 is the MD5 hash of password
* These hashes are fast to compute and widely cracked with rainbow tables.

**Easy to reverse** using free online MD5 hash databases:

* Try Googling: 5f4dcc3b5aa765d61d8327deb882cf99

**Conclusion:**

This is a **clear example of weak password storage**:

* Passwords are hashed using **unsalted MD5**, which is easily reversible.

**Recommendations:**

* Using slow, salted hashing algorithms (e.g., bcrypt, argon2).
* Never storing passwords in plaintext or with fast hashes.

# Week 2: Implementing Security Measures: