Developing a Tailored DVWA for SQL Injection Exploration

Reddyvari Venkateswara Reddy[1] ,D. Ajay[2], C. Viveka Vardhan Reddy[3], Md. Asfahan[4]
[1]Associate Professor, Department of CSE (Cybersecurity), CMR College of Engineering & Technology, Hyderabad, Telangana, India
[2,3,4]Department of CSE (Cybersecurity), CMR College of Engineering & Technology, Hyderabad, Telangana, India.
Corresponding author mail: venkatreddyvari@cmrcet.ac.in
Co-authors: vvrchennam@gmail.com, ajay.deekonda1@gmail.com, mohdasfahan786@gmail.com

Abstract - SQL injection remains a significant security challenge in modern web development, posing data integrity and confidentiality risks. This project introduces a deliberately insecure web application built with MySQL, Node.js, Express, and React to simulate and analyze SQL injection vulnerabilities. Designed as an educational tool, the application provides an interactive platform to explore common attack vectors and their impact on database security. The project includes diverse scenarios, ranging from simple injections to advanced exploitation techniques, offering a comprehensive understanding of how such vulnerabilities arise. Additionally, the work emphasizes practical mitigation strategies, such as robust input validation and the use of parameterized queries, to prevent SQL injection attacks. By enabling hands-on experimentation in a controlled environment, this project aims to advance cybersecurity education and foster the adoption of secure coding practices in web development.

Keywords- SQL injection, cybersecurity, web application vulnerabilities, Node.js, Express.js, MySQL, React.js, secure coding, input validation, database security, cybersecurity education

INTRODUCTION

In an era where online platforms are central to business operations, social interaction, and information exchange, the security of web applications is more important than ever. However, the inherent complexity and scale of these systems often lead to vulnerabilities that, when unaddressed, can be exploited by attackers to devastating effect [1], [3]. Much like a well-defended fortress, a web application's integrity relies on the strength and security of each individual component, from its databases and server configurations to its codebase and user input handling [6]. Vulnerabilities, often introduced unintentionally through coding oversights, poor configurations, or outdated software, serve as potential gateways for attackers seeking to compromise sensitive data and disrupt application functionality. Among the most widespread and damaging of these security risks is SQL injection (SQLi), a technique that enables attackers to interfere with the database layer of an application through manipulated inputs [2].

SQL injection happens when attackers use input fields to insert harmful SQL statements into the application's database query functions. Through SQLi, attackers can bypass authentication, access sensitive user information, alter or delete data, and even gain control over the underlying database server [5]. This form of attack not only exposes user credentials, financial records, and communication histories, but it can also result in extensive damage to the organization's financial standing, customer trust, and legal liabilities [6]. Despite ongoing advancements in web security, SQL injection remains a significant threat due to its ease of exploitation and the potentially catastrophic outcomes it can cause [10].

To shed light on the mechanisms of SQL injection attacks and the critical importance of preventive measures, we have developed a vulnerable web application designed explicitly for studying SQL injection [4]. This project, built with MySQL, Node.js, Express, and React, offers a controlled environment for users to interact with and observe firsthand SQL injection (SQLi) vulnerabilities [7], [9]. By simulating various insecure coding practices, the application demonstrates how different SQL injection attacks function, including techniques for bypassing login screens, manipulating database entries, and retrieving confidential information [8]. The project serves as an educational platform where students, developers, and security enthusiasts can explore the anatomy of SQL injection attacks and gain a practical understanding of the risks and mitigation strategies associated with this vulnerability [4].

A key lesson from this project is the necessity of implementing secure coding practices and proactive security measures. Safeguarding applications from

SQL injection and XSS attacks requires techniques like parameterized queries, input validation, and regular security audits [2], [7]. Through detailed demonstrations and hands-on engagement, this research emphasizes the practical steps developers and organizations can take to minimize security risks [3],[10]. Our project provides insights into common vulnerabilities while promoting awareness of secure development practices that are essential in today's rapidly changing threat landscape [5].

In summary, this project combines theoretical and practical knowledge to highlight the significance of web application security. It serves as a valuable resource for understanding SQL injection and similar vulnerabilities, underscoring the importance of robust security practices to protect both organizations and users from cyber threats [6], [10]. By equipping developers with knowledge of these attack vectors and their remedies, this project contributes to a safer digital environment where applications are resilient against both conventional and emerging threats [8], [9].

LITERATURE REVIEW

A. *Aditya N. Deshpande, Dikshant G. Borse, Atharv C.Kulkarni, "Survey on SQL Injection Attack Detection", ISSN:0039-2049,2024.*

This paper analyzes SQL injection attacks on web applications, focusing on the types of SQL injection vulnerabilities and the potential damage they may cause. It proposes prevention strategies, including machine learning algorithms, regular expressions, encryption, tokenization, and dynamic parsing, to detect and prevent these attacks. Emphasizing the importance of secure programming practices and developer education, the paper also critiques existing detection tools and advocates for novel approaches, such as machine learning and compiler techniques, to strengthen web application security.

B. *Mahmoud Baklizi, Issa Atoum, Nibras Abdullah, Ola A.Al-Wesabi, Ahmed Ali Otoom, Mohammad Al-Sheikh Hasan, "A Technical Review of SQL Injection Tools and Methods: A Case Study of SQL Map", IJISAE, ISSN:2147-6799,2022*

This paper examines SQL injection attacks, which allow unauthorized access to websites and databases, potentially leading to data theft or destruction. It evaluates current detection tools, particularly sqlmap, for their effectiveness, and demonstrates various attack types with examples. By implementing sqlmap in a controlled environment, the study reveals its capabilities for accessing databases and retrieving sensitive information, highlighting its role in enhancing web application security.

C. *Tanzila Hasan Pinky, Kaniz Ferdous, Jarin Tasnim, Kazi Shohaib Islam, "Understanding SQL Injection Attacks: Best Practices for Web Application Security", IJISRT, ISSN:2456-2165, 2024*

This paper investigates SQL injection, a security flaw that lets attackers execute SQL commands in a web application's database by exploiting poorly validated user input. Identifying SQL injection vulnerabilities requires locating areas within a web application that are vulnerable to harmful input and ensuring user inputs undergo appropriate validation. This project aims to create an attack chain to test websites for security weaknesses and possible entry points that attackers could use to break into the system. Unlike most tools, which only scan specific URLs, this project expands detection to ensure that no SQL injection vulnerabilities exist across all pages of a website.

D. *SM Sarwar Mahmud, Taofica Amrine, Muhammad Anwarul Azim, "SQL Injection Attack Vulnerabilities of Web Application and Detection", International Journal of Computer Applications(0975-8887),2023*

This paper highlights SQL injection as a critical security risk in database-driven web applications, where attackers can exploit vulnerabilities to steal, alter, or destroy sensitive data. It aims to create a dataset of SQL injection payloads to enhance vulnerability prediction and provide practical tools for penetration testers. The proposed method includes new payloads and a Web Application Firewall (WAF), which effectively reduces SQL injection attacks, strengthening web application security.

SQL INJECTION ATTACKS

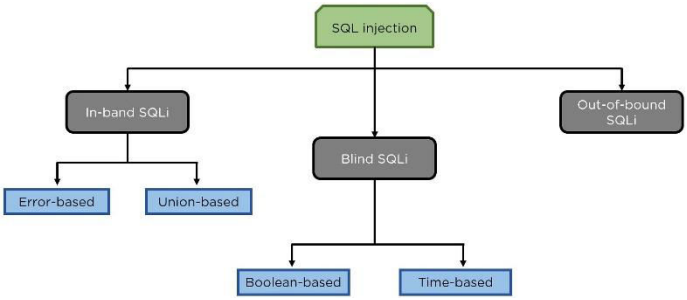

Fig1: Types of SQL Injection Attack

1. UNION-BASED SQL INJECTION:

In a union-based attack, the attacker uses the SQL UNION operator to combine their own malicious query with a legitimate one. This lets them access data from other tables in the database [5], [9]. As a result, they can combine their query results with the application's output.

2. ERROR-BASED SQL INJECTION:

This type exploits error messages generated by the database when invalid queries are run. Attackers can create an error on purpose to discover the structure of the database [1]. This includes finding out the names of tables and columns, which they can later use to access data.

3. BOOLEAN-BASED BLIND SQL INJECTION:

In Boolean-based attacks, attackers test the database by sending queries that return a "true" or "false" response without actually revealing data [2]. This is often done by altering the query logic to see if certain conditions are met, allowing them to map out database information based on the responses.

4. TIME-BASED BLIND SQL INJECTION:

In time-based attacks, the database response time is used as a clue. Attackers add delays to queries (such as SLEEP statements) and infer information based on how long it takes for the database to respond [6]. This technique reveals data indirectly by measuring response times.

5. OUT-OF-BAND SQL INJECTION:

This attack uses channels outside of the standard response to extract data, often through network or DNS requests. Out-of-band injections are useful when the database cannot respond directly or when the attacker needs to avoid detection, as they do not rely on immediate HTTP responses [3], [8].

6. STACKED QUERIES SQL INJECTION:

Stacked query attacks involve executing multiple SQL statements in a single query. This allows attackers to run additional commands (such as DROP or INSERT) along with the intended query, potentially modifying the database or altering data [7].

SQL INJECTION ATTACK TECHNIQUES

1. TAUTOLOGY-BASED INJECTION:

In this technique, attackers use statements that are always true (like 1=1) to manipulate the query logic [1]. For instance, in a login form, entering OR 1=1-- in the username field could bypass authentication checks, granting access without proper credentials.

2. COMMENTING OUT:

Attackers can use SQL comments (-- or /* ... */) to ignore parts of a legitimate query. For example, appending admin'-- to a login query could bypass the password check by effectively commenting it out, allowing unauthorized access [4].

3. PIGGYBACKED QUERIES:

This technique involves injecting additional queries (known as stacked queries) separated by a semicolon [7]. For example, an attacker could use '; DROP TABLE users;-- to drop a table from the database if multiple statements are allowed.

4. BLIND INJECTION USING BOOLEAN STATEMENTS:

In this technique, attackers send a series of queries with true or false conditions to observe the application's responses, such as altering a query to WHERE id = 1 AND 1=1 or WHERE id = 1 AND 1=2. This allows the attacker to deduce information based on different responses [2].

5. BLIND INJECTION USING TIME DELAYS:

Time delays help attackers confirm the presence of vulnerabilities by introducing commands like SLEEP(5) in the query [6]. If the application takes longer to respond, they know a vulnerable spot exists and can proceed with further attacks.

6. UNION SELECT INJECTION:

With union select, attackers append the UNION operator to combine their malicious query with a legitimate one [5]. This method lets you gather data from different tables by choosing the specific columns you need. This way, you can combine two queries effectively.

7. OUT-OF-BAND INJECTION:

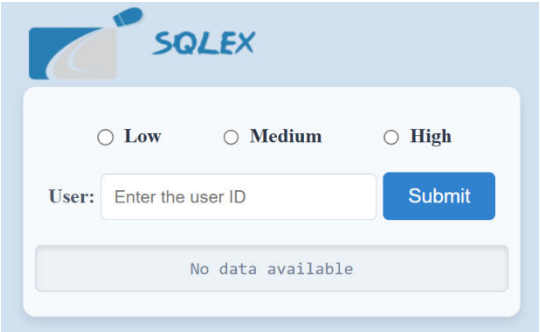

Fig2: User Interface

This technique relies on extracting data through external communication methods (like DNS or HTTP requests) rather than directly through the application's response [3]. This is useful in scenarios where direct interaction is limited or where attackers want to avoid detection.

8. STRING CONCATENATION INJECTION:

Some databases allow string concatenation with commands like +, ||, or CONCAT. Attackers use these operators to bypass filters or modify queries by altering or adding conditions to the query dynamically [9].

PROBLEM STATEMENT

Create a deliberately vulnerable web application using MySQL, Node.js, Express, and React to demonstrate SQL injection attacks. This interactive platform will help developers and security professionals understand

SQL injection risks and learn effective prevention strategies through hands-on experience.

METHODOLOGY

The methodology for developing a full-stack website that demonstrates SQL Injection vulnerabilities includes planning, design, development, and testing phases. The primary aim is to intentionally design a web application using **Node.js**, **Express.js**, **React.js**, and **MySQL** to simulate SQL injection techniques in a controlled, educational environment [4]. This section outlines the steps followed in each stage.

1. Requirements Analysis and Planning

- **Objective Definition:** The project's goal is to create a website that deliberately includes SQL injection vulnerabilities. The objective is to educate developers and security enthusiasts on identifying and mitigating SQL injection attacks [1].
- **Technology Stack Selection:** The full-stack website was built using Node.js and Express.js for the backend, React.js for the front end, and MySQL for database management. We chose these technologies because they are popular and flexible for modern web development [7].
- **Security Consideration and Ethical Boundaries:** Clear ethical guidelines were established to ensure that the project is strictly for educational purposes. All SQL injection vulnerabilities are limited to the development environment to prevent misuse [4], [8].

2. System Design and Architecture

- **Frontend Design (React.js):** The front end was developed using React.js to provide a responsive, interactive user interface. A main part of this interface is a form where users can enter search parameters to get data. The form submission triggers requests to the backend, simulating typical user interactions with an application [4].
- **Backend Development (Node.js and Express.js):** The backend API is structured with Node.js and Express.js to handle incoming requests from the front end. API endpoints retrieve data from a MySQL database based on user input. This design makes them vulnerable to SQL injection attacks. This was done to replicate real-world insecure coding practices [6].
- **Database Design (MySQL):** The MySQL database contains tables with sample user data to be retrieved based on search criteria. A user's table, for instance, includes basic fields like id, username, email, and password. The database structure is intentionally designed to allow SQL injection attacks by not using parameterized queries or prepared statements [7].

userid	name	email
1	Alice Johnson	alice.johnson@example.com
2	Bob Smith	bob.smith@example.com
3	Charlie Brown	charlie.brown@example.com
4	Diana Prince	diana.prince@example.com
5	Ethan Hunt	ethan.hunt@example.com
NULL	NULL	NULL

Fig3: Users Table

3. Vulnerability Implementation

- **Building the Injection-Prone Query:** In the backend, we create SQL queries by combining user inputs directly into the SQL statements [1]. For example:

  let query = `SELECT * FROM users WHERE id = ${userInput};`;

  This lack of input sanitization allows attackers to input SQL code directly into the query, such as 1 = 1 OR 1, which would cause the query to fetch all rows from the user's table.

```
[nodemon] 3.1.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Connected to the MySQL database
Server is running at port 9000
```

Fig5: Backend running on port 9000

- **Creating Input Fields with SQL Injection Flaws:** A web page has a search box for users to find specific records. However, this search box has a security flaw that allows SQL injection. If someone enters harmful code, like `1=1 OR 1`, it can lead to the database returning extra data that was not intended. This page demonstrates how a basic input field, when improperly secured, can be manipulated to exploit SQL vulnerabilities [6].

4. Demonstrating SQL Injection Techniques

- **Demonstrating Basic SQL Injection:** Users are instructed to input common SQL injection strings,

such as 1=1 OR 1. When entered in the search field, the backend processes the input without sanitization, resulting in the entire dataset being returned [9].

- The query transforms from:

  SELECT * FROM users WHERE id = 1;

  to:

  SELECT * FROM users WHERE id = 1 OR 1=1;

  This query retrieves all records in the table due to the 1=1 condition, which is always true.

- **Error-Based SQL Injection:** Another form of SQL injection demonstrated is error-based injection. By inputting a value that could cause an error (e.g., using an unclosed quotation mark), users can observe database error messages, gaining insight into the database structure [7].

- **Union-Based SQL Injection:** In a controlled way, union-based SQL injection can be demonstrated, where users add a UNION SELECT query to retrieve additional information from the database [5]. For example:

  1 UNION SELECT username, password FROM users;

  This technique allows attackers to retrieve sensitive information beyond the initial query's scope by merging it with a second query.

RESULT

We ran the website in a local environment:

1. We launched the project in Visual Studio Code.
2. The project's user interface runs on the port

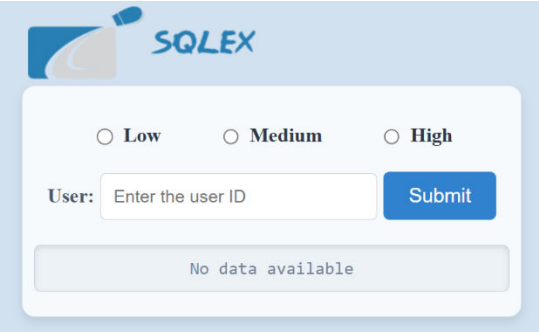

Fig4: User Interface on port 3000 on local environment

number 3000.

3. The backend is connected to a MySQL database and operates on port number 9000.
4. The user can choose the security level and retrieve user details by entering the user IDs in the input field.

Fig6: getting the user details with their ID

5. If a user enters a malicious SQL query, such as 1 OR 1, the backend retrieves all the data.

Fig7: SQL injection successful

DISCUSSION

The development of this vulnerable full-stack website highlights the critical importance of secure coding practices in modern web applications. By intentionally designing a site with SQL injection vulnerabilities, this project serves as an educational tool, demonstrating the risks associated with unfiltered user input and non-parameterized queries [3], [7]. This project shows how

SQL injection attacks can threaten user data and system security at different levels. It starts with a system that is fully vulnerable, then moves to a moderately secure setup, and finally to a secure setup that uses parameterized queries [1], [6].

The low-security level, which allows unrestricted user input directly embedded in SQL queries, effectively demonstrates how easily attackers can manipulate such vulnerabilities. Using basic SQL injection strings shows how attackers can access unauthorized data, bypass login processes, and potentially view sensitive information [5]. As security gets stronger at medium and high levels, the project shows how sanitization and parameterized queries help prevent attacks [9]. These findings emphasize the importance of using secure query practices and validating input to protect against SQL injection [10].

Furthermore, this project provides insight into the educational benefits of a controlled, demonstrative environment for SQL injection [8]. Users can see how insecure code affects their systems in real-time. This helps them understand why it's important to design with security in mind [4]. Such demonstrations can be highly effective in educating developers on the practical steps required to secure their applications. Future work could extend this project by incorporating additional security vulnerabilities (such as cross-site scripting or CSRF) to create a comprehensive learning platform for web security [3], [6]. In conclusion, the project not only emphasizes the risks posed by SQL injection but also demonstrates clear and practical countermeasures, making it a valuable educational resource in the field of cybersecurity [9].

CONCLUSION

This project demonstrates the vulnerabilities of SQL injection within a controlled web application environment, shedding light on the serious security risks posed by insufficient input validation and non-parameterized SQL queries [7]. Through the implementation of various security levels, it becomes clear that secure coding practices—such as using parameterized queries and rigorous input sanitization—are essential to prevent data breaches and unauthorized access [1], [6]. This project not only emphasizes the need for a security-centric approach to web development but also serves as an educational tool, providing practical insights into identifying and mitigating SQL injection vulnerabilities [4], [8]. In doing so, it contributes to a greater understanding of web application security and the importance of proactive defense measures [10].

FUTURE SCOPE

This project has the potential to evolve into a comprehensive learning platform by incorporating a fully developed website with a range of security levels for SQL injection. Future iterations could include interactive options to toggle between security settings, allowing users to experience and understand different SQL injection mitigation techniques firsthand [6], [8]. By providing a scalable range from low to high security, with each level implementing progressively more advanced defenses—such as parameterized queries, prepared statements, and custom input validation—this platform would serve as a valuable tool for upcoming cybersecurity professionals [9]. Such a project could foster hands-on learning, enabling users to observe how various SQL injection attacks are neutralized as security improves, ultimately bridging the gap between theoretical knowledge and practical cybersecurity skills [5].

REFERENCES

[1]. MeiJunjin, An approach for SQL injection vulnerability detection, IEEE Sixth International Conference on Information Technology: New Generations, pages: 1411 -1414,2009. https://ieeexplore.ieee.org/abstract/document/5070824

[2]. Lijiu Zhang, Qing Gu, Shushen Peng, Xiang Chen, Haigang Zhao, Daoxu Chen, "D-WAV: A Web Application Vulnerabilities Detection Tool Using Characteristics of Web Forms", IEEE Fifth International Conference on Software Engineering Advances, pages: 501-507, 2010 . https://ieeexplore.ieee.org/abstract/document/5615484

[3]. Stephen Thomas, Laurie Williams, Tao Xie, On automated prepared statement generation to remove SQL injection vulnerabilities, Journal of Information and Software Technology, Elsevier Ltd, 2009, pages: 589-598. https://www.sciencedirect.com/science/article/abs/pii/S0950584908001110

[4]. Inyong Lee, Soonki Jeong, Sangsoo Yeo, Jongsub Moon, A novel method for SQL injection attack detection based on removing SQL query attribute values, Journal of Mathematical and Computer Modeling, Elsevier Ltd, 2011, pages:1-11. https://www.sciencedirect.com/science/article/pii/S0895717711000689

[5]. João Antunes, Nuno Neves, Miguel Correia, Paulo Verissimo, and Rui Neves, Vulnerability Discovery with Attack Injection, IEEE Transactions on Software Engineering, 2010, Vol. 36, pages: 357-370.

https://ieeexplore.ieee.org/abstract/document/5374427

[6]. Abdul Bashah Mat Ali, Ala, Yaseen Ibrahim Shakhatrehb, Mohd Syazwan Abdullahc, Jasem Alostadd, SQL-injection vulnerability scanning tool for automatic creation of SQL-injection attacks, Journal of Procedia Computer Science, Elsevier Ltd, 2010, pages: 453-458. https://www.sciencedirect.com/science/article/pii/S1877050910004515

[7]. W.G.J. Halfond, A. Orso, P. Manolios, WASP: protecting web applications using positive tainting and syntax-aware evaluation, IEEE Transactions on Software Engineering, 2008, vol. 34 (1), pages: 65-81. https://ieeexplore.ieee.org/abstract/document/4359474

[8]. Z. Su, G. Wassermann, The essence of command injection attacks in web applications, 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, SC, USA, 2006, pages: 372-382. https://dl.acm.org/doi/abs/10.1145/1111320.1111070

[9]. J. Park, B. Noh, SQL injection attack detection: profiling of web application parameter using the sequence pairwise alignment, Journal of Information Security Applications, LNCS, 2007, vol. 4298, pages: 74-82. https://link.springer.com/chapter/10.1007/978-3-540-71093-6_6

[10]. Ivano Alessandro Elia, José Fonseca, Marco Vieira, Comparing SQL Injection Detection Tools Using Attack Injection: An Experimental Study, 21st IEEE International Symposium on Software Reliability Engineering, 2010, pages:289-298. https://ieeexplore.ieee.org/abstract/document/5635053