IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

A Structured Framework for Detection and Mitigation of SQL Injection Vulnerabilities in Web Applications

[1] Dr. Amit Hariyani

[1] Assistant Professor

[1] Smt. Chandaben Mohanbhai Patel Institute of Computer Applications,

[1] Charotar University of Science and Technology, Anand, Gujarat, India

Abstract: SQL Injection (SQLi) remains a significant security concern for web applications that interact with relational databases through dynamically generated queries. The exploitation of SQL injection vulnerabilities can enable unauthorized access to application data and compromise the confidentiality and integrity of database resources. This paper presents a structured SQL Injection Detection and Mitigation Framework designed to identify and reduce SQL injection vulnerabilities in web applications. The proposed framework follows a multi-stage approach comprising vulnerability detection, analysis of identified weaknesses, categorization of SQL injection attacks, and filtering of inputs associated with SQL injection attempts before they reach the database layer. The framework considers characteristics of potentially malicious inputs, including operators, special characters, logical constructs, and abnormal query patterns, to support the identification and filtering process. A web-based application was developed to validate the framework against SQL injection and malformed query inputs. The validation demonstrated that the protected application rejected the tested malicious inputs while permitting legitimate authentication requests. In addition, the framework was evaluated by researchers and industry professionals as part of the validation process. The study demonstrates the applicability of a structured, multi-stage approach for strengthening web application protection against SQL injection vulnerabilities. The proposed framework can serve as a systematic basis for integrating SQL injection detection and filtering mechanisms into the security lifecycle of database-driven web applications.

Index Terms - SQL Injection, SQL Injection Detection, Web Application Security, Database Security, Vulnerability Detection, Attack Classification, Input Filtering

I. INTRODUCTION

The rapid adoption of web-based applications has made databases an essential component of modern information systems. Web applications are widely used to support services such as online transactions, information management, communication, and business operations. In a typical web-based environment, user requests are processed by an application server and may subsequently result in database queries. This interaction between user-controlled input, application logic, and database operations creates an important security boundary that must be adequately protected. A typical web application architecture comprises the user, firewall, web server, database server, and database, and highlights that application-layer vulnerabilities may remain exploitable even when network-level security mechanisms are in place [1].

Among the threats affecting database-driven web applications, SQL Injection (SQLi) is particularly significant because malicious input can influence the structure or behavior of an SQL statement. Beyond unauthorized data retrieval, successful SQL injection can compromise the confidentiality, integrity, and availability of information stored in the underlying database [2, 3]. SQL injection is an application-level attack

that highlights how conventional network security mechanisms may not adequately address vulnerabilities arising from application requests and database interactions.

The complexity of SQL injection is increased by the diversity of attack techniques. SQL injection attacks can be characterized by various characteristics, including classical and blind attacks, as well as attacks targeting database operations. Common techniques include tautology-based attacks, inference attacks, union queries, piggy-backed queries, code injection, functional calls, and other database-oriented techniques [4]. This diversity indicates that protection mechanisms based on a single detection rule or a narrowly defined attack pattern may not provide adequate coverage across different forms of SQL injection.

Several approaches have been investigated to mitigate SQL injection. Existing techniques include techniques based on server-level protection, taint analysis, query analysis, and other input- or syntax-oriented mechanisms [5, 6]. For example, earlier studies include approaches that examine user input before it reaches application processing, as well as techniques that track potentially tainted data and SQL operators. These approaches demonstrate the importance of examining application input and query characteristics as part of a broader security strategy.

However, despite the availability of various countermeasures, there remains a continuing concern regarding the systematic integration of SQL injection detection and prevention techniques [7]. Existing countermeasures may address particular attack categories or application conditions without necessarily providing a unified procedure for identifying vulnerabilities, analyzing their characteristics, determining the associated SQL injection category, and preventing malicious input from progressing toward the database. These limitations motivate the need for a more structured approach to analyzing and mitigating SQL injection vulnerabilities[8].

To address this issue, this study presents a structured SQL Injection Detection and Mitigation Framework for database-driven web applications. The framework organizes the protection process into multiple stages. Initially, potential vulnerabilities are identified. The identified conditions are then analyzed to determine whether they constitute SQL injection vulnerabilities. The detected vulnerabilities are then categorized by attack characteristics, followed by a filtering stage to prevent abnormal or malicious SQL-related input from being forwarded to the database. The framework therefore attempts to combine detection, analysis, classification, and filtering within a single security workflow.

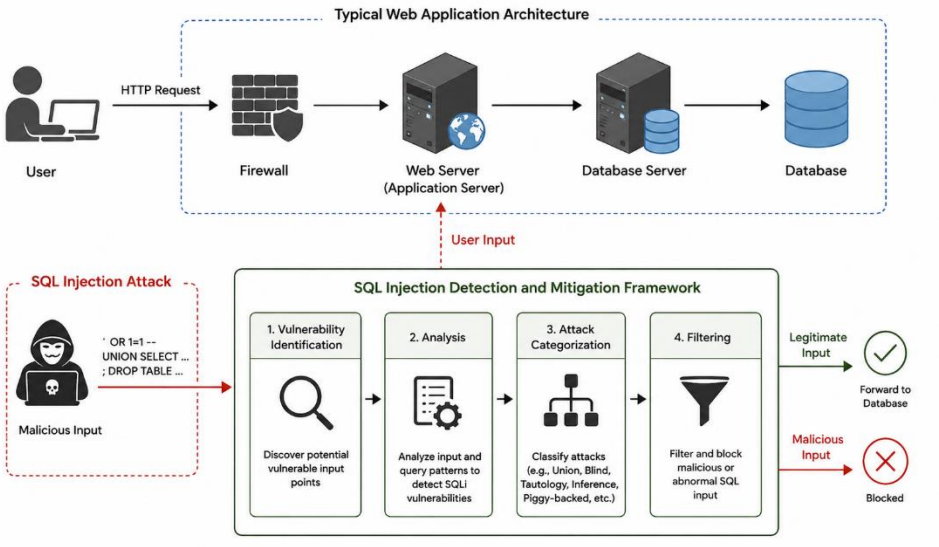

Fig. 1. Overview of SQL Injection Threats and the Proposed Framework.

An important component of the proposed approach is the examination of characteristics associated with potentially malicious input. The framework considers programming-language and query-structure characteristics and identifies elements such as special characters, operators, and logical constructs during vulnerability analysis. A filtering mechanism is subsequently positioned between the web application and the database processing to prevent inputs identified as abnormal or as SQL injection attempts from reaching the database layer.

The framework was also subjected to application-level validation. A web application was developed in accordance with the proposed security approach, and various SQL injection, malformed, and abnormal query inputs were used during testing. According to the reported validation, the protected login application rejected the tested SQL injection inputs while accepting legitimate authentication requests. In addition to application testing, the framework was provided to researchers and industry professionals for further assessment.

The objective of this paper is therefore to present the proposed framework, describe its multi-stage detection and mitigation process, and evaluate its applicability for protecting web applications against SQL injection vulnerabilities. Unlike a general survey of SQL injection attacks, the study focuses on integrating vulnerability detection, attack categorization, and input filtering into a structured security process. The work also provides a basis for further empirical evaluation of the effectiveness, limitations, and practical applicability of structured SQL injection mitigation mechanisms.

The remainder of this paper is organized as follows. Section 2 reviews existing research on SQL injection attacks and their prevention techniques. Section 3 describes the proposed SQL Injection Detection and Mitigation Framework. Section 4 presents the methodology and implementation of the framework. Section 5 discusses the experimental validation and results. Section 6 presents the discussion and limitations of the proposed approach, while Section 7 concludes the study and identifies directions for future research.

II. Related Work

SQL Injection (SQLi) has been investigated extensively from different perspectives, including vulnerability detection, static and dynamic analysis, taint tracking, query analysis, input validation, and runtime prevention. Existing research demonstrates that SQL injection cannot be addressed effectively with a single security mechanism, as attacks may exploit diverse application behaviors and database query structures. Existing research similarly highlights limitations in the systematic combination of existing countermeasures and motivates the development of structured mitigation frameworks.

2.1 Input and Server-Level Protection

Input- and server-level protection mechanisms aim to identify and block potentially malicious requests before they are processed by the application or forwarded to the database. Recent research has investigated integrated approaches to SQL injection detection, prioritization, and prevention that consider multiple SQL injection attack vectors rather than relying on a single detection mechanism [8]. Such approaches provide an additional security layer for database-driven web applications and can help reduce the possibility of malicious input reaching the database layer.

Input validation has also been considered an important component of SQL injection prevention. However, existing research distinguishes between approaches that attempt to recognize known malicious patterns and more restrictive validation strategies. In particular, previous studies have reported that blacklist-based filtering can be problematic because attackers may modify their input to avoid known patterns, while legitimate input may also contain characters that are incorrectly classified as malicious.

2.2 Taint-Based Analysis

Taint analysis represents another approach to identifying potentially dangerous data flows within applications. Nguyen-Tuong and colleagues investigated precise taint tracking to determine whether data reaching SQL statements can be considered untrusted. Their approach tracks tainted values according to programming-language semantics and examines SQL operators, identifiers, and keywords to determine whether potentially unsafe data has entered a query [9].

2.3 Query Analysis and Syntax-Aware Techniques

Another group of SQL injection prevention approaches focuses on the structure and semantics of SQL queries. Recent research has investigated query- and syntax-aware techniques that analyze SQL statement characteristics to distinguish legitimate queries from potentially malicious inputs [10, 11]. Recent SQL injection prevention research has explored mechanisms that analyze query characteristics and structural patterns to identify potentially malicious SQL statements.

2.4 Dynamic and Static SQL Considerations

The way applications construct SQL queries is also closely related to SQL injection risk. SQL query construction can be broadly distinguished between dynamically constructed queries, in which query syntax and values may be assembled at runtime, and parameterized queries, in which the query structure is defined separately from application-supplied values [13].

Dynamic construction of SQL queries can increase the possibility that user-controlled data will influence query structure. In contrast, approaches based on fixed query structures, including prepared statements and

stored procedures, restrict the extent to which supplied values can modify the SQL statement. Query construction is therefore an important consideration when examining and mitigating SQL injection vulnerabilities.

2.5 SQL Injection Classification

Classification of SQL injection attacks is another important aspect of existing research, as different attack categories may require distinct detection or prevention mechanisms. SQL injection attacks can be classified according to characteristics such as the order in which malicious input is executed and the manner in which the database is manipulated. It identifies first-order attacks, in which the result is obtained directly, and second-order attacks, in which malicious input is stored and later activated [14]. Previous research has also investigated several SQL injection attack forms, including tautology-based attacks, inference attacks, UNION-based queries, piggy-backed queries, code injection, and function-based injection [15].

2.6 Limitations of Existing Approaches

Existing research demonstrates that SQL injection can be addressed through multiple layers, including server-level protection, taint analysis, query analysis, input validation, query construction controls, and attack classification. However, these approaches differ considerably in their assumptions, implementation requirements, and coverage of attack scenarios. Existing research indicates that prevention techniques do not necessarily provide uniform protection against all SQL injection attack methods, as different approaches address different attack characteristics and application conditions.

2.7 Research Gap

Based on the existing literature, the principal research gap this study addresses is the lack of a unified, structured process that connects vulnerability identification, vulnerability analysis, SQL injection classification, and filtering within a common framework. Existing techniques generally emphasize one or more specific aspects of SQL injection protection. The proposed approach instead organizes these activities sequentially. The proposed SQLID framework first identifies potential vulnerabilities, analyzes their characteristics, categorizes detected SQL injection conditions, and subsequently applies filtering between the web application and database layers.

III. PROPOSED SQL INJECTION DETECTION AND MITIGATION FRAMEWORK

3.1 Overview of the Proposed Framework

The proposed SQL Injection Detection (SQLID) Framework is designed to provide a structured mechanism to identify and mitigate SQL injection vulnerabilities in database-driven web applications. The framework is based on the observation that SQL injection protection should not depend exclusively on a single filtering rule or detection technique. Instead, the proposed approach separates the security process into a sequence of stages: potential vulnerabilities are first identified, then analyzed, categorized by their SQL injection characteristics, and finally filtered before potentially harmful queries reach the database server.

The framework therefore establishes a security path between the web application and the database layer. Its principal objective is to prevent inputs that exhibit characteristics associated with SQL injection from being processed as legitimate database queries. The proposed framework consists of four principal stages: (1) vulnerability detection, (2) vulnerability analysis, (3) SQL injection attack categorization, and (4) filtering of inputs associated with SQL injection vulnerabilities.

The overall processing sequence of the proposed framework can be expressed as Web Request → Vulnerability Detection → Vulnerability Analysis → SQL Injection (SQLi) Categorization → Filtering → Database Server. This sequential structure is designed to provide progressively stronger examination of incoming web requests at different stages of the security process. Requests identified as potentially suspicious at an earlier stage undergo further analysis and characterization before a final decision is made. Thus, the framework avoids immediately classifying an initially suspicious request as a confirmed SQL injection attempt and instead applies a staged evaluation process to distinguish potentially malicious inputs from legitimate requests before they reach the database server.

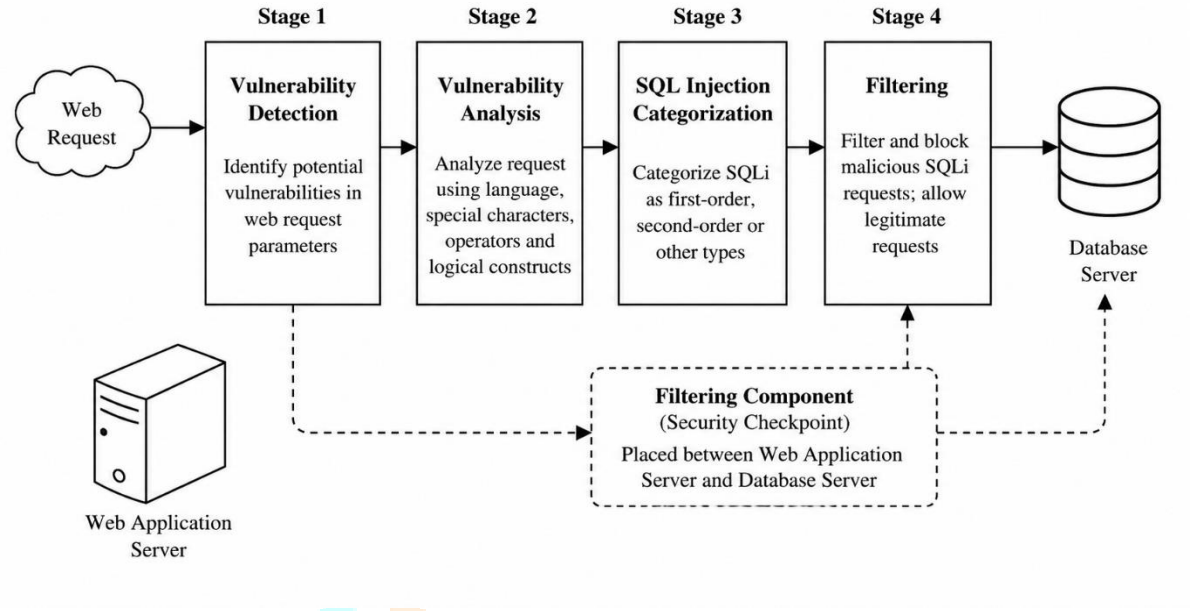

Fig. 2. Overall processing workflow of the proposed SQLID framework.

3.2 Stage 1: Detection of Potential Vulnerabilities

The first stage of the SQLID Framework focuses on identifying potential vulnerabilities within the target web application. In the proposed framework, vulnerability detection is supported by an automated mechanism that examines application URLs and their parameters. The framework's automation component parses the supplied URL and modifies parameters to simulate SQL-related input. If the resulting application response contains an error generated by the underlying database management system, the response is treated as an indication that the corresponding application parameter may require further investigation.

This stage is intended to identify potential entry points for SQL injection. Rather than immediately classifying every abnormal response as an attack, the framework forwards the identified condition to the subsequent analysis stage.

The framework also incorporates an authentication mode that remains enabled by default. This move is intended to provide additional protection for the targeted application and to assist in determining whether the detected condition constitutes undesirable activity.

Thus, the output of Stage 1 is a set of potentially vulnerable application parameters or request conditions.

3.3 Stage 2: Analysis of Existing Vulnerabilities

The second stage examines the vulnerabilities identified during the detection process. The purpose of this stage is to determine whether a detected vulnerability is actually associated with SQL injection.

The framework considers the language and structure characteristics associated with the potentially vulnerable request. In particular, the framework considers the following characteristics:

- Programming/query language used
- Special characters
- Operators
- Logical constructs

The framework specifically identifies characters and constructs such as special characters, arithmetic operators, and logical elements including OR, NOT, and LIKE as characteristics that may be examined during the analysis.

The purpose of this stage is therefore to distinguish a general application vulnerability from one that could be exploited through SQL injection. This distinction is important because not every malformed or abnormal request necessarily constitutes an SQL injection attack.

The analysis stage consequently produces a more refined set of candidate vulnerabilities associated with SQL injection behavior.

3.4 Stage 3: Categorization of SQL Injection Attacks

Once a potential SQL injection condition has been identified and analyzed, the third stage categorizes the detected SQL injection according to its characteristics.

Attack categorization provides additional information about the nature of the detected vulnerability and helps developers and security analysts understand the type of attack involved. Attack classification is an important part of the framework because knowledge of the attack category can help developers and security analysts understand the nature of the vulnerability and select appropriate defensive measures.

The proposed classification considers different forms of SQL injection, including first-order and second-order attacks and several query-oriented attack techniques. First-order attacks produce their effects during the immediate interaction with the application, whereas second-order attacks involve malicious input being stored and subsequently activated during a later operation.

The categorization stage therefore serves two purposes:

1. Characterization of the detected SQL injection condition, and
2. Selection of an appropriate subsequent mitigation response.

This stage transforms the framework from a simple input-filtering mechanism into a structured detection and analysis process.

3.5 Stage 4: Filtering of SQL Injection Vulnerabilities

The fourth stage provides the framework's principal prevention mechanism. A filtering component is positioned between the web application server and the database server. Its purpose is to prevent abnormal, malformed, or potentially malicious SQL injection queries from being forwarded to the database.

For example, when user-provided authentication information results in a request exhibiting characteristics associated with SQL injection, the filter prevents the corresponding query from being transmitted to the database. Instead, the application returns an unsuccessful authentication response to the user.

The filtering stage examines combinations of input characteristics, including operators and special or other SQL-related characters. The proposed framework uses the filter to identify abnormal combinations before database processing occurs.

The filter's position is significant because it establishes a security checkpoint between application processing and database execution. Consequently, an input identified as vulnerable to SQL injection is prevented from reaching the database layer.

3.6 Framework Processing Sequence

The four stages operate as a sequential security workflow.

Stage 1 — Detection: Potentially vulnerable parameters or application conditions are identified.

Stage 2 — Analysis: The identified conditions are examined using characteristics such as the language, query structure, special characters, operators, and logical constructs.

Stage 3 — Categorization: Conditions identified as SQL injection vulnerabilities are classified according to their attack characteristics.

Stage 4 — Filtering: The identified SQL injection-related requests are intercepted before they are forwarded to the database server.

The resulting conceptual workflow represents the central processing sequence of the proposed framework. The proposed framework places the filtering mechanism between the web application server and the database server.

3.7 Additional Security Measures

The framework also recommends supplementary security mechanisms to provide defense-in-depth. These include the principle of least privilege and input validation. The framework recommends restricting the privileges assigned to application database accounts so that a successful SQL injection does not automatically provide excessive database permissions.

Input validation is also identified as an additional defensive layer. Importantly, previous research has identified weaknesses in simple blacklist-based validation: attackers may modify malicious input to bypass predefined patterns, while legitimate values containing filtered characters may be incorrectly rejected.

This observation is particularly important for the research paper because it means that the proposed filter should not be presented as a universally sufficient defense against all SQL injection attacks. Instead, it should be evaluated as one component of a layered security architecture.

3.8 Summary of the Proposed Framework

The SQLID Framework provides a sequential approach to SQL injection mitigation by combining four activities: vulnerability detection, vulnerability analysis, attack categorization, and filtering. The first three stages progressively refine the identification of potentially malicious requests, while the fourth stage prevents identified SQL injection-related requests from reaching the database layer.

Consequently, the framework provides the foundation for the experimental evaluation presented in the following section. The evaluation should determine not only whether known SQL injection inputs are blocked, but also whether legitimate inputs continue to function correctly and what processing overhead or false-positive behavior the proposed mechanism may introduce.

IV. Experimental Methodology and Framework Validation

The proposed SQL Injection Detection and Mitigation Framework was validated through application-level testing. The objective of the validation was to determine whether the framework could identify and prevent SQL injection-related inputs while allowing legitimate authentication requests to be processed normally. The experimental study involved the development of web pages according to the proposed framework and prevention mechanisms and the testing of a login page using SQL injection, malformed, abnormal, and other invalid SQL queries.

4.1 Experimental Objective

The primary objective of the experimental evaluation was to examine the behavior of a web application protected by the SQLID Framework when it received potentially malicious SQL-related input.

The validation was designed to examine two basic conditions:

- Whether SQL injection or malformed SQL inputs could be accepted by the application.
- Whether legitimate username-password combinations continued to be accepted.

This distinction is important because an effective security mechanism should not only reject malicious requests but should also preserve the normal functionality of legitimate users.

4.2 Test Application

A web-based login application was developed in accordance with the principles of the proposed framework. The login interface was selected as the experimental application because authentication forms commonly receive user-controlled input that is subsequently used in database queries.

The experimental request flow was structured as follows:

User Input → Web Application → SQLID Detection/Analysis → SQL Injection Categorization → Filtering → Database

The SQLID framework places its filtering mechanism between the web application server and the database server. Inputs identified as abnormal or corresponding to SQL injection conditions are prevented from being forwarded to the database.

4.3 Test Input Categories

The validation considered multiple forms of invalid SQL-related input. The application was tested for SQL injection, malformed queries, and other abnormal SQL queries.

The test inputs were organized into the following categories:

Table 4.3: Test Input Categories and Their Purposes

Test Category	Purpose
Legitimate credentials	Verify normal application functionality
Malformed SQL input	Examine handling of syntactically abnormal requests
SQL injection input	Determine whether injection attempts are blocked
Abnormal SQL input	Evaluate filtering of unexpected query-related input
SQL-related special-character combinations	Examine the filtering mechanism
Logical/operator-based input	Examine potentially malicious SQL constructs

The framework specifically considers special characters, operators, and logical constructs during its vulnerability-analysis stage.

4.4 Experimental Procedure

The experiment follows the sequence below.

Step 1 — Submit user input: A username and password are entered into the login interface.
Step 2 — Vulnerability detection: The framework examines the request and identifies potentially vulnerable parameters. The proposed framework uses an automated mechanism to parse URL parameters and modify them to test for SQL-related behavior.
Step 3 — Vulnerability analysis: Potentially vulnerable input is analyzed using characteristics associated with SQL query construction, including language and structural characteristics such as special characters, operators, and logical constructs.
Step 4 — Attack categorization: If the condition indicates SQL injection, it is categorized based on the detected attack's characteristics. Attack categorization provides information about the nature and characteristics of the detected SQL injection.
Step 5 — Filtering: The filtering mechanism examines the request before it reaches the database. An input identified as corresponding to SQL injection is prevented from being forwarded to the database server.
Step 6 — Application response: A malicious or abnormal request is rejected, whereas a legitimate request proceeds to normal database processing.

4.5 Validation Criteria

The framework was evaluated using the following functional criteria:

- **Attack rejection:** SQL injection inputs should not result in successful authentication or unauthorized database processing.
- **Legitimate request acceptance:** Valid credentials should continue to provide normal application access.
- **Database protection:** Requests identified as SQL injection should not be forwarded to the database.
- **Input discrimination:** The mechanism should distinguish potentially malicious inputs from legitimate authentication data.

The experimental results showed that the tested login application rejected the SQL injection inputs and granted access only when valid username and password credentials were supplied.

4.6 Validation by Researchers and Industry Experts

In addition to application-level testing, the framework was also subjected to an external assessment. The experimental results showed that the framework was provided to researchers and industry experts for evaluation. The feedback from the evaluators was incorporated into the overall framework evaluation.

This provides a second validation perspective beyond the author's own application testing. The external evaluation considered the perceived applicability and practical relevance of the proposed framework. The number of evaluators, evaluation criteria, scoring procedure, and statistical analysis should be reported where such data are available. We should not invent these values.

4.7 Experimental Outcome

The application-level experimental results indicate that the protected login page rejected the tested SQL injection, malformed, and abnormal SQL inputs while accepting valid authentication credentials.

These results provide initial evidence that the proposed framework can prevent the tested SQL injection attempts from being processed as legitimate authentication requests.

The experimental evaluation should therefore consider measurable performance indicators, including attack-rejection rate, legitimate-request acceptance rate, false-positive rate, and processing overhead, where applicable.

V. Results and Analysis

The validation of the proposed SQL Injection Detection and Mitigation Framework was performed at two levels: application-level testing and assessment by researchers and industry professionals. The application-level evaluation examined the behavior of the protected login application when legitimate and SQL injection-related inputs were supplied, while the second evaluation considered external reviewers' assessment of the framework.

5.1 Application-Level Validation

The experimental application was a web-based authentication page. The validation involved submitting legitimate credentials as well as SQL injection, malformed, bad, and abnormal SQL inputs. The experimental results show that the login application accepted valid credentials while rejecting the tested SQL injection inputs.

The screenshots obtained during the application-level validation provide visual evidence of this behavior. A legitimate login attempt is shown producing a successful validation response, whereas SQL injection-related inputs result in an invalid-data response.

The observed behavior can be summarized as follows:

Table 5.1: Observed Application Behavior for Test Inputs

Input Type	Observed Application Behavior	Interpretation
Genuine username and password	Login permitted	Legitimate request accepted
SQL injection input	Login rejected	Injection attempt prevented
Malformed SQL input	Login rejected	Abnormal request prevented
Abnormal SQL input	Login rejected	Request did not proceed to normal authentication

The framework's filtering mechanism is intended to prevent SQL injection-related requests from reaching the database server. The framework examines combinations of operators, special characters, and other SQL-related characteristics before forwarding the request.

5.2 SQL Injection Test Behavior

The framework was tested against SQL injection patterns that attempt to alter the logical behavior of an authentication query. The evaluation includes tautology-based inputs and other SQL injection patterns representing different attack characteristics.

The significance of this test is that a vulnerable authentication query can potentially be transformed when attacker-controlled input modifies the original WHERE condition. The proposed framework aims to identify such input characteristics before the database processes the request.

The filtering mechanism also considers SQL comments and concatenation constructs. The proposed framework considers single-line comments, multi-line comments, and concatenation operators as additional characteristics that the filtering mechanism may examine.

5.3 Legitimate Request Handling

Security mechanisms must balance protection and usability. A filtering mechanism that rejects every input containing a special character may incorrectly block legitimate users. The proposed framework therefore distinguishes between individual characters and combinations that may constitute SQL injection patterns. A single operator or special character does not necessarily constitute an SQL injection attempt; therefore, the proposed filtering mechanism evaluates combinations of input characteristics and permits legitimate individual inputs while rejecting suspicious combinations.

The application-level validation showed that legitimate username-password combinations were accepted, while the tested SQL injection inputs were rejected. This observation provides preliminary evidence that the proposed mechanism can maintain basic authentication functionality while blocking the tested malicious inputs.

5.4 Internal Operation of the Authentication Protection Mechanism

The framework also describes an authentication-specific protection component referred to as SQL Injection Protector for Authentication (SQLIPA). The architecture consists of a user login interface, an SQL injection protection component, and a user account table.

The described mechanism generates hash values for username and password information when an account is created and subsequently compares the supplied credentials with the stored values during authentication. Access is granted when the values match; otherwise, access is denied.

This component complements the SQL injection filtering mechanism by providing an additional authentication-level control.

5.5 Assessment by Researchers and Industry Professionals

The second part of the validation involved external assessment of the proposed framework. The proposed framework was provided to researchers and industry professionals for external evaluation, and feedback on the assessment was collected.

The assessment form evaluated several characteristics of the framework, including:

- Relevance of the framework
- Importance of the framework
- Completeness of the proposed points
- Accuracy of the automation tool
- Clarity of the presented concepts
- Organization and layout
- User-friendliness
- Overall recommendation

The assessment responses indicate that the reviewers generally considered the framework relevant and rated its characteristics as primarily “Excellent” or “Good.” The reviewers also provided an overall recommendation of “Accept.”

The written comments in the available assessment records were generally positive. One reviewer described the framework as providing security for databases and web applications; another indicated that it could be used with different programming languages; and another reported testing the framework's steps and observing a high level of security.

5.6 Consolidated Findings

Based on the available experimental evidence, the following observations can be made:

1. The protected login application rejected the tested SQL injection-related inputs.
2. Legitimate authentication credentials were accepted.
3. The filtering stage acted as an intermediate security checkpoint before database processing.
4. The framework considered multiple SQL-related characteristics, including special characters, operators, logical constructs, and comment syntax.
5. External reviewers assessed the framework positively and recommended acceptance in the available assessment records.

However, the present evaluation does not provide a sufficiently comprehensive dataset of numerical attacks to compute accuracy, precision, recall, false-positive rate, false-negative rate, or processing overhead. Therefore, these metrics should not be reported without conducting additional systematic experiments.

5.7 Discussion of Results

The application-level results indicate that the proposed framework can prevent the tested SQL injection inputs from being accepted by the authentication application. The result is consistent with the framework's intended operation, in which potentially malicious inputs are analyzed and filtered before reaching the database layer.

The external assessment provides additional qualitative support for the framework. However, expert feedback should be considered supplementary validation rather than a substitute for quantitative security testing.

Further evaluation should use a larger, systematically categorized attack dataset. Such an evaluation should include different SQL injection categories, legitimate inputs, modified payloads, and measurements of false positives, false negatives, detection rate, blocking rate, and processing overhead.

This limitation is particularly relevant because previous research has shown that blacklist-style filtering can be bypassed when attackers modify or obfuscate malicious patterns, and that it may also result in legitimate inputs being incorrectly rejected.

VI. Acknowledgment

The author would like to express sincere gratitude to all those who contributed to the successful completion of this research. Special thanks are extended to the researchers and industry professionals who evaluated the proposed framework and provided valuable feedback during its validation.

The author also acknowledges the support and guidance received from the academic and research community throughout the development and evaluation of the proposed SQL Injection Detection and Mitigation Framework.

Finally, the author is grateful to the institution, colleagues, and family members for their continuous encouragement, support, and motivation throughout this research.

References

[1] A. H. Alhowiti and A. M. Awadelkarim, "A New Database Integrity Protection Approach Against SQL Injection Attacks (SQLIAs)," in *Proc. 2025 4th IEEE Int. Conf. on Computing and Information Technology (ICCIT)*, 2025, doi: 10.1109/ICCIT63348.2025.10989381.

[2] D. Muduli, S. Shookdeb, A. T. Zamani, S. Saxena, A. S. Kanade, N. Parveen, and M. Shameem, "SIDNet: A SQL Injection Detection Network for Enhancing Cybersecurity," *IEEE Access*, vol. 12, pp. 176511–176526, 2024, doi: 10.1109/ACCESS.2024.3502293.

[3] A. Hariyani and P. Dolia, "Comprehensive review of advanced techniques for mitigating SQL injection vulnerabilities in modern applications," *International Journal of Innovative Science and Research Technology*, vol. 10, no. 3, pp. 3063–3070, Mar. 2025, doi: 10.38124/ijisrt/25mar1982.

[4] A. Coscia, V. Dentamaro, S. Galantucci, A. Maci, G. Pirlo, *et al.*, "PROGESI: A PROxy Grammar to Enhance Web Application Firewall for SQL Injection Prevention," *IEEE Access*, 2024, doi: 10.1109/ACCESS.2024.3438092.

[5] G. Floris, C. Scano, B. Montaruli, L. Demetrio, A. Valenza, L. Compagna, D. Ariu, L. Piras, D. Balzarotti, and B. Biggio, "ModSec-AdvLearn: Countering Adversarial SQL Injections With Robust Machine Learning," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 6693–6705, 2025, doi: 10.1109/TIFS.2025.3583234.

[6] R. F. de Oliveira, E. C. Vilas Boas, G. P. Aquino, and F. A. P. de Figueiredo, "A Real-Time Machine Learning-Assisted SQL Injection Detection for Web Applications," in *Proc. 2025 IEEE International Conference on Computing (ICOCO)*, 2025, pp. 219–223, doi: 10.1109/ICOCO67189.2025.11334146.

[7] A. Hariyani and P. Dolia, "A Cryptography-Enforced SQL Query Integrity Framework for Complete SQL Injection Prevention," *International Journal of Scientific Research in Engineering and Management (IJSREM)*, vol. 10, no. 03, Mar. 2026, doi: 10.55041/IJSREM58457.

[8] A. Paul, V. Sharma, and O. Olukoya, "SQL injection attack: Detection, prioritization & prevention," *Journal of Information Security and Applications*, vol. 85, Art. no. 103871, Sep. 2024, doi: 10.1016/j.jisa.2024.103871.

[9] M. Al-Shaaby *et al*., "An Enhanced Static Taint Analysis Approach to Detect Input Validation Vulnerability," *Journal of King Saud University - Computer and Information Sciences*, vol. 35, no. 2, pp. 682–701, 2023, doi: 10.1016/j.jksuci.2023.01.009.

[10] Y. Chen, G. Liang, and Q. Wang, "Research on SQL Injection Detection Technology Based on Content Matching and Deep Learning," *Computers, Materials & Continua*, vol. 84, no. 1, pp. 1145–1167, 2025, doi: 10.32604/cmc.2025.063319.

[11] Y. Zhang *et al*., "Telescope: A Learned What-If Call for Column Store Selection in HTAP Databases," in *Proc. 2026 IEEE 42nd International Conference on Data Engineering (ICDE)*, Montreal, QC, Canada, 2026, pp. 2641–2653, doi: 10.1109/ICDE65706.2026.00197.

[12] A. Hariyani and P. Dolia, "An innovative method for detecting SQLi attacks by altering SQL query attribute values," *International Journal of Advanced Computer Research*, vol. 14, no. 68, pp. 89–96, 2024, doi: 10.19101/IJACR.2024.1466005.

[13] J. K. Chowdhury, D. K. Yadav, and P. V. S. S. R. C. Mouli, "Query comparison engine to detect and prevent SQL injection attacks," *International Journal of System Assurance Engineering and Management*, vol. 17, pp. 1908–1920, 2026, doi: 10.1007/s13198-026-03352-3.

[14] A. Hariyani and P. Dolia, "CryptoSQLShield: A comprehensive study on cryptography-assisted methods for SQL injection defense," *International Journal of Engineering Research & Technology (IJERT)*, vol. 15, no. 1, Jan. 2026, doi: 10.17577/IJERTV15IS010090.

[15] K. S. Fathi, S. Barakat, and A. Rezk, "An effective SQL injection detection model using LSTM for imbalanced datasets," *Computers & Security*, vol. 153, Art. no. 104391, 2025, doi: 10.1016/j.cose.2025.104391.