**A black background with a black square

Description automatically generated with medium confidence Comprehensive Security Framework For Detecting and Prevention Of Different Injection Attacks**

**Mini-Project Synopsis**

***Submitted to***

Manipal School of Information Sciences, MAHE, Manipal

****

|  |  |  |
| --- | --- | --- |
| **Reg. Number** | **Name** | **Branch** |
| **241059029** | **SIDDHARTH HEBBAR B.P** | **CYBERSECURITY** |
| **241059015** | **ANUJNA K** | **CYBERSECURITY** |
| **251059039** | **MADHUSUDAN REDDY MS** | **CYBERSECURITY** |

**Under The Guidance Of**

Mrs.Keerthana.B

Manipal School Of Information Science

MAHE, MANIPAL

# ABSTRACT

In today's digital landscape, web applications are increasingly targeted by attackers exploiting various vulnerabilities, particularly injection attacks. This project focuses on understanding and analyzing three common types of injection attacks: SQL Injection, Code Injection, and HTML Header Injection.

The study begins with an in-depth exploration of SQL Injection, a technique where attackers manipulate SQL queries to gain unauthorized access to database information. Next, the project delves into Code Injection, where attackers introduce malicious code into a program, causing unintended behaviors or compromising the system. Finally, the research covers HTML HeaderInjection, which involves injecting unauthorized content into HTML headers, leading to potential information disclosure or session hijacking.

Through the examination of these attacks, the project aims to highlight their mechanisms, real-world implications, and the critical importance of secure coding practices. Additionally, it proposes effective mitigation strategies and best practices to prevent such vulnerabilities, thereby enhancing the overall security posture of web applications. This comprehensive analysis serves as a valuable resource for developers, cybersecurity professionals, and organizations to understand and protect against these prevalent threats.

# TABLE OF CONTENT

[ABSTRACT ii](#_Toc188701461)

[TABLE OF CONTENT iii](#_Toc188701462)

[INTRODUCTION 1](#_Toc188701463)

[SQL Injection: 1](#_Toc188701464)

[Code Injection: 1](#_Toc188701465)

[HTTP Header Injection: 1](#_Toc188701466)

[OBJECTIVE 2](#_Toc188701467)

[LITERARTURE SURVEY 3](#_Toc188701468)

[LITERATURE SURVEY 1. 3](#_Toc188701469)

[LITERATURE SURVEY 2. 5](#_Toc188701470)

[LITERATURE SURVEY 3. 7](#_Toc188701471)

[LITERATURE SURVEY 4. 9](#_Toc188701472)

[LITERATURE SURVEY 5. 13](#_Toc188701473)

[LITERATURE SURVEY 6: 17](#_Toc188701474)

[LITERATURE SURVEY 7: 21](#_Toc188701475)

[FLOW CHART 24](#_Toc188701476)

[SOFTWARE REQUIRMENTS 25](#_Toc188701477)

[APPLICATIONS 27](#_Toc188701478)

[REFERENCES 28](#_Toc188701479)

# INTRODUCTION

Injection attacks are a significant threat to web application security, allowing attackers to manipulate input data to execute unintended commands or access unauthorized data. Among the various types of injection attacks, SQL Injection, Code Injection, and HTML Header Injection are particularly noteworthy due to their unique attack vectors and impacts.

## SQL Injection:

SQL Injection targets the database layer of a web application by inserting malicious SQL queries into input fields, enabling attackers to manipulate the backend database. This can lead to unauthorized access to sensitive information, data modification, deletion, or even execution of administrative operations. The attack exploits vulnerabilities in how the application constructs and executes SQL queries, often due to poor input validation, improper sanitization, or the lack of prepared statements. This makes SQL Injection one of the most common and severe security risks in web applications.

## Code Injection:

Code Injection occurs when an attacker inserts arbitrary code into an application, which is subsequently executed by the system. This can result in various harmful outcomes, including data corruption, unauthorized access, or even full system compromise. Unlike SQL Injection, which targets database queries, Code Injection directly exploits the execution of code within the application, making it a serious threat to system integrity and security.

## HTTP Header Injection:

HTTP Header Injection is a web vulnerability where attackers inject malicious input into HTTP headers due to improper input validation. HTTP headers are key-value pairs that manage communication between clients and servers. By inserting special characters like \r and \n, attackers can manipulate the structure of headers, leading to exploits such as HTTP response splitting, cross-site scripting (XSS), session hijacking, and open redirects. Common causes include directly using unsanitized user inputs in headers. Preventive measures include input validation, encoding special characters, avoiding dynamic header creation, and leveraging secure frameworks. Properly hardening HTTP headers can significantly reduce the risk of this attack.

# OBJECTIVE

* Gain a basic understanding of different types of injection attacks, such as SQL Injection, Code Injection, and HTML Header Injection, by researching their mechanics, methods of exploitation, and potential impact on web applications.
* Structured planning on how to develop a tool that will analyze and sanitize user input, ensuring that it is free from malicious patterns before it reaches the backend system, thereby securing the application from possible threats.
* To create a system that can identify malicious input by recognizing attack patterns associated with SQL Injection, Code Injection, and HTTP Header Injection, and flag them for immediate action.
* To design a tool to block harmful input in real-time, preventing it from being processed by the application and protecting the system from unauthorized access, data manipulation, or other forms of attack.
* To integrate the tool into existing web applications in a way that doesn’t compromise performance.

# LITERARTURE SURVEY

## LITERATURE SURVEY 1.

**Title:** “Defining Code-Injection Attacks”

**Objective:**

The paper aims to address shortcomings in the traditional definitions of code-injection attacks (e.g., SQL injection) that lead to detection issues such as false positives (flagging benign inputs as attacks) and false negatives (missing actual attacks). By introducing a novel definition based on the concept of “normal-form values,” the paper strives to provide a more precise understanding of code-injection vulnerabilities, improving the accuracy of detection mechanisms and the evaluation of mitigation strategies.

**Methodology:**

**Analysis of Existing Definitions**

* The authors identify flaws in conventional definitions of code-injection attacks, which often focus on changes to the syntactic structure of application outputs. These definitions fail to capture certain injection scenarios and mislabel benign operations as malicious.

**Proposal of a New Definition**

* The paper introduces a new definition based on whether injected symbols in the application’s output are used as normal-form values. Normal-form values are fully evaluated terms that cannot function as executable code when injected.
* This approach abstracts from specific programming languages and environments, making it applicable to various contexts such as SQL injection, LDAP injection, and XSS attacks.

**Evaluation Through Examples**

* Scenarios of successful and unsuccessful code-injection attacks are analyzed to demonstrate how the new definition improves detection accuracy.
* A theoretical model, named "SQL◦," is introduced as a simplified SQL-like language for testing the applicability of the new definition.

**Use Cases and Limitations**

* The new definition is tested for its ability to classify benign vs. malicious inputs across a range of use cases.
* Limitations of static and black-box analysis methods for detecting injection attacks are discussed, emphasizing the necessity of runtime monitoring

**Challenges and Observations**

**Challenges in Conventional Definitions**

* Conventional definitions fail to identify certain attacks, such as injecting functions like exit(), and misidentify benign inputs as attacks.
* Definitions tied to syntactic changes struggle with cases where malicious inputs produce no visible structural alterations in the output program.

**Implementational Hurdles**

* The proposed definition requires runtime monitoring and precise tracking of tainted inputs, which can be computationally expensive and complex to implement.

**Findings**

* The proposed definition improves the understanding of when code-injection attacks occur by focusing on how symbols are used rather than merely on syntactic changes.
* It enables more accurate detection of attacks and reduces false positives/negatives compared to existing mechanisms.
* White-box, runtime-monitoring mechanisms are necessary for precise detection, as static and black-box approaches fail to capture all cases.

**Future Scope**

* Development of **runtime monitoring systems** based on the new definition for real-world deployment.
* Exploration of automated tools to integrate the new definition into secure coding practices.
* Extending the definition to accommodate emerging injection techniques, such as those targeting multi-language and hybrid applications.
* Investigating the integration of this approach into machine learning models for automated threat detection and classification.

## LITERATURE SURVEY 2.

**Title:** “A Review of Defense Against Slow HTTP Attack”

#### **Objective**

The paper explores the nature and mechanisms of Slow HTTP attacks, a specific form of Denial of Service (DoS) attack targeting the application layer (Layer 7) of the network stack. The objective is to review existing defense mechanisms to counteract this attack, which exploits vulnerabilities in the HTTP protocol by keeping server resources occupied with incomplete or extremely slow requests. The study aims to identify effective configurations and preventive techniques for various web servers, including Apache, Nginx, and IIS, to enhance their resilience against such threats.

#### **Methodology**

**Understanding the Slow HTTP Attack**

* Exploits the design of the HTTP protocol, where servers wait for complete requests before processing.
* Types of Slow HTTP attacks discussed include:
* **Slow Headers**: Exploiting HTTP headers to send incomplete requests, exhausting server resources.
* **Slow Body**: Sending POST bodies very slowly to keep server connections open.
* **Slow Read**: Slowing down the reading of received data, forcing servers to maintain idle connections.

**Detailed Study of Defense Mechanisms**

* **General Preventive Measures** : Use of tools like xinetd to control connection rates and resource limits on Linux-based web servers.
* **Specific Server Configurations** **Nginx**: Adjusting buffer size, timeout limits, and maximum simultaneous connections. **Apache**: Utilizing built-in modules like mod\_reqtimeout, mod\_security, and mod\_antiloris to limit request handling times and connection counts. **IIS**: Leveraging features such as WebLimits and RequestLimits to control connection timeouts, buffer sizes, and query string lengths.

**Evaluation of Existing Literature**

* The paper reviews academic studies and real-world configurations, analyzing the efficacy of detection systems and mitigation strategies.
* Detection approaches include analyzing abnormal patterns in HTTP traffic and deploying anomaly-based detection systems.

**Implementation Examples**

* Demonstrates how tools like netstat and iptables can be used to identify and block malicious IPs conducting Slow HTTP attacks.
* Configuration examples for Nginx and Apache provide practical guidelines for defense.

#### **Challenges and Observations**

* **Challenges in Detection**

Slow HTTP attacks are harder to trace compared to bandwidth-heavy DoS attacks because they consume minimal network bandwidth. Attackers often use HTTPS to mask their origin, complicating mitigation efforts.

* **Resource Management Issues**

Vulnerable servers struggle with resource allocation under Slow HTTP attacks, often reaching their maximum connection limits and denying access to legitimate users.

#### **Findings**

* Slow HTTP attacks exploit server-side vulnerabilities, making proper configuration essential to mitigating their effects.
* A combination of general preventive measures and specific server configurations significantly reduces the risk of Slow HTTP attacks.
* Anomaly-based detection systems, which monitor deviations in traffic patterns, are particularly effective for early detection and prevention.

#### **Future Scope**

* Development of **adaptive defense mechanisms** that can dynamically respond to emerging Slow HTTP attack patterns.
* Integrating machine learning and AI techniques to analyze traffic patterns and predict potential attack scenarios in real-time.
* Exploring the use of blockchain for decentralized and secure resource allocation in web servers.
* Further research into hybrid defense strategies that combine detection, prevention, and rapid response for mitigating Slow HTTP attacks across multiple platforms.

## LITERATURE SURVEY 3.

**Title:** “A Taxonomy of SQL Injection Attacks”

#### **Objective**

The paper aims to present a detailed taxonomy of SQL Injection Attacks (SQLIAs), which exploit vulnerabilities in database-driven web applications. It provides an in-depth classification of attack types, along with practical examples to illustrate each type. By understanding the mechanics of these attacks, developers and security professionals can adopt robust security measures during application development. The ultimate goal is to mitigate the risk of SQLIAs by fostering awareness and recommending preventive strategies.

#### **Methodology**

The paper categorizes SQL injection attacks based on their underlying mechanisms and objectives, using examples and theoretical models to demonstrate vulnerabilities and countermeasures. The methodology includes:

**Classification of SQLIAs**

* **Tautologies**: Exploits logical conditions to make SQL statements always true, bypassing authentication mechanisms. Example: SELECT \* FROM USERS WHERE USERNAME = 'Admin' OR '1' = '1'.
* **Illegal/Logically Incorrect Queries**: Injects incorrect SQL statements to retrieve database schema details via error messages. Example: Breaking a query with malformed syntax to expose column names.
* **Union Queries**: Combines legitimate queries with malicious ones to extract data from other tables. Example: UNION SELECT CreditCardNumber FROM CreditCardTable.
* **Piggy-Backed Queries:** Adds an independent malicious query after a legitimate one using delimiters like ;. Example: Injecting DROP TABLE USERS; to delete user data.
* **Stored Procedures**: Exploits poorly secured stored procedures to execute malicious commands. Example: Injecting commands like SHUTDOWN into stored procedure variables.
* **Inference-Based Attacks**: Includes Timing and Blind SQL injection attacks that infer information without visible error messages. Example: Using delays (WAITFOR) to identify database configurations.
* **Alternate Encodings:** Uses obfuscation techniques like Hex, Base64, or comments to bypass intrusion detection systems (IDS). Example: Encoding OR 1=1 as %4F%52%2031%3D31.

**Illustration Through Examples**

* Each attack type is accompanied by detailed SQL statements and use cases to demonstrate how the attack manipulates queries and extracts data.
* Tools like error messages, timing delays, and obfuscated payloads are analyzed for their role in exploiting vulnerabilities.

**Recommendations for Secure Coding**

* Emphasizes the importance of parameterized queries and prepared statements to prevent SQL injection.
* Discusses input validation and output encoding as critical measures for reducing attack surfaces.

**Evaluation and Comparison**

* Explores real-world scenarios to highlight how different attack types affect web application security.
* Compares the severity and impact of each attack type, providing insights into their practical implications.

#### **Challenges and Observations**

* **Dynamic Query Vulnerabilities**: Applications using dynamic queries without proper input validation are particularly susceptible to SQLIAs.
* **Error Handling:** Inadequate error handling practices reveal database structures and configurations, aiding attackers in launching sophisticated attacks.
* **Evasion Techniques:** Advanced encoding methods make it difficult for intrusion detection systems to identify malicious payloads.

#### **Findings**

* The taxonomy serves as a comprehensive guide for understanding SQL injection techniques and their potential impact on web applications.
* Robust input validation, parameterized queries, and secure coding practices are essential to prevent SQLIAs.
* Security measures like disabling error messages and implementing least privilege principles significantly reduce attack risks.

#### **Future Scope**

* **Automated Detection Tools**: Developing AI-based tools to detect and prevent SQLIAs in real time.
* **Dynamic Security Models:** Exploring adaptive security frameworks that respond to evolving attack methods.
* **Comprehensive Training**: Enhancing developer training programs to include advanced SQL injection techniques and mitigation strategies.
* **Extending the Taxonomy**: Incorporating emerging attack methods, such as hybrid injection techniques targeting multi-cloud environments.

## LITERATURE SURVEY 4.

**Title:** “Analysis of SQL Injection Attack Detection and Prevention on MySQL Database Using Input Categorization and Input Verifier”

#### **Objective**

The primary objective of this study is to propose and evaluate a comprehensive framework for detecting and preventing SQL Injection Attacks (SQLIAs) in MySQL databases. The framework focuses on two key components: Input Categorization and Input Verifier mechanisms. By combining these techniques, the study aims to enhance database security by identifying malicious inputs and preventing unauthorized database manipulation. The research emphasizes real-world applicability by testing the framework under various attack scenarios and configurations.

#### **Methodology**

**Analysis of SQL Injection Mechanisms**

* The study begins by identifying common SQL injection techniques, such as tautology-based attacks, union-based attacks, and error-based attacks.
* Real-world SQL injection scenarios are analyzed to understand the methods attackers use to exploit vulnerabilities in web applications and databases.

**Proposed Framework**

* **Input Categorization**

Input data is categorized into predefined groups based on its structure and intended use.

Categories include alphanumeric inputs, numerical inputs, and special characters. Each category is assigned specific validation rules to ensure compliance with expected patterns.

* **Input Verifier**

A verification mechanism is implemented to check the integrity of input data before it is executed in SQL queries.

The verifier uses a whitelist approach, allowing only validated inputs to interact with the database.

**Implementation**

* The proposed framework is implemented in a MySQL database environment. A testbed is set up with sample web applications vulnerable to SQL injection.
* The Input Categorization module is integrated into the application’s input-handling logic, while the Input Verifier operates as an intermediary between the application and the database.

**Testing Scenarios**

* The framework is tested against multiple SQL injection scenarios, including tautology-based attacks, union-based attacks, and time-based blind SQL injections.
* Test cases include variations in input length, encoding methods, and attack complexity to evaluate the framework’s robustness.

**Performance Metrics**

* The effectiveness of the framework is measured using metrics such as detection accuracy, false positive rate, and processing overhead.
* Comparative analysis is conducted with existing SQL injection prevention techniques to highlight the advantages of the proposed approach.

#### **Challenges and Observations**

**Challenges in Input Validation**

* Implementing comprehensive input validation rules for diverse application scenarios requires significant effort and customization.
* Handling complex input patterns, such as those involving nested queries or encoded payloads, poses additional challenges.

**Observations**

* The Input Categorization module effectively reduces the attack surface by enforcing strict validation rules for each input category.
* The Input Verifier adds an additional layer of security by ensuring that only pre-validated inputs reach the database, minimizing the risk of SQL injection.
* Performance overhead introduced by the framework is minimal, making it suitable for real-time applications.

#### **Findings**

**Enhanced Detection Accuracy**

* The proposed framework achieves high detection accuracy across various SQL injection scenarios, outperforming traditional prevention techniques.
* The combination of Input Categorization and Input Verifier ensures comprehensive protection against known and emerging SQL injection methods.
* **Reduction in False Positives**

By using a whitelist-based approach, the framework significantly reduces false positives, allowing legitimate inputs to pass through without disruption.

* **Scalability and Flexibility**

The modular design of the framework allows it to be easily adapted to different application environments and database systems.

The Input Categorization module can be customized to accommodate specific application requirements.

#### **Future Scope**

**Integration with Machine Learning**

* Incorporating machine learning models into the Input Categorization module could enhance the framework’s ability to detect complex and evolving SQL injection patterns.
* Machine learning algorithms can analyze historical attack data to predict and prevent future SQL injection attempts.
* **Extension to Multi-Database Environments**

Future research could focus on extending the framework to support multiple database systems, including NoSQL databases, to address a broader range of application scenarios.

* **Real-Time Monitoring and Alerts**

Implementing real-time monitoring and alerting mechanisms could enhance the framework’s ability to respond to SQL injection attempts as they occur.

Integration with Security Information and Event Management (SIEM) systems could provide centralized monitoring and incident response capabilities.

* **Developer Training and Adoption**

Educating developers about the importance of input validation and secure coding practices is critical for the widespread adoption of the proposed framework.

Future efforts should include developing training programs and resources to help developers integrate the framework into their applications.

## LITERATURE SURVEY 5.

**Title:** “Injecting Comments to Detect JavaScript Code Injection Attacks”

#### **Objective**

This paper aims to address the challenges posed by JavaScript Code Injection Attacks, a subset of Cross-Site Scripting (XSS) vulnerabilities, which exploit client-side scripts to execute malicious actions in users’ browsers. The study’s primary goal is to propose a server-side detection mechanism that distinguishes injected JavaScript code from legitimate code through the use of comment statements containing unique tokens and feature-based policies. This approach provides a layer of defense at the server level to mitigate XSS vulnerabilities, enhance detection accuracy, and reduce reliance on browser-side protections. The proposed method emphasizes efficiency, practicality, and compatibility with existing web technologies without requiring browser modifications or client-side scripting changes.

#### **Methodology**

**Analysis of JavaScript Code Injection Mechanisms**

* **Types of Attacks Studied**

**Method Call Injection**: Attacks involve injecting calls to pre-defined methods in order to execute malicious actions.

**Method Definition Overriding**: Attackers modify existing method definitions to alter their intended behavior, such as switching the functionality of banUser and activateUser methods.

* Real-world examples of XSS vulnerabilities were examined to identify common patterns and attack techniques, such as injecting arbitrary JavaScript or overriding host object methods.

**Proposed Framework**

**Comment Injection Mechanism**

* Legitimate JavaScript code blocks are pre- and post-pended with comment statements containing unique random tokens. These tokens serve as identifiers for valid code.
* Features of legitimate code, such as method names and parameter structures, are encoded within the comments.

**Policy-Based Validation**

* Features extracted from legitimate JavaScript code are stored in XML policy files. During runtime, the server-side filter compares code features in response pages with these policies to detect injected JavaScript.
* The validation process involves checking for the presence of valid comments, detecting duplicate or missing tokens, and verifying code features against policy specifications.

**Implementation Details**

* **Prototype Tool** A prototype tool was developed in Java to automate comment injection and policy generation. This tool uses the Jericho HTML parser to modify JavaScript code in server-side JSP programs.
* **Runtime Filter** The server-side filter intercepts all outgoing HTTP responses, evaluates JavaScript code blocks, and removes any detected injected code before delivering the page to the client.
* **Feature Extraction** The tool utilizes the Rhino JavaScript parser to identify method definitions, calls, and argument structures, which are then encoded into policy files for runtime validation.
* **Evaluation Methodology** The framework was tested on three open-source JSP programs (“JVote,” “JAuction,” and “JInsure”), each with varying levels of JavaScript complexity and vulnerabilities.

Multiple attack scenarios were simulated, including:

Arbitrary method calls (e.g., alert('xss')).

Legitimate method call injection.

Method definition overriding.

Injection of duplicate or incorrect comments.

The effectiveness of the approach was measured based on detection rates, false positives, and runtime overhead.

#### **Challenges and Observations**

**Challenges**

* **Handling Legitimate Code Similarity**: Malicious code resembling legitimate code (e.g., legitimate method calls or overridden definitions) posed challenges in distinguishing benign and injected content.
* **Complex Code Structures**: Nested method calls and anonymous functions required additional preprocessing to extract and validate features.
* **Scalability**: While the approach was effective for small-scale programs, its applicability to larger web applications with extensive JavaScript usage needs further optimization.

**Observations**

* The proposed method achieved a zero false-negative rate across all tested scenarios, demonstrating its robustness against various JavaScript injection techniques.
* Runtime overhead was minimal, as the framework avoided executing JavaScript code during validation, focusing solely on structural analysis.
* The approach effectively mitigated XSS vulnerabilities even in programs with weak input validation, making it suitable for retrofitting into existing systems.

#### **Findings**

**Effectiveness of Comment Injection**

* Random tokens and feature-based policies provided a reliable mechanism for distinguishing legitimate JavaScript from injected code.
* The method detected a wide range of injection techniques, including arbitrary and legitimate method calls, as well as definition overriding.

**Comparison with Related Work**

* Unlike existing server-side approaches that rely on browser modifications or static analysis, this framework integrates seamlessly into server environments without altering client-side functionality.
* It addresses limitations of previous methods, such as failure to detect injected code resembling benign scripts or code originating from trusted third-party sources.

#### **Future Scope**

**Enhancing Automation**

* Developing automated preprocessing tools to handle complex JavaScript structures, such as nested functions and event handlers, would improve the framework’s scalability and usability.

**Integration with Browser Security**

* While the current approach focuses on server-side detection, integrating complementary client-side techniques, such as CSP (Content Security Policy), could provide a multi-layered defense.

**Evaluating Runtime Overhead**

* Further studies on the performance impact of the framework on large-scale web applications are necessary to ensure its practical deployment in enterprise environments.

**Generalization for Other Scripting Languages**

* Extending the approach to detect injection attacks in other client-side scripting languages, such as TypeScript or frameworks like React and Angular, would broaden its applicability.

**Developer Training**

* Educational resources and training programs for developers can promote the adoption of comment injection techniques and secure coding practices.

## LITERATURE SURVEY 6:

**Title:** “SQL Injection Countermeasures Methods”

#### **Objective**

This paper aims to address the challenges posed by SQL injection attacks, one of the most prevalent threats to web application security. These attacks exploit vulnerabilities in user input handling to execute unauthorized SQL queries, potentially compromising data confidentiality, integrity, and availability. The study's primary goal is to evaluate and propose effective countermeasures to prevent SQL injection attacks. It emphasizes techniques such as prepared statements, SQLrand, taint analysis, and runtime monitoring to mitigate SQL injection vulnerabilities. The proposed methods focus on practicality, scalability, and compatibility with existing database systems, providing a multi-layered defense strategy to enhance overall application security.

#### **Methodology**

##### **Analysis of SQL Injection Mechanisms**

**Types of Attacks Studied**:

* **Tautologies**: Exploiting conditional clauses that always evaluate as true. Example: SELECT \* FROM users WHERE username = 'admin' OR 1=1 --';
* **Piggyback Queries**: Appending malicious queries to legitimate ones. Example: SELECT \* FROM users; DROP TABLE users; --
* **Union Queries**: Combining malicious queries with legitimate ones to extract additional data. Example: SELECT username FROM users UNION SELECT password FROM admin;
* **Stored Procedure Exploits**: Leveraging vulnerabilities in stored procedures to execute malicious commands.

##### **Proposed Framework**

**Prepared Statements and Parameterized Queries**:

* Enforces strong separation between data and code by using placeholders for input variables.
* Example in PHP:
* $stmt = $db->prepare("SELECT \* FROM users WHERE username = ? AND password = ?");
* $stmt->execute([$username, $password]);

**SQLrand**:

* Implements instruction-set randomization by appending random tokens to queries, making injection attacks fail.

Example: SELECT user\_name FROM users#12345 WHERE id#12345 = '1';

**Taint Analysis:**

* Tracks and restricts untrusted user inputs by marking and monitoring data flow.

**Escape Functions**:

* Sanitizes user inputs by escaping dangerous characters (e.g., single quotes).
* Example: Using mysqli\_real\_escape\_string() in MySQL.

##### **Implementation Details**

* **Prototype Tool**:

A prototype tool was developed to implement prepared statements, SQLrand, and taint analysis. This tool integrates with database management systems to automate query sanitization and monitoring.

* **Runtime Filter**:

A server-side filter intercepts all incoming SQL queries, evaluates them for potential injection patterns, and blocks malicious queries before execution.

* **Feature Extraction**:

Features such as query structure, parameters, and reserved keywords are analyzed to identify and block injection attempts.

##### **Evaluation Methodology**

The framework was tested on several web applications with varying levels of SQL query complexity and vulnerabilities. Multiple attack scenarios were simulated, including:

* Tautology-based injections.
* Piggyback and union queries.
* Stored procedure exploits.

Effectiveness was measured based on:

* Detection rates.
* False positives.
* Performance overhead during runtime.

#### **Challenges and Observations**

##### **Challenges**

**Handling Complex Queries**:

* Nested subqueries and dynamically generated SQL statements posed challenges for prepared statements and taint analysis.

**Scalability**:

* High-traffic applications experienced performance degradation when using computationally intensive methods like taint analysis.

**Similar Code Patterns**:

* Malicious queries closely resembling legitimate queries were harder to distinguish.

##### **Observations**

* **Effectiveness of Prepared Statements**:

Achieved a near-zero false-negative rate across all tested scenarios.

* **Runtime Overhead**:

Prepared statements and SQLrand introduced minimal overhead, making them suitable for real-time applications.

* **Combination Approach**:

Layering multiple countermeasures significantly improved detection rates and reduced false positives.

#### **Findings**

##### Effectiveness of Countermeasures

* Prepared statements and parameterized queries provided robust protection against all tested attack scenarios.
* SQLrand effectively mitigated tautologies, piggyback queries, and union-based attacks but was less effective against stored procedure exploits.
* Taint analysis demonstrated high accuracy but required optimization for large-scale applications.

##### Comparison with Related Work

* Unlike static analysis or browser-based solutions, the proposed framework integrates seamlessly into server environments and does not require client-side modifications.
* Addressed limitations of previous methods, such as their inability to detect advanced SQL injection techniques or handle dynamically generated queries.

#### **Future Scope**

##### Enhancing Automation

* Develop automated tools to analyze and sanitize dynamically generated SQL queries in real-time.

##### Integration with Application Frameworks

* Integrate prepared statements and taint analysis into popular frameworks like Django, Laravel, and Spring Boot to simplify adoption.

##### Performance Optimization

* Investigate lightweight methods to reduce runtime overhead for high-traffic applications.

##### Expanding Scope

* Adapt the framework to secure NoSQL databases and other query languages like GraphQL and SPARQL.

##### Developer Training

* Create training modules and best practice guides to promote secure coding practices among developers.

## LITERATURE SURVEY 7:

**Title:"Fatal injection: a survey of modern code injection attack countermeasures"**

**Objectives**

The paper aims to tackle the persistent and damaging issue of **code injection attacks (CIAs)**, which exploit vulnerabilities in software systems to execute malicious code. The key objectives are:

* **Categorization of CIAs**: Analyze different types of code injection attacks, such as SQL injection, buffer overflows, and JavaScript-based attacks, and describe how they exploit flaws in applications.
* **Examination of Countermeasures:** Evaluate the mechanisms available to address these vulnerabilities:

**Static Analysis Countermeasures**: Methods to detect and eliminate vulnerabilities during the software development lifecycle.

**Dynamic Detection Countermeasures**: Techniques to safeguard systems at runtime against ongoing attacks.

**Critical Analysis**: Assess these countermeasures based on:

* **Flexibility**: Their ability to adapt to various attack types.
* **Effectiveness:** How well they minimize false positives and false negatives.
* **Implementation Independence**: Whether they are dependent on programming languages or specific system configurations.
* **Computational Overhead:** Their impact on system performance.
* **Guidance for Practitioners and Researchers**: Provide a framework to help choose appropriate countermeasures and develop new, more robust solutions.

**Methodology**

**Analysis of Code Injection Attacks (CIAs):**

* **Definition:** CIAs involve introducing malicious code by exploiting flaws in applications, such as unchecked inputs or programming assumptions.
* **Categorization:**

**Binary Code Injection Attacks**: Exploits in low-level system code, such as buffer overflows and format string vulnerabilities.

**Source Code Injection Attacks**: Includes attacks using domain-specific languages (e.g., SQL and XML) and dynamic languages (e.g., JavaScript and PHP).

* **Evaluation of Countermeasures:** The countermeasures are divided into two main categories:

**Static Analysis:**

Techniques: Pattern matching, lexical analysis, data-flow analysis, model checking, symbolic execution, and type system extensions.

Objective: Detect vulnerabilities in code before deployment.

**Dynamic Detection:**

Techniques: Runtime tainting, instruction set randomization, policy enforcement, and whitelisting.

Objective: Identify and block attacks during system operation.

**Comparative Analysis**:

* **Flexibility**: Whether the approach can handle various attack types.
* **Effectiveness**: Validation through testing for false positives and false negatives.
* **Independence**: How much the method relies on specific languages or environments.
* **Performance Overhead**: Evaluation of runtime or computational resource costs.
* **Data Compilation and Visualization**: Tables summarize the analysis of tools and mechanisms in terms of flexibility, effectiveness, independence, and overhead.

**Conclusion**

The paper provides several key conclusions:

**Diverse Countermeasures**: Both static and dynamic methods are essential, as each addresses different aspects of CIAs:

* Static analysis is effective in identifying vulnerabilities before deployment but cannot detect runtime threats.
* Dynamic detection safeguards systems in production but may incur computational overhead.

**Challenges in Countermeasures**:

* **False Positives and False Negatives**: Many tools struggle to balance accurate detection without overwhelming users with irrelevant warnings or missing attacks.
* **Implementation Complexity**: Some countermeasures require significant changes to existing systems or applications, limiting their practicality.

**Importance of Flexibility**: Countermeasures must adapt to evolving attack vectors and target environments, as no one-size-fits-all solution exists.

**Need for Rigorous Testing**: Many proposed mechanisms lack thorough evaluation under real-world conditions, reducing confidence in their effectiveness.

**Future Scope**

The paper highlights several areas for future research and development to enhance the field of code injection countermeasures:

**Integration of Static and Dynamic Approaches**:

* Develop frameworks that combine the strengths of static analysis (proactive detection) with dynamic detection (real-time protection).

**Focus on Emerging Attack Vectors**:

* Adapt countermeasures for new technologies, such as NoSQL databases, modern JavaScript frameworks, and cloud-native architectures.

**Minimizing False Positives/Negatives**:

* Research advanced algorithms, such as machine learning-based approaches, to improve the precision of detection mechanisms.

**Scalability and Performance Optimization**:

* Address computational overhead challenges to make countermeasures viable for large-scale, high-performance systems.

**Automation in Security Processes**:

* Build tools that automate the identification and remediation of vulnerabilities, reducing the burden on developers and security teams.

**Focus on Real-World Applications**:

* Test countermeasures in realistic scenarios, including complex enterprise systems and distributed environments, to ensure their practical effectiveness.

# FLOW CHART

A diagram of a software process

Description automatically generated

# SOFTWARE REQUIRMENTS

1. **Programming Language:**

* Python or Java for backend processing and middleware logic (depending on integration needs with the application).
* JavaScript for frontend validation and user input sanitization if applicable.

1. **Web Framework:**

* Flask (Python) or Spring Boot (Java) for building the middleware as a service that integrates easily with web applications.

1. **Database:**

* A relational database for logging detected injection attempts and maintaining a record of sanitized input data.
* Maybe consider the use of NoSQL database for flexible storage of logs and patterns.

1. **Security Libraries:**

* OWASP Dependency-Check for dependency analysis and checking for known vulnerabilities in third-party libraries.
* OWASP ESAPI for input validation and escaping functions to prevent injection attacks.

1. **Regular Expressions Library:**

* Python re module or Java Regular Expressions to create patterns for detecting known injection signatures in user inputs.

1. **Logging and Monitoring Tools:**

* Log4j (Java) or Python’s logging module to log suspicious input and attack attempts for further review.
* Prometheus and Grafana for performance monitoring and alerting on injection detection.

1. **Input Sanitization Libraries:**

* HTMLPurifier or Bleach (for Python) to sanitize HTML input and prevent HTML Header Injection.
* SQLAlchemy or JDBC for secure database interactions and protection against SQL Injection.

1. **Web Application Integration Tools:**

* Apache or NGINX for integrating the middleware layer into existing web applications.
* Docker for containerization of the tool to ensure easy deployment and scalability across different environments.

# APPLICATIONS

**Web Application Security**

The tool can be integrated into web applications to sanitize and validate user inputs, preventing SQL Injection, Code Injection, and HTML Header Injection, thus ensuring the security of the application.

**E-commerce Platforms:**

It can be used in e-commerce websites to protect sensitive customer and transaction data from injection attacks, ensuring secure and reliable transactions.

**Content Management Systems (CMS):**

The tool can safeguard CMS platforms by validating and sanitizing user-generated content, preventing attackers from injecting malicious code that could compromise the website's integrity.

**Healthcare Systems:**

Protects sensitive patient data in electronic health record (EHR) systems by mitigating injection vulnerabilities in forms and APIs, ensuring HIPAA compliance and data integrity.

**Financial Applications:**

Secures banking and financial systems against injection attacks that could lead to unauthorized transactions or sensitive data breaches.

**Educational Platforms:**

Safeguards online learning platforms by sanitizing inputs in discussion boards, forms, and exam systems, preventing malicious attacks that could compromise user data or system functionality

# REFERENCES

[1] Ray, D., & Ligatti, J. (2012). Defining code-injection attacks. *Acm Sigplan Notices*, *47*(1), 179-190.

[2] Suroto, S. (2017). A review of defense against slow HTTP attack. *JOIV: International Journal on Informatics Visualization*, *1*(4), 127-134.

[3] Arif, A. A. S., Purwoko, R., Qomariasih, N., & Setiawan, H. (2022, October). Analysis of SQL injection attack detection and prevention on MySQL database using input categorization and input verifier. In *2022 IEEE 8th Information Technology International Seminar (ITIS)* (pp. 190-194). IEEE.

[4] Shahriar, H., & Zulkernine, M. (2011, July). Injecting comments to detect JavaScript code injection attacks. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops* (pp. 104-109). IEEE.

[5] Alsobhi, H., & Alshareef, R. (2020, September). Sql injection countermeasures methods. In *2020 International Conference on Computing and Information Technology (ICCIT-1441)* (pp. 1-4). IEEE.

[6] Mitropoulos, D., & Spinellis, D. (2017). Fatal injection: A survey of modern code injection attack countermeasures. *PeerJ Computer Science*, *3*, e136.

[7] Sadeghian, A., Zamani, M., & Abdullah, S. M. (2013, September). A taxonomy of SQL injection attacks. In *2013 International Conference on Informatics and Creative Multimedia* (pp. 269-273). IEEE.