Secure Syntax: Boolean-Based Blind SQL Injection

Himani Ravindra Borana*

Luddy School of Informatics, Computing, and Engineering,
Indiana University Bloomington
hborana@iu.edu

***Abstract*— This study rigorously investigates the complex mechanisms and strategies underlying Boolean-based Blind SQL Injection (SQLi) attacks, a sophisticated subtype of SQL injection. These attacks manipulate the logic of SQL queries to covertly extract sensitive data from databases, even without direct access. This research thoroughly examines the attack vectors, assesses the vulnerability of diverse web applications, and develops a comprehensive framework for robust defensive measures. The concept of "Secure Syntax" is introduced as a methodological core to strengthen query integrity and mitigate risks. The effectiveness of these prevention strategies is empirically validated through a series of precise experiments, establishing a solid foundation for enhancing database security against SQLi attacks.**

I. INTRODUCTION

In the contemporary digital landscape, data breaches and security lapses have elevated the importance of robust cybersecurity measures. As organizations continue to digitize their operations, they inevitably increase their vulnerability to cyber attacks. Among these, SQL Injection remains a prevalent technique employed by adversaries to manipulate databases, posing a severe threat to information integrity. This malevolent practice enables unauthorized access to confidential data, potentially leading to financial fraud, identity theft, and a substantial compromise of personal and corporate security.

The gravity of SQL Injection attacks is underscored by their frequent utilization in high-profile breaches, disrupting services and eroding trust in digital systems. Despite years of awareness and defensive efforts, SQL Injection remains alarmingly effective due to its evolution in sophistication and the persistent oversight in secure coding practices.

This paper focuses on a particularly insidious variant of SQL Injection—Boolean-Based Blind SQL Injection. Unlike classic SQL Injection, which retrieves data directly, this variant allows attackers to infer sensitive information from databases indirectly. By manipulating SQL queries with logical operations, attackers deduce the existence, value, or format of hidden data based on the application's response or behavior. This indirect method of attack makes detection and prevention significantly more challenging.

Boolean-Based Blind SQL Injection is not merely a theoretical concern but a practical threat that exploits the subtle vulnerabilities within an application's database interaction. Its stealthy nature allows it to bypass conventional detection mechanisms that search for overt signs of database tampering, making it a preferred tool for attackers seeking to maintain a low profile while conducting reconnaissance or exfiltrating data.

To address this nuanced threat, we propose a novel defense mechanism termed "Secure Syntax." Secure Syntax is designed to preemptively identify and neutralize the patterns indicative of Boolean-Based Blind SQL Injection before they can be exploited. Our approach hinges on a syntactic analysis that fortifies the structural integrity of SQL queries, ensuring that logical operations used in day-to-day database transactions do not become unwitting conduits for data leakage. By embedding this methodology directly into the application's database interaction layer, Secure Syntax aims to serve as an imperceptible shield, preserving the application's functionality while systematically repelling malicious query attempts.

The inception of Secure Syntax stems from a dual understanding of both the attacker's perspective and the systemic vulnerabilities inherent within SQL interfaces. Recognizing that prevention is more effective than remediation, Secure Syntax seeks to revolutionize the traditional reactive paradigms of cybersecurity. Instead of merely deflecting the symptoms of injection attempts, it seeks to restructure the query-processing edifice in such a way that the opportunities for exploitation are vastly diminished. In doing so, Secure Syntax aspires to not only safeguard against existing attack vectors but also to adaptively fortify against the continually evolving landscape of cyber threats.

II. OUTLINE OF THE REAL-WORLD SCENARIO

In standard web application operations, the login process is a fundamental point of user verification and data security. During this process, a user typically enters their username and password into a designated login form. These credentials are then submitted to the server, where an SQL query is constructed to verify the entered information. The typical SQL command executed would be:

```
SELECT * FROM users
WHERE username = 'admin'
AND
password = 'password123';
```

This query checks the database for matching user records. If

*Use the \thanks command to put information here

a record with the corresponding username and password is found, the system grants access; otherwise, access is denied.

However, this login process can be compromised through various types of SQL injections if user inputs are not properly sanitized before being incorporated into SQL queries. SQL injections exploit security vulnerabilities arising from inadequate input validation, allowing attackers to manipulate or corrupt the SQL queries executed by the database server.

A common form of SQL injection on login pages involves the attacker inserting SQL control characters or commands into input fields. For example, inserting ' OR '1'='1 into a username field could modify an authentication query to:

```
SELECT * FROM users
WHERE
username=''
OR '1'='1' AND password='whatever';
```

This modified query effectively bypasses authentication checks by forcing the query condition to always evaluate to true, thus granting unauthorized access to the attacker.

While SQL injections can be performed in various ways, this paper specifically examines Boolean-based Blind SQL Injection—a technique that does not depend on the error messages returned by the database. Instead, it relies on interpreting the application's response to injected SQL queries that return true or false values. In the context of a login page, an attacker might manipulate the SQL query by altering the URL to include parameters that affect the truth value of the SQL statement. For instance, an attacker might change the URL to include a query such as:

```
http://example.com/login?username=admin'
AND 1=1--
```

Here, 1=1 is a universally true statement, and appending it with a comment symbol -- negates the rest of the SQL command, ensuring the query validates only the modified condition. If 1=1 is swapped with 1=0, the query becomes:

```
http://example.com/login?username=admin'
AND 1=0--
```

This would fail to execute properly if the username does not exist, allowing the attacker to deduce whether certain usernames are present in the database based on the application's response (e.g., an error message or redirect behavior).

Beyond confirming user presence, attackers may also execute more sophisticated attacks to deduce passwords. They often employ a trial-and-error technique to guess each character of the password using a SUBSTRING SQL function. An example of this method is:

```
http://example.com/login?username=Eli'
AND SUBSTRING((SELECT Password
FROM User
WHERE user_name = 'Eli'), 1, 1) = 'h'--
```

This query will instruct the server to check if the first character of user Eli's password is 'h'. The attacker sends this query and observes the response from the server. If there's a different outcome, such as a change in the content of the HTTP response or a longer load time, it suggests the character might be correct. If the response is the same, the attacker concludes the character is incorrect and tries the next one. By repeatedly applying this technique, character by character, the attacker can eventually deduce the entire password:

```
SELECT * FROM user
WHERE id = 1
AND SUBSTRING((SELECT Password
FROM User
WHERE user_name = 'Eli'), 1, 1) = 'h';
```

This laborious process, commonly referred to as a dictionary or brute force attack, can be automated by scripts, making it feasible to test thousands of combinations. It underscores the need for countermeasures such as account lockouts after several failed attempts, two-factor authentication, and complex password requirements to mitigate such threats.

Furthermore, the arsenal of SQL injection techniques extends to the strategic employment of CASE statements, offering a clandestine avenue for data inference. Through the execution of a crafted query, an attacker is enabled to observe differential responses from the system, contingent upon the validity of a conditional check. For instance, the query:

```
SELECT * FROM Users
WHERE Username = 'xyz'
AND (SELECT CASE WHEN
        (Username = 'Eli'
        AND SUBSTRING(Password, 1, 1)
        > 'm')
    THEN 1/0 ELSE 'a' END) = 'a';
```

Serves to ascertain the alphabetical sequence of a user's password character. It manipulates the application's response by inducing an error through a division by zero when the condition is true—thereby subtly revealing the password, one character position at a time. Such a method underscores the criticality of incorporating fail-safe input validation and secure coding practices to fortify web applications against these sophisticated SQL injection exploits. In response to these advanced injection tactics, the adoption of Secure Syntax principles provides a robust countermeasure. By parsing and binding SQL statements strictly through parameterized queries and prepared statements, Secure Syntax effectively neutralizes malicious code, rendering such exploitation attempts futile and thereby reinforcing the security posture of database-driven applications.

III. Methodological Approach

The Secure Syntax workflow represents a methodical approach to safeguarding web applications against malicious injections, particularly SQL injection threats. The process initiates with the user interaction at the login interface, where credentials are inputted. This pivotal juncture is safeguarded

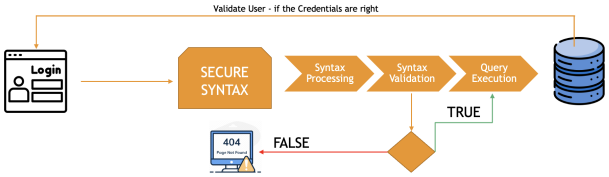

Fig. 1. Workflow of Secure Syntax

by the Secure Syntax system, a defense mechanism specifically designed to scrutinize and sanitize incoming data. The Secure Syntax serves as a preliminary barrier, meticulously inspecting the syntax of user inputs for any anomalous patterns that might signify an injection attempt.

Following this initial line of defense, the workflow transitions into the syntax processing stage. Here, the input undergoes a series of parsing and normalization routines, aligning it with the security protocols and ensuring its readiness for validation. The syntax validation phase acts as a gatekeeper, rigorously examining the processed inputs against a set of stringent security criteria. Inputs failing to meet these standards trigger an immediate diversion within the workflow, leading to a "404 Not Found" response. This method effectively obfuscates the application's feedback to unauthorized users, a critical feature in preventing attackers from gleaning system information from error messages.

For inputs that successfully navigate through the validation checkpoint, the workflow proceeds to the query execution phase. In this phase, secure, parameterized SQL queries are constructed and executed within the confines of the database management system. The database then becomes the final arbiter of authentication, determining the legitimacy of the user credentials.

The architecture culminates with a dual-path response mechanism. Verified credentials segue to the granting of access rights, allowing the user to proceed to the secured areas of the application. Conversely, inputs that fail to authenticate, whether due to incorrect credentials or failed security checks, result in a nondescript "404 Not Found" error, thereby maintaining the security posture and providing no leverage for potential attackers. This comprehensive Secure Syntax workflow is reflective of a robust security-oriented architecture, designed to thwart SQL injection attempts and ensure the integrity of user authentication processes.

IV. Application of Secure Syntax

The Application of Secure Syntax is an integral component of contemporary web application security, designed to mitigate SQL injection risks through a structured three-stage analysis process. Each query received by the system undergoes Lexical Analysis, breaking down the query into a series of tokens to confirm adherence to established lexical conventions. Subsequently, Syntax Analysis is applied, structuring these tokens according to SQL grammar rules to verify the query's format and structure. Finally, Semantic Analysis is conducted to ensure the query's operations are valid within the database's context and semantic rules. This systematic approach ensures rigorous validation of each query, safeguarding the database from potentially malicious interactions.

```
def parseSQL(sqlQuery):
    tokens = tokenize(sqlQuery)
    if not tokens:
         return "Lexical Error:
             Invalid tokens found"

     parseTree = parse(tokens)
    if not parseTree:
        return "Syntax Error:
            Query does not
            match SQL grammar"

    if not validateSemantics(parseTree):
        return "Semantic Error:
        Query semantics invalid"

    return parseTree
```

A. Lexical Analysis

Lexical Analysis is an essential procedure in the parsing of SQL queries, aimed at ensuring security and correctness before execution. This process initiates by scanning the query string from the beginning, methodically progressing character by character. It commences with the omission of any whitespace, which, while important for human readability, holds no significance for the parser and is thus disregarded.

As the scan continues, when the analyzer encounters characters that constitute the commencement of a known SQL keyword, such as **'SELECT'**, **'FROM**, or **'WHERE'**, or symbols like **=**, *****, and **;**, it continues to read until the end of this token is reached. The recognized token is then segregated and categorized appropriately. A particularly intricate aspect of Lexical Analysis is the handling of string literals, which are typically enclosed in quotation marks. The analyzer is designed to detect the opening quotation mark and read through, including any escaped quotes within the string, until the closing quotation mark is found. The entire substring, quotation marks included, is then recorded as a single token.

The method uses a regex pattern to execute lexical analysis, a critical step in the Secure Syntax process for parsing SQL queries. The regex pattern is `[\w']+|[\s]+|[,;()=*]`, which facilitates the segmentation of the query into tokens. Specifically, the pattern `[\w']+` captures both identifiers and keywords, which include alphanumeric sequences, as well as string literals encapsulated in single quotes. Sequences of whitespace are matched by `[\s]+` and serve as separators in the input, aiding in token delineation without becoming tokens themselves. Symbols that are integral to SQL syntax, such as commas, semicolons, parentheses, the equals sign, and the asterisk, are matched by [,;()=*].

The lexical analysis progresses by scanning the query from left to right, applying the regex pattern to slice the string into a sequence of tokens. Each matched sequence is recognized as a distinct token—identifiers, literals, or symbols based on the part of the pattern it matches. Whitespace is used to separate tokens but is not included in the output.

After the application of the regex pattern, the resulting tokens undergo a filtration process to remove any elements that consist solely of whitespace, ensuring that only syntactically meaningful tokens are retained. The output is an array of tokens that represent the parsed SQL query, such as:

```
'['SELECT', 'user_id', 'FROM', 'users',
'WHERE', 'username', '=', "'Eli'",
'AND', 'password', '=', "'1234'"]'
```

This output array is the end product of the lexical analysis stage—each token is now ready to be further validated by the syntax and semantic analysis phases of the Secure Syntax process, which collectively ensure the security and structural integrity of the SQL query before it is executed against the database.

B. Syntax Analysis

Within the Secure Syntax system, Syntax Analysis plays a crucial role in evaluating the SQL queries specifically designed for operations like user authentication on login pages. After the Lexical Analysis phase has tokenized the input, the Syntax Analysis phase takes over, ensuring that each token adheres to SQL's rigid syntactic structure. The process commences by ascertaining the presence of the 'SELECT' keyword, which is imperative for queries that retrieve data.

The array of tokens is meticulously organized into a parse tree, serving as both a visual and structural embodiment of the hierarchical nature of SQL grammar. The 'SELECT' keyword forms the root, and ensuing tokens branch out to represent columns, tables, and conditions—elements that collectively construct the SQL command.

However, should the tokens fail to align with the expected grammatical framework—indicating either a malformed query or a potential injection attempt—the Secure Syntax system will deliberately obfuscate the response. Instead of providing explicit error messages that could aid an attacker, the system generates a generic "404 Not Found" error. This strategic ambiguity serves a dual purpose: it shields the application from revealing any structural information about the database or authentication logic, and it leaves the attacker uncertain as to whether the injected query was syntactically incorrect or if they have reached a non-existent page. This intentional design decision is a cornerstone of Secure Syntax's defensive posture, ensuring that no inadvertent clues are provided that could lead to a security breach.

Each node and leaf on this tree **Fig 2.** is a crucial checkpoint in the query's syntax validation, ensuring that each segment—column selection, table source, and conditional logic—is in its rightful place. This careful structuring is

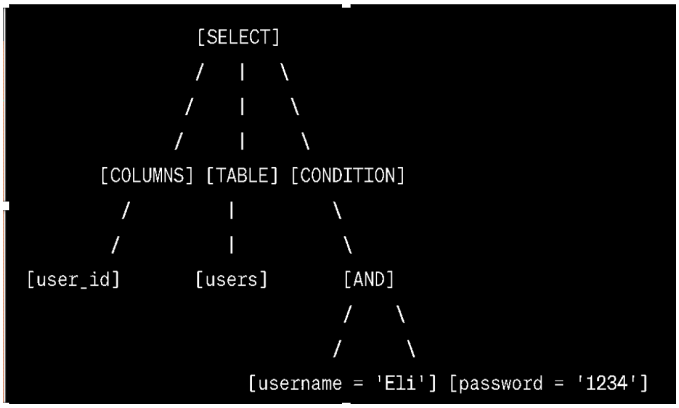

Fig. 2. Syntax Analysis

critical in preventing SQL injection attacks, maintaining the security and integrity of the authentication process on the login page.

C. Semantic Analysis

Semantic Analysis, the concluding phase in the Secure Syntax workflow, takes on the task of verification post-Syntax Analysis. This stage scrutinizes the parse tree to determine the meaningfulness of the query in the context of the database's schema and constraints. It extends beyond the structural correctness of the query to interpret its intended meaning and applicability.

The process involves checking that each token in the parse tree aligns with the database logic. For instance, it ensures that the identifiers like 'user-id' and 'users' refer to actual columns and tables in the database, and that the operations requested—such as retrieving a user with a given username and password—are permitted and rational within the existing database structure.

Moreover, Semantic Analysis confirms that the relationships and dependencies between the tokens are logical. This includes validating that the conditions specified in the 'WHERE' clause make sense for the data types of the corresponding columns and that they adhere to the rules and relationships defined within the database.

If the Semantic Analysis discovers inconsistencies or references to non-existent database entities, the query is rejected to prevent erroneous database operations. In the context of a login page, this means that if the verification of the parse tree against the database schema fails, the system will generate a response that deliberately avoids revealing specific details of the failure. Typically, a generic "404 Not Found" error is returned, obfuscating the outcome and thereby ensuring that no insights are inadvertently provided to an attacker who might be probing for vulnerabilities.

In this capacity, Semantic Analysis is not merely a gatekeeper of structural and syntactic integrity but also a robust guardian of the database's semantic coherence and security.

V. Evaluation and Key Findings

In the evaluation section of our study, we scrutinize the output and implications of the Secure Syntax's semantic

analysis. The process culminates in a hierarchical structure that logically organizes the components of the SQL query, an arrangement that demonstrates compatibility with programmatic requirements and sets the stage for meticulous validation. Specifically, the parseSQL function's efficacy is highlighted by its ability to deconstruct the query into a discernible, secure format, as demonstrated by the output received after semantic analysis:

```
{
      'SELECT': ['user_id'],
      'FROM': ['users'],
      'WHERE': [('username', '=', 'Eli'),
      ('password', '=', '1234')]
}
```

This structured output, while adept for analysis, must undergo reconversion into a SQL string to be processed by the database. The current scope of Secure Syntax does not encompass this conversion, designating it as an area for future development. The function's current iteration validates syntactic structure and semantic content but stops short of reconstituting the validated query into its executable form.

The evaluation asserts that the transition from a parse tree back to a SQL command is crucial. It is not merely a syntactic requirement but a bridge connecting the fortified security checks to the database's operational framework. Thus, the study's findings urge further research to address the conversion from tree-based structure to a string-based SQL command, a transformation that is pivotal for a holistic, secure query processing system.

Consequently, the research sets forth an agenda for future work: the crafting of a function capable of securely and accurately reconstructing a query into its original string format, post-validation. This function would represent the final piece of the Secure Syntax workflow, ensuring that the database executes only queries that have been vetted for structural and logical soundness, thereby fortifying the validation mechanism for database credentials.

A. Key Finding

In the key findings of our research, we underscore the Secure Syntax system's efficacy in mitigating SQL injection threats. The system employs placeholders within SQL queries, a technique that, coupled with parameter binding outside the query text, significantly enhances security. This method ensures that user-supplied input is treated as data, not executable code, effectively neutralizing the risk of injection. By delegating the handling of parameters to the database engine, the potential for malicious SQL execution is thwarted.

Moreover, the Secure Syntax approach streamlines the process of pre-compilation. Prepared statements with placeholders allow the SQL statement to be compiled in advance, creating an execution plan that is reused with different parameters. This not only improves performance by reducing the need for the database to compile queries on the fly but also closes the window of opportunity for injection, as the structure of the SQL statement is fixed and unalterable by user input.

These findings position the Secure Syntax system as a robust defense mechanism, combining security with efficiency, and they serve as a foundation for future enhancements in secure database interaction methodologies.

Conclusion

In conclusion, this research has rigorously examined the Secure Syntax framework as a potent defense against the prevalent threat of SQL injection attacks. By dissecting the methodology into its constituent stages—Lexical, Syntax, and Semantic Analysis—our study has illuminated a structured approach to validating SQL queries, ensuring their syntactic correctness and semantic integrity. The employment of placeholders and external parameter binding emerges as a particularly effective measure, neutralizing executable code in user input and relegating parameters to be securely handled by the database engine. Moreover, the pre-compilation of SQL statements represents a fusion of enhanced security with improved performance, highlighting the dual benefits of the Secure Syntax system.

ACKNOWLEDGMENT

I extend my deepest appreciation to all those who have played a part in the success of this project. My sincere gratitude goes to Professor Dr. Hang Zhang, for his unwavering guidance, mentorship, and support throughout the semester. His insights have been invaluable in shaping my understanding of the subject matter and refining the research methodology for this project work.

References

[1] https://journalofbigdata.springeropen.com/articles/10.1186/s40537-022-00678-0
[2] https://medium.com/@BhaktiKhedkar/exploiting-boolean-based-sql-vulnerability-42d387434a6d
[3] https://owasp.org/www-community/attacks/Blind-SQL-Injection
[4] https://journalofbigdata.springeropen.com/articles/10.1186/s40537-022-00678-0
[5] https://arxiv.org/abs/1605.02796.
[6] https://journalofbigdata.springeropen.com/articles/10.1186/s40537-022-00678-0
[7] https://akimbocore.com/article/sql-injection-exploitation-blind-boolean/
[8] SQL INJECTION STRATEGIES: PRACTICAL TECHNIQUES TO SECURE OLD VULNERABILITIES AGAINST MODERN ATTACKS (PAPERBACK) by Edoardo Caselli, Ettore Galluccio, Gabriele Lombari
[9] SQL Injection Attacks and Defense" by Justin Clarke-Salt
[10] SQL Injection Strategies" by Ettore Galluccio
[11] SQL Injection Defenses" by O'Reilly Media
[12] https://digitalcommons.usf.edu/cgi/viewcontent.cgi?article=7246context=etd