Product Search Application and SQL Injection Attack Prevention Efforts Through Pattern Detection System Using Regular Expression

Clarissa Nethania Tambunan 13523016[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung 40132, Indonesia
E-mail: [1]13523016@std.stei.itb.ac.id, [2]cntkreasi@gmail.com

***Abstract*—This article discusses the implementation of a pattern detection system as one of the prevention efforts for SQL Injection attacks in a product search application. The application has product data connected to a database that is highly vulnerable to such malicious attacks. SQL injection is a cyber attack technique in which an attacker inserts malicious SQL commands into data input from the user side with the aim of manipulating or damaging queries that will be executed in the database. The main purpose of creating a pattern detection system in this application is to utilize Regular Expression (Regex) to analyze and validate user input when searching for the desired product. The implemented system works by identifying textual patterns that have similarities with the characteristics of SQL injection attacks before user input is forwarded to be stored in the database which can then be processed. Based on the program, this article also discusses the analysis to evaluate system performance through a series of experimental tests involving simulations of safe search input and those that have the potential to be SQL injection, then analyzing how the system detects this pattern by responding to each type of query to protect data integrity.**

Keywords—SQL injection; Regular Expression; pattern detection; textual patterns; data integrity

I. Introduction

In today's digital economy, data has become the most valuable asset for almost every organization or company. The integrity and confidentiality of the data depend entirely on the security of the applications that manage it. So application security is not only a technical aspect but also the main basis of public trust and business continuity of the organization or company. However, the ever-growing scope of cyber threats shows that creating a strong defense against all attacks is an ongoing challenge. Security-related incidents that occur in the world, for example, a security hole in the PostgreSQL database system that was exploited as a zero-day to attack the security company BeyondTrust [1], are a reminder that no system is completely unattackable. This incident clearly emphasizes that vulnerabilities can emerge from anywhere, even from our trusted software components, and their impact can spread throughout the application environment.

The incident that compromised BeyondTrust demonstrates a crucial fact in cybersecurity that is every point in an application that interacts with a data layer is a potential source of attack. While zero-day attacks are sophisticated, their underlying principles often stem from exploiting how an application processes untrusted input or can lead to attacks on the database. So the search feature in this application, although seemingly simple, is essentially a direct interface to the heart of the system, the database. Every word a user types into the search box has the potential to be part of a command sent to the database. If this user input is not processed very carefully, it can lead to one of the most classic and effective attack techniques, SQL injection.

The mechanism of this attack can first be understood through what language is the target of this attack. SQL (Structured Query Language) is a standard language designed to manage and manipulate data in a relational database management system (RDBMS) [2]. Then, SQL injection is a cyber attack technique in which an attacker inserts or "injects" a series of malicious SQL commands into data input from the user side with the aim of tricking the application into running unwanted commands on the database. If this attack is successful, the impact can be fatal, starting from the breach and leakage of all data in the database which can also potentially change or delete data permanently. The consequences are not only financial losses, but also result in the loss of customer trust and reputational damage that is difficult to restore.

In overcoming this problem, a defense layer is needed that is able to validate all input before entering the database. This article focuses on the implementation of a detection system that is able to act as a filter for incoming input from users to prevent SQL injection attacks. This approach is fundamentally based on the algorithmic concept of string matching, which is why Regular Expression (Regex) is implemented in this product search application. By using Regex, an effective pattern detection system can be created to distinguish between reasonable input text and suspicious input.

The system works as a validation gateway that ensures the search keywords performed by the user by identifying textual patterns or signatures that are syntactically identical to common characteristics found in various SQL injection attack techniques. This system is built to detect anomalies and command structures that should not be present in a reasonable search input without having to execute the query itself. By

performing this pattern-based filtering before the input is forwarded for processing, the system can reject dangerous commands or requests in the application so that the threat cannot reach the database.

The main purpose of creating this application is to demonstrate how vulnerabilities to SQL injection attacks can be exploited in a simulated product search. In achieving this goal, this article not only shows a vulnerable search method, but also implements a pattern-based defense system using Regular Expression to filter potentially dangerous input from the database. Then, an analysis is carried out to evaluate the effectiveness of each method implemented, starting from the vulnerable method, the method with Regex detection, to the safe prevention method in order to show which is more practical in building an important layer of defense in protecting data integrity in a system.

II. Theoritical Foundation

A. Application Security

Application Security is the science and practice of protecting application software from external threats through the application of security measures during the software development lifecycle (SDLC). The goal is to prevent theft or piracy of data and code. The foundation of application security is often based on three main principles known as the CIA Triad: Confidentiality, Integrity, and Availability [8]. SQL Injection attacks directly threaten two of these three pillars: they violate confidentiality by leaking data that should not be accessed, and they violate integrity by giving an attacker the ability to change or delete data.

One of the most fundamental practices in application security to maintain integrity and confidentiality is input validation. This principle states that all data originating from untrusted sources, especially from user input, must be strictly checked and filtered before being processed by the application. Failure to perform proper input validation is often the root of many security vulnerabilities, including SQL Injection. The pattern detection system discussed in this article is one form of implementation of input validation practices.

B. SQL and Relational Datbases

Structured Query Language, better known as SQL, is a standardized language designed specifically for managing data in a relational database system. SQL serves as a communication bridge between applications and databases, allowing programs to perform fundamental operations such as storing, manipulating, and most importantly, retrieving data [2]. Its comprehensive capabilities make it the foundation for almost all modern applications that rely on structured data storage.

SQL commands, or SQL statements, are generally categorized based on their function. The most commonly used category in application development is Data Manipulation Language (DML), which consists of commands to interact with existing data. For example, in a product search application, the SELECT command is used to retrieve product data from the database to display to the user. The command might look like this: SELECT nama_produk, harga FROM produk WHERE kategori = 'Electronics';. Other DML commands include INSERT to add new data and UPDATE to change data. In addition to DML, there is also Data Definition Language (DDL) which is used to define the structure of the database itself, with commands such as CREATE TABLE used to build table schemas [3]. In the context of SQL injection attacks, attackers focus on manipulating DML statements, especially SELECT, to change their logic and extract data outside of their intended scope.

A relational database itself is a database model in which data is organized into one or more interrelated tables. Each table consists of rows, which represent an individual record or entity, and columns, which represent attributes or properties of that entity. As a concrete example in this application, the product table is used to store information about each item sold. Each row in this table represents a unique product, for example 'Baju Kemeja Pria Lengan Panjang'. Meanwhile, columns such as id, nama_produk, kategori, and harga are attributes that describe the product. This structured model is what allows SQL queries to retrieve data precisely. MariaDB, which is used in the implementation of this project, is one example of a popular open-source RDBMS that uses SQL as its main interaction language.

C. MariaDB

Fig 2.1 Illustration of MariaDB
(*Source: https://github.com/mariadb*)

MariaDB is an open-source, community-developed Relational Database Management System (RDBMS) and one of the most popular forks of MySQL. The project was started by the original MySQL developers after concerns about the takeover of MySQL by Oracle Corporation [4]. MariaDB was designed to be a direct (drop-in) replacement for MySQL with high compatibility, while offering better performance, richer features, and a completely free and open license (GPL). As an RDBMS, MariaDB relies on SQL as its primary language for all data operations. In this study, MariaDB was chosen as the database platform to implement and test the product search application because of its stability, speed, and open-source nature.

D. SQL Injection Attack

According to OWASP (Open Web Application Security Project), Injection attacks are generally ranked third in the list of the ten most critical web application security risks in 2021, with 94% of tested applications having traces of Injection vulnerabilities in some form [9]. Among the various types of injection attacks, SQL injection is one of the most damaging to databases. This attack occurs when malicious data that usually comes from user input is combined into an SQL query by the

application. Attackers exploit this vulnerability to insert malicious SQL syntax, which is ultimately executed by the database as part of the original command.

The basic mechanism of this attack is to "break" the data context and enter the command context. For example, in a query that searches for products by name (... WHERE nama_produk = 'user_input'), an attacker could enter input such as ' OR '1'='1'. The first single quote (') is intended to close the data string early. The next part, OR '1'='1', is then interpreted by the database not as a product name, but as a new logical condition that is always true. As a result, the WHERE clause becomes true for all rows, causing the database to return the entire data in the table [9]. Another common attack is to use the UNION SELECT operator to combine the results of completely different queries, allowing the attacker to read data from other tables that would otherwise be inaccessible.

The impact of a SQL Injection attack varies greatly depending on the access rights the application has to the database, but is almost always serious. The most common impact is the leakage of sensitive data, such as user details, financial information, or trade secrets. In addition, attackers can also modify or even delete data (for example by inserting DELETE or DROP TABLE commands), which can completely disrupt business operations. In some cases, SQL injection can even be used to take full control of the database server itself [9].

E. String Matching

String matching is a classical problem that focuses on designing algorithms to efficiently find all occurrences of a short string, known as a pattern (P), within a much longer string, called a text (T) [6]. The application of this concept is very broad, ranging from the "search" function in text editors to DNA sequence analysis. In the context of application security discussed in this paper, the concept is applied directly: the "text" is the entire input string sent by the user, while the "pattern" is the signature or fingerprint of a known malicious code fragment.

While there are many highly efficient classic string matching algorithms, such as brute-force, Knuth-Morris-Pratt (KMP), or Boyer-Moore, these algorithms are generally designed to find exact matches of literal strings. However, cyberattacks rarely come in one fixed form. An attacker can vary their attack in many ways, such as using different cases or adding spaces. Therefore, a more flexible matching approach is needed, one that does not only look for the same text, but is able to recognize a more abstract pattern. This is where Regular Expressions play a major role in this application.

F. Regular Expression (Regex)

Regular Expression (Regex) is a technique in the form of a sequence of characters that defines a specific search pattern. Rather than searching for text literally, Regex allows searching based on abstract rules that can recognize complex text formats [6]. The main strength of Regex lies in the use of special characters called metacharacters, each of which has its own meaning and function in defining a pattern. This ability makes Regex a very effective tool for tasks such as input validation, log scanning, and in the context of this research, security anomaly detection. The following is the general notation of Regular Expression.

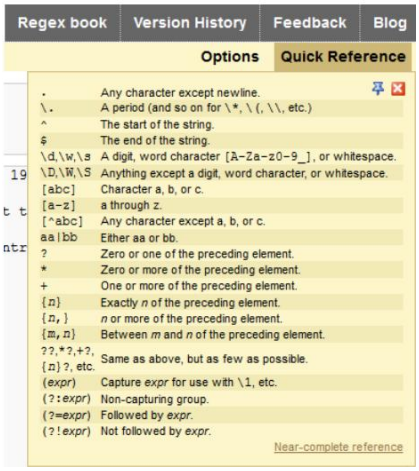

Fig 2.2 Basic Notation of Regex
(*Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/24-String-Matching-dengan-Regex-(2025).pdf*)

In the SQL Injection detection system built, several of these metacharacters are combined to form a comprehensive detection pattern. The pipe operator |, which functions as an "OR" logic, is used to combine several sub-patterns into one. For example, a pattern can be designed to search for the presence of a single quote character (') OR the SQL comment symbol (--) OR the or keyword enclosed in spaces. Thus, a single Regex expression is able to perform efficient multi-condition checks on an input string to identify various possible red flags.

G. PreparedStatement as a Prevention Method

In addition to detection, prevention is an important aspect of security. In the context of Java programming language, the most recommended SQL injection prevention method is to use PreparedStatement. This is a feature of the JDBC (Java Database Connectivity) interface that allows the execution of pre-compiled SQL queries [7]. The security mechanism of PreparedStatement is the separation of logic and data. The SQL query template with placeholders (?) is first sent to the database for compilation. After that, user input is sent separately only as parameter values. In this way, user input will never be interpreted as part of the SQL command, effectively eliminating the root cause of SQL injection vulnerabilities.

III. Product Search Application

This application describes the technical aspects of the research, starting from the implementation of the database structure used, the design of classes in the application, to a detailed explanation of each search method implemented. The purpose of this application is to provide a clear picture of how the pattern detection system and prevention methods are built in a simulation application, which will be the basis for experimental testing in the next chapter.

A. *Product Database Implementation*

The foundation of this application is a relational database designed to store product data. This database is created on a MariaDB server with the name data_product. The main structure of this database is a single table named produk, which is designed to store essential information about each item. The schema of this table and the initial data used for testing are initialized using the following SQL code.

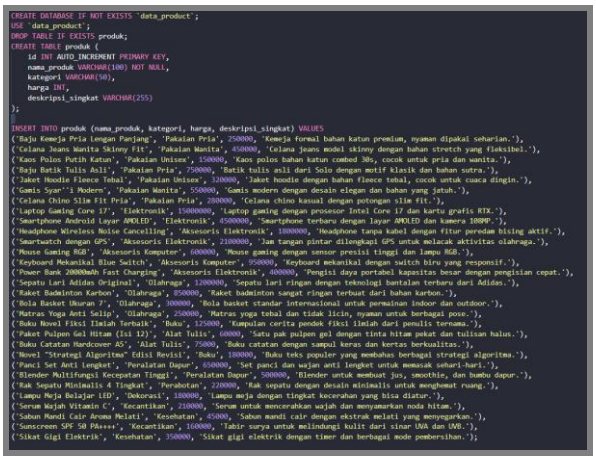

Fig 3.1 Implementation of product.sql
(*Source: https://github.com/ClarissaNT44/Product-Search-Application_13523016/blob/main/database/product.sql*)

B. *Product Representation Implementation*

Before the search algorithm can be formed, two main classes are needed in Java language to manage data and services. The first class is Produk contained in src/Produk.java, which serves as a data model to represent each row of the product table as an object. This allows data to be managed in a more structured way within the application. Here is the code for the Product class that was created.

Fig 3.2 Implementation of Produk.java
(*Source: https://github.com/ClarissaNT44/Product-Search-Application_13523016/blob/main/src/Produk.java*)

The second class is SearchProduk which is located in src/SearchProduk.java, which acts as a service layer. This class is responsible for accommodating all logic related to interaction with the database. It contains three different search methods that are the core of this experiment. The relationship between these two classes is very close, namely the SearchProduk class will take raw data from the database and convert it into a list of Produk objects before displaying it. Here is the code for the SearchProduk class that was created.

Fig 3.3 Implementation of SearchProduk.java
(*Source: https://github.com/ClarissaNT44/Product-Search-Application_13523016/blob/main/src/SearchProduk.java*)

C. *Implementation of Vulnerable Method*

The first method, searchVulnerable(), was designed to demonstrate unsafe programming practices and serve as a basis for comparison. The crux of the vulnerability lies in the way the SQL query is constructed, by concatenating user keyword input directly into the SQL string using the + (string concatenation) operator.

String sql = "SELECT * FROM produk WHERE nama_produk LIKE '%" + keyword + "%'";

With this approach, the application does not differentiate between data and commands, leaving an opening for attackers to insert malicious SQL syntax into keywords and manipulate the logic of the query to be executed.

D. Implementation of Regex Detection Method

The second method, searchWithRegexProtection(), implements a preventive defense layer. Before building the SQL query, the user's keyword input is validated using a predefined Regular Expression pattern to look for signs of attacks.

```
String sqlInjectionPattern = ".*'.*|.*--.*|.*#.*|.*\\s(or|and|union|select)\\s.*";
if (Pattern.matches(sqlInjectionPattern, keyword.toLowerCase())) {
    return;
}
```

This Regex pattern serves as a security rule to detect the presence of suspicious textual patterns, such as single quote characters ('), SQL comments (--), or the keywords or and union. If such a pattern is found, this method will immediately reject the input and stop the process before a malicious query can be formed and sent to the database.

E. Implementation of Safe Prevention Method

The third method, searchSecure(), implements the best practice for database interaction in Java, namely using PreparedStatement. This approach is fundamentally different and eliminates the root cause of the vulnerability.

```
String sql = "SELECT * FROM produk WHERE nama_produk LIKE ?";
PreparedStatement pstmt = conn.prepareStatement(sql)
pstmt.setString(1, "%" + keyword + "%");
```

The process consists of two steps, that is first, an SQL "template" with placeholders (?) is sent to the database for pre-compilation. Second, the user's keyword input is sent separately as a parameter value using the setString() method. This way, the database will never interpret the keyword as part of the SQL command, but only as plain literal string data, thus completely thwarting SQL Injection attacks.

IV. Results and Analysis

This chapter presents the results of experimental testing conducted on a product search simulation application. Each search method implemented in the previous chapter was tested using a series of input scenarios, both normal input and input containing SQL injection attack loads. The purpose of this chapter is to present the test results objectively and to conduct an in-depth analysis of the behavior of each method to evaluate its effectiveness.

A. Test Scenario

Testing was done by running the Main class found in src/Main.java, which then ran three different experiments to compare each search approach. The three methods tested were searchVulnerable() which uses string concatenation, searchWithRegexProtection() which implements Regex pattern validation, and searchSecure() which uses PreparedStatement. To ensure a fair comparison, each method was tested using the same set of inputs. The inputs, which included normal search keywords and various variations of malicious keywords to simulate SQL injection attacks defined directly in the Main.java code, as will be shown in the following test results section.

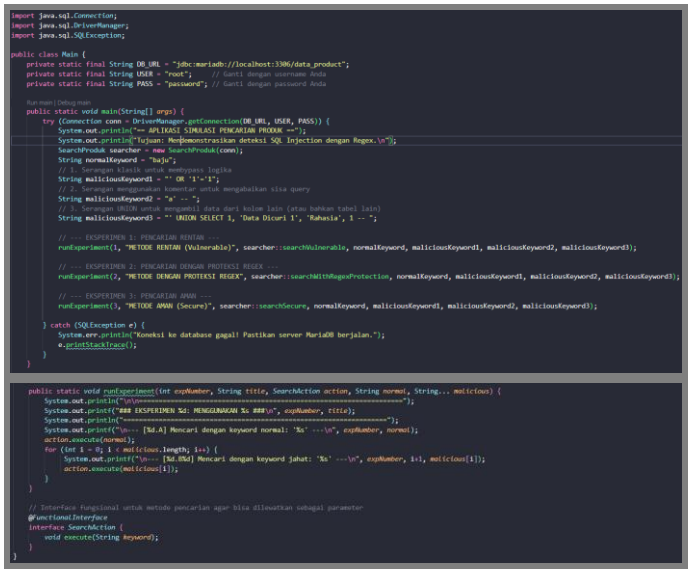

Fig 4.1 Implementation of Main.java
(*Source: https://github.com/ClarissaNT44/Product-Search-Application_13523016/blob/main/src/Main.java*)

B. Test Results

The result of executing the Main.java program produces output recorded in the table. The following is a summary of the results.

Table 4.1 Summary of Test Results

Experiment	Method	Input	Result	Initial Conclusions
Vulnerable	searchVulnerable	baju	Found 2 products	Functioning normally
		' OR '1'='1'	Show all 30 products	Very vulnerable
		a' --	Found 1 product	vulnerable
		' UNION SELECT ...	SQL error	Very vulnerable
Regex Protection	searchWithRegexProtection	baju	Found 2 products	Functioning normally
		' OR '1'='1'	Request Blocked	Safe
		a' --	Request Blocked	Safe
		' UNION SELECT ...	Request Blocked	Safe

Safe Search	searchSecure	baju	Found 2 products	Functioning normally
		' OR '1'='1'	No Results	Safe
		a' --	No Results	Safe
		' UNION SELECT ...	No Results	Safe

C. Result Analysis

The result of executing the Main.java program produces output recorded in the table. The following is a summary of the results.

1. Vulnerable Method Analysis (searchVulnerable)
   The searchVulnerable method showed a complete failure to handle malicious input and validated the existence of a critical security vulnerability. In the classic ' OR '1'='1' attack, the application concatenates the input into a SQL string, resulting in a query ... LIKE '%' OR '1'='1'%'. The OR '1'='1' clause, which is always true, effectively breaks the logic of the WHERE clause, causing the database to return all 30 existing product records. This is a clear demonstration of a massive data leak. Then, a comment-based attack such as a' -- also successfully changes the query logic to search for products whose names end with the letter 'a', proving that an attacker can manipulate the search function to their liking. Although the UNION SELECT attack results in a SQLIntegrityConstraintViolationException error, this result only further strengthens the evidence of the vulnerability. The error occurs not because of a security mechanism, but because of a column mismatch. This shows that the application is willing to pass malicious commands to the database, and the attack could have succeeded if the attacker guessed the number of columns correctly.
2. Regex Detection Method Analysis (searchWithRegexProtection)
   The searchWithRegexProtection method consistently demonstrated its effectiveness as a preventive defense layer. When tested with normal input, the system correctly identified it as safe input and continued the search process without any false positives. However, in all malicious input scenarios, the Regex pattern-based detection system successfully identified and blocked the request before the SQL query could be constructed. The ' OR '1'='1' attack was detected because the pattern .*'.*|.*\\s(or|and...)... matched the presence of single quotes and the or keyword. Similarly, comment and UNION-based attacks were also successfully stopped because they contained textual patterns (--, select, union) that have been defined as malicious. These results prove that the string matching algorithm strategy implemented through Regular Expressions is capable of functioning as a reliable validation filter at the application layer to reject input that does not conform to the expected format.
3. Safe Prevention Method Analysis (searchSecure)
   The searchSecure method validates this approach as the most robust and fundamentally secure. Across all simulated malicious input types, it consistently returns "No products found". This behavior occurs because PreparedStatement removes the root cause of the SQL injection vulnerability itself, by separating the command logic from the data. When the application uses a placeholder (?), malicious strings such as ' OR '1'='1' are no longer interpreted as part of the SQL command. Instead, they are sent to the database purely as literal data values. As a result, the database will safely try to find product names that literally contain the text ' OR '1'='1', which of course does not exist in the table. This method has been proven to not only detect the symptoms of an attack, but prevent the attack from occurring in the first place by ensuring that user input is never executed as code.

V. CONCLUSION

Based on the implementation and analysis of the test results, this study shows that the application development method that relies on direct string concatenation to build SQL queries is proven to be very unsafe. Simulation testing validates that this approach not only fails to handle malicious input, but can also be easily exploited to cause all data leakage, which reaffirms how dangerous the practice is. In contrast, this study successfully proves that the implementation of Regular Expression (Regex) as an implementation of the string matching algorithm strategy is an effective detection method. The pattern-based system is able to identify and block all variations of simulated attacks, confirming its role as a reliable defense layer at the application level to filter input before interacting with the database.

Although Regex-based detection has proven to be effective, the most fundamental and robust prevention method is the use of PreparedStatements in Java. This approach essentially eliminates the root cause of SQL injection vulnerabilities by clearly separating the SQL command logic from the data inputted by the user. Therefore, it is recommended that the use of PreparedStatements be made a standard practice in developing secure applications. In the meantime, Regex-based detection systems can serve as an invaluable additional layer of security, such as in Web Application Firewalls (WAFs) to log and reject attack attempts as an initial defense. Further research can be conducted to explore the development of more complex Regex patterns to be able to recognize more sophisticated attack techniques.

VI. Appendix

The application or program in this article can be accessed here.

The author also made a video explanation of this article can be accessed here.

Acknowledgment

The author would like to express gratitude and thanks to God Almighty because thanks to His grace the author was able to complete this article well. In particular, the author would also like to thank the lecturers and people who have provided much support in the preparation of this paper, namely:

1. Mrs. Dr. Nur Ulfa Maulidevi, S.T., M.Sc. as a lecturer of IF2211 Strategi Algoritma class 01 who has guide the author in teaching/knowledge to understands this course.
2. Mr. Dr. Ir. Rinaldi Munir, M.T. as a lecturer of IF2211 Strategi Algoritma class 02, and the author uses many of his books.
3. Friends who provided a lot of input and ideas about this article.
4. The family who always supports the author and gives general views.

This article is certainly still not perfect, therefore the author expects constructive criticism and suggestions from various parties for improvement in the future. Hopefully this article is useful for readers.

References

[1] Abrams, L. 2024. *PostgreSQL flaw exploited as zero-day in BeyondTrust breach*. BleepingComputer. https://www.bleepingcomputer.com/news/security/postgresql-flaw-exploited-as-zero-day-in-beyondtrust-breach/ (Accessed 9 June 2025)

[2] GeeksforGeeks. (2023). *What is SQL?*. https://www.geeksforgeeks.org/what-is-sql/ (Accessed 9 June 2025)

[3] MariaDB Corporation. (n.d.). *SQL Statements*. MariaDB Knowledge Base. https://mariadb.com/docs/server/reference/sql-statements (Accessed 14 June 2025)

[4] MariaDB Foundation. (n.d.). *About MariaDB*. MariaDB.org. https://mariadb.org/about/ (Accessed 14 June 2025)

[5] MariaDB Foundation. (n.d.). *MariaDB Server Source Code Repository*. Github. https://github.com/mariadb (Accessed 14 June 2025)

[6] Munir, R. 2024. *String Matching dengan Regular Expression*. Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/24-String-Matching-dengan-Regex-(2025).pdf (Accessed 8 June 2025)

[7] Oracle Corporation. (n.d.). *Interface PreparedStatement*. Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification. https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/java/sql/PreparedStatement.html (Accessed 21 June 2025)

[8] OWASP Foundation. (n.d.). *OWASP Developer Guide: Security Fundamentals*. https://devguide.owasp.org/en/02-foundations/01-security-fundamentals/ (Accessed 22 June 2025)

[9] OWASP Foundation. (2021). *A03:2021 – Injection*. OWASP Top 10. https://owasp.org/Top10/A03_2021-Injection/ (Accessed 22 June 2025)

Statement of Originality

I hereby declare that the article I wrote is my own writing, not an adaptation or translation of someone else's article, and is not plagiarized.

Bandung, 24 Juni 2025

Clarissa Nethania Tambunan 13523016