# *SQL Injection Attack: Demonstration of SQL Injection and Prevention with SQLDetc on Database System*

Chong Ting Fung
*School of Engineering and Technology*
No. 5, Jalan Universiti, Bandar Sunway, 47500, Selangor Darul Ehsan, Malaysia
23101686@imail.sunway.edu.my

Lee Shu Ann
*School of Engineering and Technology*
No. 5, Jalan Universiti, Bandar Sunway, 47500, Selangor Darul Ehsan, Malaysia
23099740@imail.sunway.edu.my

Tong Zi Qian
*School of Engineering and Technology*
No. 5, Jalan Universiti, Bandar Sunway, 47500, Selangor Darul Ehsan, Malaysia
22087852@imail.sunway.edu.my

***Abstract*—SQL Injection Attack (SQLIA) is a widespread vulnerability in Database Management System (DBMS) that allows attackers and hackers to alter SQL queries which can result in sensitive data being accessed by them or modified without authorization. Due to the poor implementation and evolving attack methods by the attackers, many DBMS systems are still susceptible to SQLIA although numerous prevention techniques have been introduced. This paper examines various SQLIA and evaluates effectiveness of SQLDetc a tool developed by the team implemented in Oracle DBMS. The findings and results of SQLIA potency and effectiveness of SQLDetc in preventing SQLIA will be discussed in the research paper.**

***Keywords*—*Structured Query Language (SQL), SQL Injection Attack (SQLIA), PHP, MySQL, Database Management System (DBMS), Oracle Database System, SQL Prevention*

## I. INTRODUCTION

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by DBMS. As database become increasingly integrated with web applications, the security and integrity has become more significance. SQLIA is an attack that exploits improper handling of user inputs in SQL queries. By inserting malicious code into user input fields, attackers or hackers can alter database queries to gain unauthorized access, retrieve confidential information, or corrupt user data [2].

SQLIA is one of the most common vulnerabilities in web applications. According to the Open Worldwide Application Security Project (OWASP) Top Ten report in 2017, SQL injection was ranked as the top challenge to Web application systems. In OWASP 2021, the ranking for injection has been slightly decreases. SQL injection attacks exploit weakness using queries and access sensitive information directly. The lack of robust security practices makes systems highly vulnerable. In this research study, the objectives are to demonstrate how SQLIAs are executed through a DBMS and propose a prevention solution.

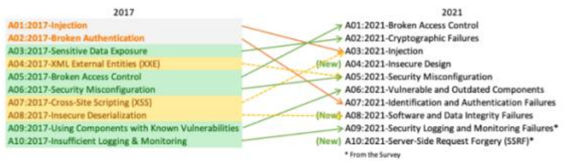

2017
A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring
2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
(New) A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
(New) A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
(New) A10:2021-Server-Side Request Forgery (SSRF)*
* From the Survey

Fig. 1. OWASP Top 10 for 2017 and 2021 [22]

### A. *Overview of SQL Injection Attack*

SQLIA is an extreme injection or attack in a website that gives attackers full access to DBMS which runs on the applications [1]. A SQLIA can also refer to as a web attack which accesses the dataset that stored inside the DBMS. This can be done by inserting inappropriate SQL code through user input and immediately concatenating it with the original SQL queries issued by the client application [2], [4]. This could create chances for the hackers or attackers to gain unauthorized access or demonstrate against the unauthorized dataset. SQLIA may succeed if user input is not properly validated and sanitised. The hackers or attackers can gain the unauthorized access in a number of methods especially injection through user input and injection through cookies [3]. Injection through user input is the most common method for them to gain unauthorized access or data. Most of the website typically use forms to either gather user data or allow users to select what kinds of information they want to get. The forms include the text fields that the hackers will inject malicious SQL code [3]. For example, the hackers or attackers will inject malicious SQL code such as "test' OR 1=1 -- " into the user input text field. By injecting this SQL code, they can just bypass the password check because OR 1=1 will always return true. Another example is the hackers or attackers can also delete or modify the dataset that stored in the DBMS if they gain the unauthorized access. Attackers might compromise the availability and integrity of the dataset by corrupting, removal, or even deleting the entire dataset tables using SQL statement such as ALTER, UPDATE, DELETE, or DROP [4].

Furthermore, injection through cookies is also such a method for hackers or attackers to gain the unauthorized access or data. Cookies are used by modern website to store user preferences. Cookies also known as the files that are kept on the client system device which include state data that produced by the website. For instance, inappropriate SQL code is added by the attacker to the cookies that are saved on their devices. The database system will become vulnerable to attacks if SQL queries were created using these cookies without the proper validation and sanitation [3]. Overall, SQLIA allows hackers or attackers to bypass authentication procedures and access private information that stored inside the database system without authorization. This means that they have the high-level access to the user's account by knowing all private information of that user [3]. If they gain full access control, the database system may lose of control. It is because the attackers can modify or delete the dataset that stored in the DBMS, and this can lead to the disruption of the server in the database system.

TABLE I. TYPES OF SQL INJECTION ATTACKS (SQLIA)

| Types of Attack | Attack Intent | Description |
|---|---|---|
| Tautologies | Bypassing authentication, extracting data, identifying injectable parameters. | Injects code to force a condition to always evaluate to true |
| Logical Incorrect Queries | Database fingerprinting, identifying injectable parameters | Exploits error message which gather important information about the database structure |
| Union Query | Extracting data | Union is used to combine the injected query with a given query to retrieve information about other tables |
| Piggy-Backed Queries | Adding or modifying data, perform denial of service | Inject additional SQL commands into the original one |
| Stored Procedures | Privilege escalation, denial of service | Exploits vulnerabilities in database-stored procedures |
| Inference | Identifying injectable parameters, determining database schema | |
| 1. Boolean-based | | The attacker sends an SQL query to the database by asking TRUE or FALSE questions. Information return depends on the query is true or false. |
| 2. Time-based | | Attackers obtain information by observing timing delays through the database responses |
| Alternate Encodings | Evading detection | Modified injected text to the attack to evade detection by several prevention techniques. This attack is usually a combination of other attack techniques. |

## II. LITERATURE REVIEW

### A. *Structure of Literature Review*

The literature review mainly focuses on the existing advancements in detection and prevention techniques of SQLIA in the database system. The performance, stability, and efficiency are being focused and examined based on the detection and prevention methods against SQLIA. The articles reviewed were selected based on their relevance to detection and prevention against SQLIA in DBMS. The reviewed articles were published in high-quality peer-reviewed journals and conferences in order to give more thorough and accurate explanations with an emphasis on SQLIA detection and prevention.

### B. *Review of Articles*

Halfond et al. [1] suggested a grammar-based method to identify unauthorized changes. As they explored how poor coding practices make SQL queries vulnerable to SQL injection attacks. The study found this approach is effective against common attacks.

Kindy et al. [2] discuss the threat of SQL Injection, where attackers manipulate databased queries due to poor coding practices. Solutions like SAFELI are covers for early detection.

Jemal et al. [3] examined how SQLIA affects web application security and the weaknesses of current protection methods. They found that machine learning and ontology-based solutions are effective in detecting SQLIA, but there are still small issues with handling certain types of attacks.

Paul et al. [4] addresses the limitations in detecting different types of SQL injection attacks. A hybrid CNN-LSTM model was introduced because it performs well in detecting multiple types of attack.

Orso et al. [5] introduced a tool named WASP to prevent SQLIA. A hybrid solution combining static analysis and real-time monitoring is introduced. WASP successfully blocks all SQL injection attacks without false positives, ensuring a safe query construction.

Alwan et al. [6] explored the widespread vulnerability of SQLIA due to poor input validation. Tools like AMNESIA, WASP, and machine learning are introduced to handle traditional SQLIA, but they struggle into handling more complex attacks that combine SQL injection with other threats.

Azman et al. [7] used machine learning to detect SQLIA by analysing access logs. Methods like signature-based detection and Boyer's Moore algorithm were implement. The study showed high accuracy but highlighted challenges in real-time detection.

Shahriar et al. [8] discuss the widespread SQLI vulnerabilities and their impact on web security. The study explores solutions like static analysis and run-time monitoring to spot harmful queries.

Falor et al. [9] compared different methods of machine learning, including decision trees, SVM, and CNN. These machine learnings help to detect SQLIA. In this study, they found that CNNs performed the best, as it provides highest accuracy and precision.

Nasereddin et al. [10] explored the most useful ways to detect each type of SQLIA. The study proposed the importance of security concerns to secure web applications.

Singh et.al [11] found that penetration testers have faced issues when using manual testing to find the vulnerabilities of web application and SQL Injection Attack. The method of automated penetration testing in which consists dynamic technique is being introduced to violate SQL Injection Attack.

Jahanshahi et.al [12] introduced a hybrid static-dynamic analysis which is known as SQLBlock to prevent SQL Injection Attack. Existing methodologies cannot support the object-oriented programming leading to ineffectiveness of the prevention methodologies on real-world web applications such as Wordpress, Joomla, and Drupal. The study focused on the performances of SQLBlock to prevent SQLIA.

Joy et.al [13] found that the SQLIA is one of the factors that a web application become the target for cyberattacks. Detection and prevention methods such as Non-Machine Learning, Machine Learning, Reinforcement Learning, and Pattern Matching are being focused on this study.

Abdullah et.al [14] reviewed on the existing detection techniques of SQLIA such as SAFELI. Some considerations also being provided when developing a web application such as query parameterization and prepared statements is required to prevent SQLIA.

Haixia et.al [15] explored that the SQLIA is one of the factors that leading security issues on a web application become serious. Therefore, they introduced a database scheme of security testing for web applications. The study shows that the detection of SQL Injection Points is quicker and more effective.

Abikoye et.al [16] identified that SQLIA is unauthorized access from attackers for the purpose of accessing, editing, and destroying the data and information. An algorithm which is known as Knuth-Morris-Pratt (KMP) string matching is implemented to prevent SQLIA and cross-site scripting.

Deriba et.al [17] stated that existing prevention methodologies may failed to prevent the SQL Injection Attack. The study produced a combination of static and dynamic analysis which also contains machine learning concept. This study showed that hybrid analysis and ANN approaches are more efficient to detect SQLIA.

Mishra et.al [18] introduced a three-tiers logical design view of a web applications to easily analyse the SQLIA. The study also found that using multi-core architecture can help to minimize the response time of a web application.

Nagpal et.al [19] reviewed different types of detection and prevention methodologies such as static, dynamic, encryption, defensive coding, and obfuscation. The study also shows the probability of successful detection SQLIA by different techniques and comparison is done between different techniques and methodologies.

Zhang et.al [20] stated that the SQL Injection Attack is not just cause the security of individual website, but all the entire database system. Manual testing of detection for SQLIA is done by using PHP code to check the vulnerability of database system. Prevention methods also being proposed such as filtering the sensitive keywords for user input data. Apache server that focusing on the Rewrite module also being used to prevent SQLIA.

### *C. Comparison Table*

TABLE II. COMPARISON TABLE FOR REVIEWED ARTICLES

| **Reference** | **Problem Statement** | **Algorithm/ Methodology** | **Key Findings** |
|---|---|---|---|
| Halfond, Viegas, and Orso, 2006 [1] | SQL Injection affects numerous web applications through user-input vulnerabilities | Analysis of countermeasures like static analysis tools, runtime monitoring systems and defensive coding | Identified various detection and prevention techniques of SQLIA |
| Kindy and Pathan, 2011 [2] | SQL Injection allows attacker to manipulate database queries, leading them to access data without authorization | Combination of static analysis and runtime monitoring to enhance accuracy of detection | Uses SAFELI Framework to present high precision in identifying vulnerabilities |
| Jemal, Cheikhrouhou, Hamam, Mahfoudhi, 2020 [3] | Impact of SQLIA on web application security | Review of Ontology and ML-Based solutions for SQLI detection | ML techniques showed results of SQLIA in terms of accuracy and adaptability |
| Paul, Sharma, and Olukoya, 2024 [4] | AMNESIA as a traditional solutions struggle to detect complex or unknown SQLi attacks | Implemention solution where reflecting real-world application of SQL Injection attacks | Improve detection accuracy of AMNESIA over the traditional methods. |
| Orso, Lee, and Shostack, 2008 [5] | SQLIA exploit the lack of input validation and leads to unauthorized database access | A hybrid solution which combines static analysis is introduced to identify vulnerabilities. WASP tools were used | WASP blocks all tested SQLIA successfully. |
| Alwan and Younis, 2017 [6] | Explores widespread vulnerability of applications of SQLIA | Uses different techniques and focuses on query analysis. (AMNESIA, WASP, SQL Prevent) | Tools are effectively address the issues of SQLIA |
| Azman, Marhusin, and Sulaiman, 2021 [7] | SQL injection still the most significant threat to web system | Studies machine learning to classify web requests harmful or malicious | Find out on enhancing for integrating real-time detection |
| Shahriar and Zulkernine, 2012 [8] | This paper highlights the widespread of SQLI vulnerabilities and the impact of it | Runtime monitoring for known and unknown vulnerabilities | Information-theoretic models can detect unknown vulnerabilities |

| | | | |
|---|---|---|---|
| Falor, Hirani, Vedant, Mehta, and Krishnan, 2022 [9] | SQLI attacks are harmful | Compared ML algorithms on a comprehensive SQLi dataset | CNN models are highly effective than traditional ML models |
| Nasereddin, ALKhamaiseh, Qasaimeh, and Al-Qassas, 2021 [10] | Attackers could manipulate SQL queries and access sensitive data. | Explore most efficient and useful ways to detect each type of SQL attack | Importance of security concerns to secure web applications |
| Singh and Kumar, 2020 [11] | Manual testing used by penetration testers faced issues while finding the vulnerabilities and SQLi on a web application. | Automated penetration testing which contains dynamic technique focused in assigning the concept of shape Status (FS) as an attack. | Proposed algorithms can be implemented in any programming language such as Java, and so on. |
| Jahanshahi, Doupé, and Egele, 2020[12] | Object-oriented programming cannot be supported by existing methodology for SQLi prevention methods. | A combination of static and dynamic analysis is proposed which is known as SQLBlock. | Ability of SQLBlock against SQLi is evaluated in terms of performance, precision, and defense capability. |
| Joy and Daniel, 2022 [13] | SQL Injection Attack gives chances for a web application to become a target for cyberattack. | Review of detection and prevention methods such as Non-Machine Learning Methods, Machine Learning Methods, and so on. | Comparison between each technique in terms of different evaluation parameters. |
| Abdullah, Muhammad, and Malik, 2022[14] | SQL Injection Attack becomes easily attack the web application as technology advances these few years. | Review of existing detection techniques of SQL Injection Attack such as SAFELI. | Prevention methods should be considered when developing web applications. |
| Haixia and Zhihong, 2009[15] | Security issues on web database due to SQL Injection Attack. | A database scheme of security testing for web applications. Attack rule library is being set up and test cases are generated. | SQL Injection Points that proposed are fast and quicker, but the ability of detection techniques needed to be improved. |
| Abikoye, Abubakar, Dokoro, Akande, and Kayode, 2020[16] | Unauthorized access from attackers for accessing, editing, and destroying data and information. | Knuth-Morris-Pratt (KMP) string matching algorithm is used as prevention method for SQL Injection and Cross-Site scripting. | Proposed algorithms can log the attack entry and block the operating system using its MAC address. |
| Deriba, SALAU, Mohammed, Kassa, and Demilie, 2022 [17] | Existing methodologies have higher chances of failing to counter SQL Injection Attack. | A combination of static and dynamic analysis, and machine learning concept of approaches. | Three injection parameters such as user input field, cookies, and server variables. |
| Mishra, Mehra, and Dubey, 2023 [18] | Exploitation on user input field by attackers lead to SQL Injection Attack. | A three-tiers logical view design of a web application is used to analyze the SQLi which are the user interface tier, business logic tier, and database tier. | Minimize the response time (search time and reaction time) by using multi-core architecture. |
| Nagpal, Chauhan, and Singh, 2017 [19] | The evolution of SQL Injection Attack makes the government worry and financial institutions. | Review of different SQLi detection and prevention techniques from different parameters. | The possibility of successful detection is being proven out. |
| H. Zhang and Zhang [20] | SQL injection vulnerabilities will affect entire database system. | Manually testing the vulnerabilities of website by using PHP code. | Prevention methods such as filtering sensitive keywords for user data input. |

## D. *Writing the Literature Review*

The reviewed articles defined different types of methodologies for detecting and preventing SQLIA which emphasising their advantages, limitations, and potential areas of development. Studies by [1], [2], [5], [8] stated that static and runtime monitoring analysis is important to identify the vulnerabilities of a website. Tools such as WASP has been used by [5] to blocks all SQLIA against the database system. While studies by [2] and [14] examines the uses of SAFELI to detect the SQLIA. Machine-based learning methods also being proposed by [3], [7], [9], [13] as a potential way to detect SQLIA. Study by [9] found that CNN models are more accurate and efficient than the ML models. Studies by [3] and [7] uses machine learning methods to efficiently detect the SQLIA. Study by [13] compare between different techniques against SQLIA by comparing in different evaluation parameters. Algorithm is being proposed by [16] such as Knuth-Morris-Pratt string matching to prevent SQLIA. Studies by [14] and [20] stated that prevention methods are important to prevent SQLIA such as prepared statements, query parameterization, and so on. Study by [19] reviewed and compared different types of detection and prevention methods based on the possibility of successful rate. There are still important gaps despite these developments. According to [4] and [6], tools such as AMNESIA and WASP are unable to manage sophisticated attacks that combine SQLIA with additional threats. Study by [12] discovered the weakness of current approaches in object-oriented environments and proposed SQLBlock as a legitimate solution. Study by [15] proposed SQL Injection Points which are faster and efficient, but it lacks improvement in the detection techniques. According to [17] and [18], scalability problems in large-scale environments continue to be a concern. Therefore, proper prevention methods against SQLIA should be proposed to prevent them.

### *E. Summary of Literature Review*

The reviewed article highlights those methods like prepared statements, input validation and machine learning are effective in preventing SQLIA. Machine learning systems like CNNs and hybrid models are the most effective models concluded in the literature review. Tools like SQLBlock and SAFELI improve security by combining static and dynamic analysis. However, there are some limitations met in this research. For example, performance overheads, lack of user-friendly implementation and inconsistent real-world application.

## III. RESEARCH METHODOLOGY

### *A. Introduction to Research Methodology*

This section outlines the methodology that used to simulate scenarios for SQLIA and prevention through SQLDetc within a DBMS. PHP-based code and MySQL are used to simulate SQLIA scenario, provide prevention methods to SQLIAs and use that knowledge for SQLDetc. The goal of the research methodology is to evaluate strength of SQLIAs through tests and from that result later implement into SQLDetc for prevention. The SQLIAs that will be focused on the methodology is tautology, logical incorrect query, union query, piggy-backed query, stored procedure, inference (Boolean-based and Time-based) and alternate encodings. The reason focusing on these seven types of SQLIA is because they are the most extensively used attack techniques. Each SQLIA is vary in their attack strategies, therefore prevention methods are required to prevent each type of SQLIA. The methodology will also focus on the successful rate of the prevention method to prevent or reduce the SQLIA so it can help develop SQLDetc, our prevention tool. As for SQLDetc, the implementation of it and the test of it will be mentioned too.

### *B. Research Design*

The research applies a quantitative approach and an experimental study to investigate the simulation of SQLIA and prevention methods in terms of their effectiveness. The purpose of the research methodology is to simulate the SQLIA by simulating in test scenarios in a controlled environment and proposing prevention methods to prevent SQLIA. The methodology is done using a PHP-based code that connected to the MySQL database system.

### *C. Steps Involved in the Methodology*

#### *1) Identification of Research Objectives*

The primary objective of this is to examine the vulnerabilities of SQL databases through SQLIA, demonstrate the potential impact of different types of attack techniques and implement prevention methods to secure database against these threats. This study will explore seven types of SQL injection techniques, including tautologies, logical incorrect queries, union query, piggy-backed queries etc. By simulating these attacks, this study aims to illustrate how attackers can exploit the weaknesses of SQL databases to steal or change data.

Additionally, this research will focus on implementing prevention techniques for each type of SQL injection to demonstrate the risks of SQLIA. This is done so by developing our tool, SQLDetc where it can identify SQL injection attacks by analyzing queries and logging suspicious activities into a dedicated audit table of your database. As of now, SQLDetc is only accessible in Oracle. The tool provides real-time monitoring of database activities, categorizing attack types as mentioned before, assessing risk levels and offers prevention recommendations. These techniques help to ensure that only safe inputs are processed by the database, keeping it secure. The research will observe how successful the prevention techniques are to detect SQLIA on databases by calculating the successful rate.

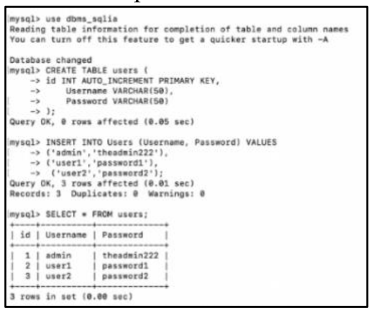
#### *2) Literature Review and Analysis*

A thorough literature review from IEEE Xplore, ResearchGate Net and Science Direct was carried out to understand and prevent SQLIA. This research goes through studies like database security, the latest techniques for preventing SQLIA and the consequences of SQL injection attacks. The proposed solutions in this study aim to address these shortcomings by increasing the accuracy of detecting malicious queries and providing prevention recommendation for the user. Not only that, but it is also capable of monitoring the concurrent database and protecting the database from malicious queries.

#### *3) Data Collection*

This step involves creating a test database to simulate SQLIA scenarios in a controlled environment. A simple database named dbms_sqlia was set up in MySQL, containing three columns: id, username and password. This setup allowed for the systematic testing of normal and malicious inputs.

In addition to testing regular login details like username: admin and password: 123, various malicious inputs were tested. These inputs were designed to exploit weaknesses in the system. Vulnerabilities in the login system are identified by using these harmful inputs.

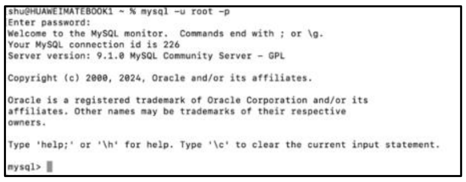


```
mysql> use dbms_sqlia
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> CREATE TABLE users (
    -> id INT AUTO_INCREMENT PRIMARY KEY,
    ->     Username VARCHAR(50),
    ->     Password VARCHAR(50)
    -> );
Query OK, 0 rows affected (0.05 sec)

mysql> INSERT INTO Users (Username, Password) VALUES
    -> ('admin','theadmin222'),
    -> ('user1','password1'),
    ->  ('user2','password2');
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM users;
+----+----------+-------------+
| id | Username | Password    |
+----+----------+-------------+
|  1 | admin    | theadmin222 |
|  2 | user1    | password1   |
|  3 | user2    | password2   |
+----+----------+-------------+
3 rows in set (0.00 sec)
```

Fig. 2. Table and Data Creation

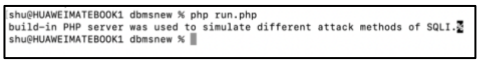
#### *4) Experimental Setup*

The experimental setup was created to simulate SQLIA scenarios in a controlled environment. The experiment is setup by using a build-in PHP server and MySQL. A build-in PHP server was used to run the terminal for easy testing, it is also used for backend coding to handle user login and simulate different attack methods. Then, MySQL is used in this study to manage the database. The database sample of user information like username and password was stored in MySQL. This setup enable the SQL injection vulnerabilities was test accurately and evaluate security solutions in a practical environment.



```
shu@HUAWEIMATEBOOK1 ~ % mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 226
Server version: 9.1.0 MySQL Community Server - GPL

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> 
```

Fig. 3. Simulation on MySQL



```
shu@HUAWEIMATEBOOK1 dbmsnew % php run.php
build-in PHP server was used to simulate different attack methods of SQLI.
shu@HUAWEIMATEBOOK1 dbmsnew % 
```

Fig. 4. Simulation on PHP-based

### 5) *Implementation of Solutions*

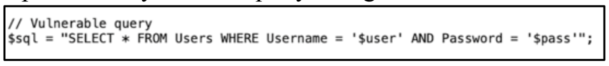
#### a) *Simulating Vulnerable System*

The objective of simulating vulnerable system is to create a basic login system to act as an example of a vulnerable database. The absence of protections like user input validation or sanitization was purposely built into the system. This setup is to test how attackers might exploit the system using SQLI techniques.

A PHP-based script was created to simulate basic login functionality. Within this script, input by users – specifically the username and password, are directly incorporated into an SQL query string without undergoing user input validation or sanitization. A simple HTML-based user interface was implemented to demonstrate the testing of this vulnerable system. The interface collects user inputs for username and password and transmits them to the PHP scripts for processing.

The lack of security designed reflects the types of errors that sometimes occur in systems with inadequate design. By running the experiment, a safely research on how attackers exploit these weaknesses and implement countermeasures in between.

The script processes user input through a login form created by html, specifically the username and password fields. The SQL query is constructed by concatenating these inputs directly into the query string.



```
// Vulnerable query
$sql = "SELECT * FROM Users WHERE Username = '$user' AND Password = '$pass'";
```

Fig. 5. SQL Query Run by Database System

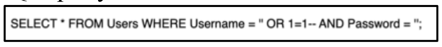
In this approach, the absence of prevention mechanisms of SQL injection attacks allows attackers to manipulate the query. By injecting malicious SQL code into the input fields, username and password, the impact of attacks caused and how the attacks work can be observed. The purpose of this experiment setup is to offer insights into how such vulnerabilities might exist in real-world applications and ensures that it is in a safe and controlled environment.

#### b) *Executing Attack Payloads*

For this experiment, seven types of SQLI payloads were tested: tautologies, logical incorrect queries, union queries, piggy-backed queries, stored procedures, inference, and alternate encodings.

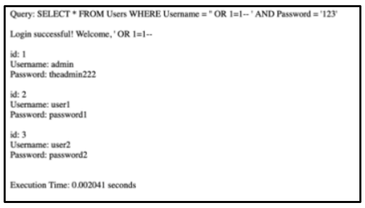
- Tautology-Based Attack

In a tautologies attack, the attacker inputs a malicious input that alters the logic of SQL query, ensuring it always evaluate to true. In this case, the attacker inputs ' OR 1=1-- in the username area. The password in this field could be any value. The comment operator -- is used to ignore the rest of the query after it, but only if a space is entered. Any quotes could be inserted after the space as it is already ignored. The resulting SQL query becomes:



```
SELECT * FROM Users WHERE Username = '' OR 1=1-- AND Password = '';
```

Fig. 6. SQL Query Run in Tautology Case



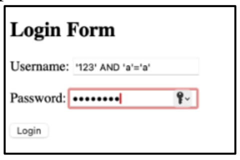
```
Query: SELECT * FROM Users WHERE Username = '' OR 1=1-- ' AND Password = '123'

Login successful! Welcome, ' OR 1=1--

id: 1
Username: admin
Password: theadmin222

id: 2
Username: user1
Password: password1

id: 3
Username: user2
Password: password2


Execution Time: 0.002041 seconds
```

Fig. 7. Results in the website

As a results, the query always returns a valid result as ' OR 1=1-- converts the condition into a tautology as 1=1 is always true. This attack enables the attacker to bypass authentication process without requiring the correct username or password.

- Logical Incorrect Queries

In a logical incorrect queries attack, the attacker tries to mess up the logic of SQL query which lead to an error. This can expose the hidden information about the database, like how the data is structured. In this case, the attacker creates errors by providing data that does not match with the expected output of database. For this experiment, the attackers try to input a string like '123' AND 'a'='a' when the system might expect a number for the username.



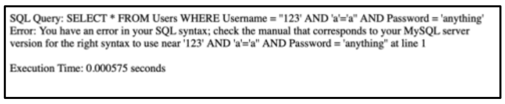
**Login Form**

Username: '123' AND 'a'='a'

Password: ••••••••

Fig. 8. Malicious Input by Attackers for Logical Incorrecy Queires

The system expects a username as a text, but when attacker input ‘123’, the system tries to match it with the database, and the query breaks.



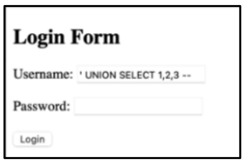
```
SQL Query: SELECT * FROM Users WHERE Username = ''123' AND 'a'='a'' AND Password = 'anything'
Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near '123' AND 'a'='a'' AND Password = 'anything'' at line 1

Execution Time: 0.000575 seconds
```

Fig. 9. SQL Query and Error Details

This results a syntax error, where the database then returns an error message that points out the issue. This is an example of a logical incorrect query attack, where attacker breaks the query in purpose. The objective from this logical incorrect query attack is to force database to return an error, which can reveal useful information to the attackers.

- Union Query

A union-based SQLIA is an attack in which an attacker utilized the UNION operator to join the results of multiple queries. This technique enables unauthorized access to data from tables that are restricted. This technique often exploits how SQL combines a table sets from the original query and the injected statement.



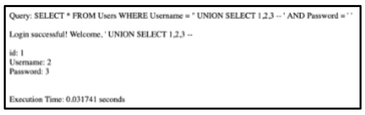
Login Form

Username: ' UNION SELECT 1,2,3 --

Password:

Fig. 10. SQL Malicious Input by Attackers for Union Query

From the diagram above, an attacker input the following SQL, ' UNION SELECT 1,2,3 -- into the username field. The resulting SQL command executed by the system and appeared as:



```
Query: SELECT * FROM Users WHERE Username = '' UNION SELECT 1,2,3 -- ' AND Password = ' '

Login successful! Welcome, ' UNION SELECT 1,2,3 --

id: 1
Username: 2
Password: 3


Execution Time: 0.031741 seconds
```

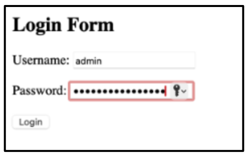
Fig. 11. SQL Query Run and Results in Website

The first part of the query, SELECT * FROM Users WHERE Username = ‘’ AND Password = ‘ ’ becomes ineffective because the provided username and password are invalid. This causes no results are return from the first portion of the query. The second part (UNION SELECT 1,2,3) attempts to inject dummy data 1,2,3 into the query output. Therefore, the results return as id:1, username:2, password:3. The attacker combines the results of the original query with their malicious query.

- Piggy-backed Queries

A piggy-backed SQLI is an attack where an attacker inputs multiple SQL queries together using a semicolon (;) operator. This will allow the attacker to execute one or more additional queries after the intended query. This happens when a semicolon is used to separate the real query from the malicious one.

A table named Usersforpiggytest was created in the dbms_sqlia databases to demonstrate for piggy-backed queries attack. An attacker has input admin into the username field, followed by the query ' OR 1=1; DROP TABLE Usersforpiggytest; -- in the password field. After the semicolon (;), the attackers append the DROP TABLE Usersforpiggytest query. This query will drop the table Userforpiggytest from the database. The semicolon effectively separates the two queries. Since the database executes both queries, hence the table Usersforpiggytest table is dropped. After the query is executed, the table is deleted, hence any attempts to query the table will fail.

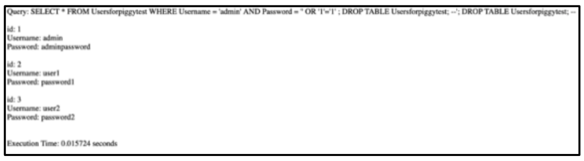


Login Form

Username: admin

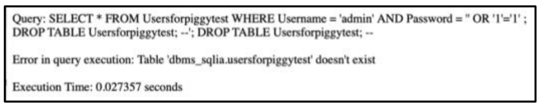
Password: ••••••••••••••••

Fig. 12. Malicious Input by Attackers for Piggy-backed Queries



```
Query: SELECT * FROM Usersforpiggytest WHERE Username = 'admin' AND Password = '' OR '1'='1' ; DROP TABLE Usersforpiggytest; --'; DROP TABLE Usersforpiggytest; --

id: 1
Username: admin
Password: adminpassword

id: 2
Username: user1
Password: password1

id: 3
Username: user2
Password: password2


Execution Time: 0.015724 seconds
```

Fig. 13. SQL Query and Results in Website

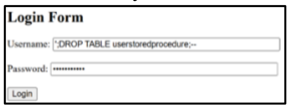


```
Query: SELECT * FROM Usersforpiggytest WHERE Username = 'admin' AND Password = '' OR '1'='1' ;
DROP TABLE Usersforpiggytest; --'; DROP TABLE Usersforpiggytest; --

Error in query execution: Table 'dbms_sqlia.usersforpiggytest' doesn't exist

Execution Time: 0.027357 seconds
```

Fig. 14. Fail to attempt as table has been deleted

- Stored Procedure

Stored procedure is stored inside a database system to acts as a pre-defined set of SQL instructions that the database system will execute to perform out tasks. Common uses of stored procedure are connecting with other parts of the database system and verifying user credentials. However, an attacker can exploit the stored procedure by injecting malicious input into the parameters that used by a stored procedure. An attacker can inject SQL commands such as DROP TABLE or SHUTDOWN [3][5]. The database system may act in an unexpectedly detrimental manner. In this case, a table named userstoredprocedure was created in the dbms_sqlia database to demonstrate SQLIA inside stored procedure. Originally, the database has two tables which are users and userstoredprocedure. A procedure named isAuthenticated will be created. Attacker will inject command such as ';DROP TABLE userstoredprocedure;-- to drop the table. If website is relaunching again, there will be prompt an error message by MySQL shows that the table does not exist. Attackers can use this type of SQLIA to simply drop or shutdown other database system.

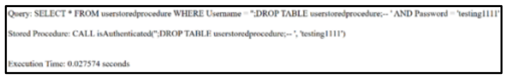


Login Form

Username: ';DROP TABLE userstoredprocedure;--

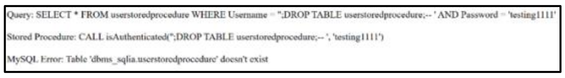
Password: •••••••••••

Fig. 15. Malicious Input by Attackers for Stored Procedure



```
Query: SELECT * FROM userstoredprocedure WHERE Username = '';DROP TABLE userstoredprocedure;-- ' AND Password = 'testing1111'

Stored Procedure: CALL isAuthenticated('';DROP TABLE userstoredprocedure;-- ', 'testing1111')


Execution Time: 0.027574 seconds
```

Fig. 16. SQL Query Run in Stored Procedure Case



```
Query: SELECT * FROM userstoredprocedure WHERE Username = '';DROP TABLE userstoredprocedure;-- ' AND Password = 'testing1111'

Stored Procedure: CALL isAuthenticated('';DROP TABLE userstoredprocedure;-- ', 'testing1111')

MySQL Error: Table 'dbms_sqlia.userstoredprocedure' doesn't exist
```

Fig. 17. Error Details for Relaunching Website

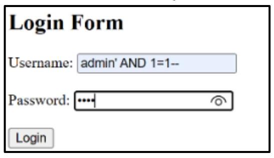
- Inference-based SQL Injection Attack

Inference-based SQLIA which is also known as blind SQLIA is a type of attacks that attackers or hackers can copy the database structure by examining how the web application reacts. Although this SQLIA takes longer time for attacker to exploit, it still considered as a dangerous SQLIA. There are two main types of inference-based which is Boolean-based and Time-based. Each of them has different attack techniques. This types of SQLIA are focusing on inferencing the structure of database system rather than extracting the data or information [21].

**Boolean-based**

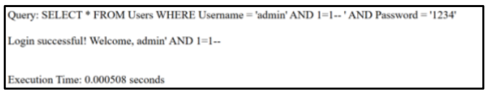
Boolean-based SQLIA is a type of SQLIA that sending a SQL query to the database system in which prompts the application to return a different type of replies based on whether the query will return TRUE or FALSE. Due to the unsecure database system, the security of the database system has been improved by the developer to block the error message as shown at the website. The attackers will not receive any error message due to this factor [21]. Therefore, Boolean-based SQLIA is used to know whether the database structure have vulnerabilities or not.

In this case, same database system named 'dbms_sqlia' and 'users' table which shown at Figure 2 will be used to demonstrate Boolean-based SQLIA.



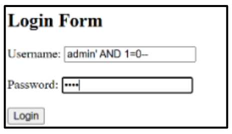
```
Login Form

Username: admin' AND 1=1--
Password: ••••
Login
```

Fig. 18. Malicious Input by Attackers for Boolean-based (Case 1)



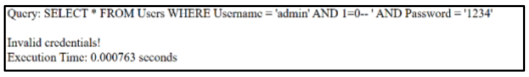
```
Query: SELECT * FROM Users WHERE Username = 'admin' AND 1=1-- ' AND Password = '1234'

Login successful! Welcome, admin' AND 1=1--


Execution Time: 0.000508 seconds
```

Fig. 19. SQL Query Run and Results in Website



```
Login Form

Username: admin' AND 1=0--
Password: ••••
Login
```

Fig. 20. SQL Malicious Input by Attackers for Boolean-based (Case 2)



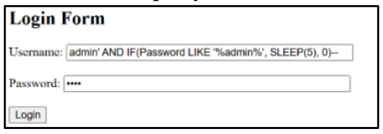
```
Query: SELECT * FROM Users WHERE Username = 'admin' AND 1=0-- ' AND Password = '1234'

Invalid credentials!
Execution Time: 0.000763 seconds
```

Fig. 21. SQL Query Run and Results in Website

From the figure shown above, there are two cases to show how attackers use Boolean-based SQLIA to know the vulnerability of a database system. The attacker tries to inject 'admin ' AND 1=1-- ' or 'admin ' AND 1=0-- 'to the username filed [19]. Even though the attackers may inject wrong password for 'admin', the system will not check for the password because the password field is ignored. The 'AND 1=1 shows that the attacker is successfully login to 'admin' while the 'AND 1=0 shows that the attacker is unsuccessfully to login to 'admin'. From above result, the attacker can simply know whether a database system is vulnerable or not by using Boolean-based SQLIA. Although it is not extracting the data, but the attacker can inference the database structure and reconstruct the database system again to steal or delete the secret data and information.

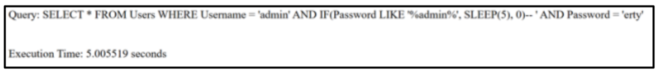
**Time-based**

Time-based SQLIA is a type of SQLIA that attackers can create a SQL query purposefully causes a delay in the query execution on the database system. Whether the query condition evaluates to true or false is shown by the response time. It is same with the concept of Boolean-based SQLIA where developers have improved the security of database system, therefore error message will not be appeared. Time-based can also be one of the attacks used by attackers to inject malicious input such as WAITFOR or SLEEP to force a delay of the database system. If the response is delayed, it shows that the query condition is TRUE. If the response is no delayed, it shows that the query condition is FALSE.



```
Login Form

Username: admin' AND IF(Password LIKE '%admin%', SLEEP(5), 0)--
Password: ••••
Login
```

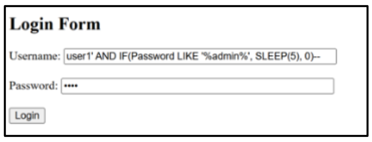
Fig. 22. Malicious Input by Attackers for Time-based (Case 1)



```
Query: SELECT * FROM Users WHERE Username = 'admin' AND IF(Password LIKE '%admin%', SLEEP(5), 0)-- ' AND Password = 'erty'


Execution Time: 5.005519 seconds
```

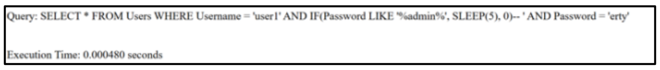
Fig. 23. SQL Query Run and Results in Website

From the figure shown above, the attacker tries to inject admin' AND IF(Password LIKE '%admin%', SLEEP(5), 0)-- to the input field. The attacker wants to check whether the password for 'admin' user has consists of 'admin' or not [5]. The test scenarios shown that the password for 'admin' consists of the word of 'admin'. The SLEEP(5) will be executed to slower than the database system for about 5 seconds. When relaunching the website, a slower execution time can be determined.



```
Login Form

Username: user1' AND IF(Password LIKE '%admin%', SLEEP(5), 0)--
Password: ••••
Login
```

Fig. 24. Malicious Input by Attackers for Time-based (Case 2)



```
Query: SELECT * FROM Users WHERE Username = 'user1' AND IF(Password LIKE '%admin%', SLEEP(5), 0)-- ' AND Password = 'erty'


Execution Time: 0.000480 seconds
```

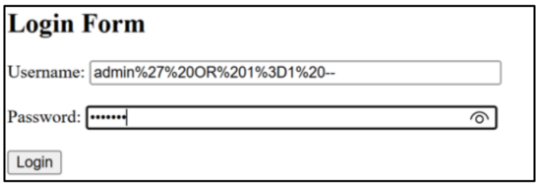
Fig. 25. SQL Query Run and Results in Website

Another test scenarios can be executed to see the differences. In here, attack wishes to know whether password for 'user1' has the word of 'admin' or not. Above figure has shown that the password for 'user1' does not consists of the word of 'admin'. This can be observed that the execution time is less than 5 seconds. Therefore, the SQL query is FALSE.

- Alternative Encodings

This SQLIA is the encoding methods that alter the representation of characters or strings in ways that make them more difficult for detection. Common alternative encoding methods such as ASCII, hexadecimal, URL encoding, and Unicode can be used to enable an attack escape detection method that just look for the specific known "bad character". The example of "bad character" is single quotes, semicolon, and dashes. Attackers can hide their injection payload and make it difficult for the system to detect and filter. The

developer might look for the single quotes that encoded in ASCII and fail to notice that the attacker can use ASCII to encode the same character [2]. The purpose of this types of attacks is to evade the detection by automated prevention systems and secure defensive coding. It usually combined with other attack techniques such as tautology and so on.
In this case, URL encoding is being injected in the username input field which is admin%27%20OR%201%3D1%20-- , the database system will convert the URL encoding into plain text and results in admin' OR 1=1 -- . By using this type of SQLIA, attackers can hide their malicious input in the URL encoding. The developers may not be aware of the URL encoding consists of malicious input. Figure shown below shows that the attacker successfully injects malicious input by using URL encoding and login successfully for the user 'admin'.



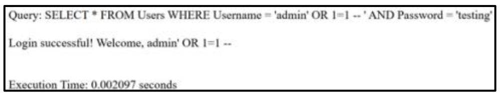
**Login Form**

Username: admin%27%20OR%201%3D1%20--

Password: •••••••

Login

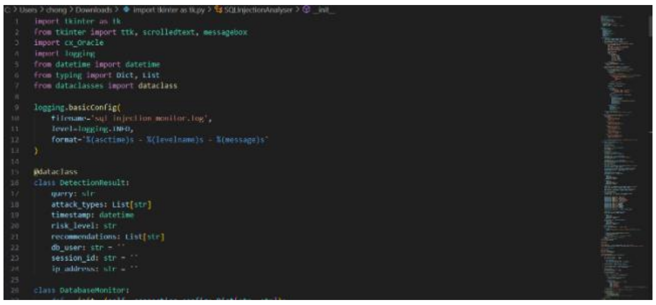
Fig. 26. Malicious Input by Attackers for Alternative Encodings



```
Query: SELECT * FROM Users WHERE Username = 'admin' OR 1=1 -- ' AND Password = 'testing'

Login successful! Welcome, admin' OR 1=1 --

Execution Time: 0.002097 seconds
```

Fig. 27. SQL Query Run and Results in Website

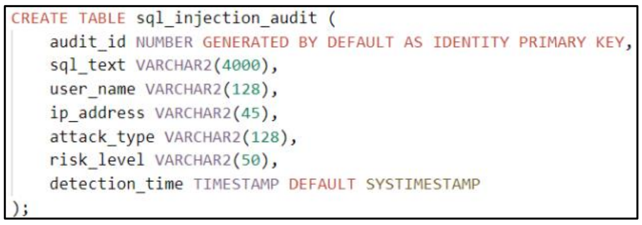
With the research provided, it helped to quickly develop SQLDetc as the solution of the SQLIAs are already available. The SQLDetc is developed with Python.



```
C: > Users > chong > Downloads > import tkinter as tk.py > SQLInjectionAnalyser > __init__
import tkinter as tk
from tkinter import ttk, scrolledtext, messagebox
import cx_Oracle
import logging
from datetime import datetime
from typing import Dict, List
from dataclasses import dataclass

logging.basicConfig(
    filename='sql_injection_monitor.log',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

@dataclass
class DetectionResult:
    query: str
    attack_types: List[str]
    timestamp: datetime
    risk_level: str
    recommendations: List[str]
    db_user: str = ''
    session_id: str = ''
    ip_address: str = ''

class DatabaseMonitor:
```

Fig. 28. Code of SQLDetc

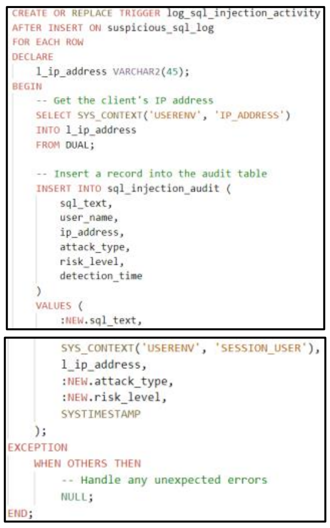
The implementation of the tool requires some setting up from the user. Firstly, a user with privileges that can create an audit table.



```
CREATE TABLE sql_injection_audit (
    audit_id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    sql_text VARCHAR2(4000),
    user_name VARCHAR2(128),
    ip_address VARCHAR2(45),
    attack_type VARCHAR2(128),
    risk_level VARCHAR2(50),
    detection_time TIMESTAMP DEFAULT SYSTIMESTAMP
);
```

Fig. 29. Audit for tracking

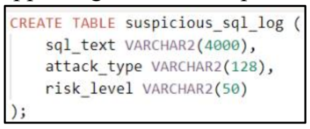
Next, create the trigger for logging suspicious activities.



```
CREATE OR REPLACE TRIGGER log_sql_injection_activity
AFTER INSERT ON suspicious_sql_log
FOR EACH ROW
DECLARE
    l_ip_address VARCHAR2(45);
BEGIN
    -- Get the client's IP address
    SELECT SYS_CONTEXT('USERENV', 'IP_ADDRESS')
    INTO l_ip_address
    FROM DUAL;

    -- Insert a record into the audit table
    INSERT INTO sql_injection_audit (
        sql_text,
        user_name,
        ip_address,
        attack_type,
        risk_level,
        detection_time
    )
    VALUES (
        :NEW.sql_text,
        SYS_CONTEXT('USERENV', 'SESSION_USER'),
        l_ip_address,
        :NEW.attack_type,
        :NEW.risk_level,
        SYSTIMESTAMP
    );
EXCEPTION
    WHEN OTHERS THEN
        -- Handle any unexpected errors
        NULL;
END;
```

Fig. 30. Trigger for suspicious activity

Then, create a supporting table for suspicious SQL log



```
CREATE TABLE suspicious_sql_log (
    sql_text VARCHAR2(4000),
    attack_type VARCHAR2(128),
    risk_level VARCHAR2(50)
);
```

Fig. 31. Table Creation for Suspicious SQL Log

The *sql_injection_audit* table is designed to store details about suspicious queries like the SQL text, user information, IP address, attack type, risk level and timestamp. To automate this process, the *log_sql_injection_activity* trigger monitors insert into the *suspicious_sql_log* table and logs these details into the audit table. Additionally, the trigger uses the *SYS_CONTEXT('USERENV', 'IP_ADDRESS')* function to capture the client's IP address which makes it easier to track potential SQL injection attempts. Any logical altercation can be made.

After this, users need to install cx_Oracle for Python, Oracle SQL Developer for database management, and Oracle Instant Client for database connectivity. With that, the user can run the python script then setup database connection in *db_config* and just like that user should be able to use the tool.

The GUI of the tool is built with Tkinter and it features two tabs, Query Analysis and Database Monitoring.

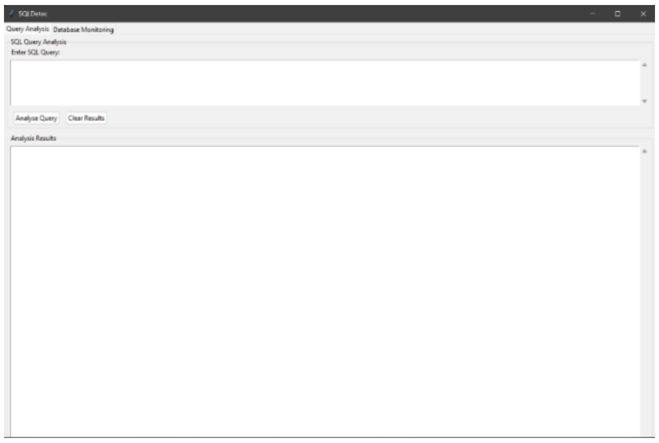


Fig. 32. SQLDetc GUI

The Query Analysis tab allows users to input and evaluate queries for SQL injection risks, displaying results with attack types, risk levels and prevention recommendations. The Database Monitoring tab dynamically fetches recent suspicious queries from the audit table and refreshes every 30 seconds to provide real-time updates. A *DetectionResult* data class shows the results while a recommendation system offers targeted suggestions. The tool provides clear user feedback and shows whether a query is safe or dangerous. Lastly, the results and insights of the test for SQLDetc will be mentioned in the next section.

#### *c) Implementing Countermeasures*

SQLDetc used multiple prevention techniques to be built as efficient and effective as it is. One of the many that was use is parameterised queries which separate SQL code from user input, thus prevent SQLi risks [1], [2]. Moreover, using stored procedures helps to keep database operations safe by acting like a secure gatekeeper. Instead of letting users directly interact with the database, they allow access only through predefined instructions. This reduces the chance of directly interacting with SQL statements and makes executing queries more secure [4], [5]. If does have a SQLIA bypass, it can be easily traced as it integrates the IP and user context logging of their session by using *SYS_CONTEXT('USERENV', 'IP_ADDRESS')* [3], [6]. Auditing is also implemented which stores critical details of flagged queries such as the SQL text, user information, IP address, attack type, risk level and timestamp for post-incident analysis so that we can learn from it [2], [8]. Furthermore, we have implemented real-time monitoring is with triggers which automatically flags suspicious queries during database interactions [7], [9]. Additionally, SQLDetc provides risk assessment and prevention recommendations for users [11], [13]. Lastly, the graphical user interface (GUI) allows users to analyse queries for SQL injection risks, monitor suspicious activity in real time and report back to the use [6], [14]. These are the techniques used to develop SQLDetc.

## *D. Tools and Technologies Used*

The experiments were carried out using several tools and technologies. First, MySQL was used as a database management system to store and retrieve user data. It enables the creation and modification of data. PHP served as a programming language for the simulation of SQLIA. The testing environment was hosted on a built-in PHP server, providing a simple and effective local server for running the queries. These tools played an integral role in identifying system vulnerabilities. As for SQLDetc, the prevention tool developed by the team mainly uses python is fundamentally built using Python due to its app-friendly nature and integration capabilities. it was integrated with Oracle technologies like cx_Oracle, Oracle SQL Developer and Oracle Instant Client so that it can be integrated into Oracle databases. Oracle was selected as the first DBMS since it was something familiar to the team as it was the DBMS used throughout the semester, both previously and currently. Additionally, our tests were conducted using Oracle Database 19c, ensuring compatibility and practical validation of the tool.

## *E. Flow Diagram or Architecture*

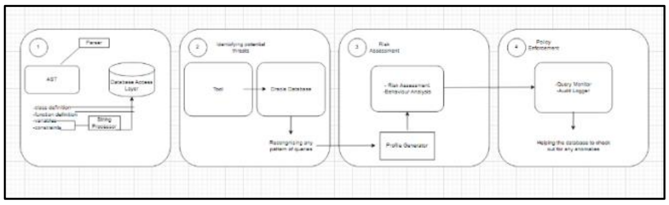


Fig. 33. SQLDetc extracts SQL queries from web application, creates a mapping between query patterns, generates profiles based on pattern analysis and enforces these profiles through Oracle DBMS wiht an plugin for monitoring real time and SQLIA prevention.

#### *1) System Overview of SQLDetc*

SQLDetc is a comprehensive security tool that combines static analysis, dynamic monitoring and real-time enforcement to protect DBMS. In step 1 depicted in Figure above, SQLDetc system starts with the static analysis layer where incoming SQL queries are processed through a parsing mechanism. This mechanism utilizes an Abstract Syntax Tree (AST) generation process to deconstruct each query into its very basic. The parsing process includes string normalization to make sure a consistent pattern can be found. This normalization step is important as it helps the tool to build a map of any potential attacks. By doing so, the system establishes a baseline of normal query behavior and identifies any anomalies like possible SQLIAs attempts.

In step 2, SQLDetc uses a database of 157 common SQL injection patterns. These patterns cover all kinds of sneaky attacks attackers might use and each one is tagged with a risk level and detection metric. The pattern matcher works by combining regex-based detection by finding exact matches in query text with semantic analysis which makes it reliable. By using machine learning, the system adapts to new attack styles.

In step 3, profile generation is where the system creates detailed behavioral profiles for normal database operations. These profiles are generated by analyzing historical query patterns, mapping function calls to their corresponding SQL so that the tool can look out for any anomalies.

In step 4, the enforcement module represents the final layer of defense, implementing real-time monitoring and protection through an Oracle database plugin. This component intercepts and analyzes queries before they reach the database, applying the profiles and patterns established in earlier stages to prevent malicious queries from executing.

## *F. Challenges and Limitations*

There are several challenges and limitations were encountered during the research on SQL injection attacks. The most significant limitation was this study relied completely on synthetic data, which is generated artificially rather than real data from the live systems. The main purpose of choosing synthetic data is to avoid legal or privacy concerns. Synthetic data is a perfect use to simulate the queries, but it also brings disadvantages in study, as these data does not fully represent the real-world scenarios. Real data would have provided more accurate insights but using it could raise ethical and privacy issues. Lastly, not all possible types of SQLI were explored within the research. Only specific types of SQLIA were focused on this research, such as classic and blind injections. Therefore, the study might not demonstrate all the types of SQLIA that attackers used to exploit vulnerabilities in systems.

### G. *Summary*

In summary, the research methodology examines the vulnerability of SQL database system that focuses on several types of SQLIA. The execution time of each test case scenarios is being recorded to test the scalability of database system. Besides, the SQL Injection Detection Tool also being proposed to protect the Oracle database system from SQLIA. Combination of static, dynamic, and real-time enforcement are used in this method. Detection is also done inside the methods and detection rate is recorded which can be observed. From the result, observations can be done according to the detection rate is about 91.7% to 97.8% with false positives are under 3% in a controlled environment. The research methodologies emphasise the importance of customised prevention methods to protect the database system against SQLIA. Future research is needed to concentrate on testing prevention methods in a real-world environment to improve the resilience and adaptability of a database security system.

## IV. RESULTS AND DISCUSSION

### A. *Introduction to Results and Discussion Section*

This section discusses the results of demonstrating different types of SQLIA and evaluating the effectiveness of SQLDetc. It compares these SQLIA methods based on the execution time, vulnerability impact, database integrity and error messages. By doing so, it highlights the potential risks associated with each type of SQLIA. By understanding these details allowed the team to understand how effective is SQLDetc at preventing SQLIA. Tests for SQLDetc were done in controlled setups and simulating a real-world scenario with 500 connections running 2,000 queries per minute over three days in Oracle DBMS. The effectiveness of SQLDetc keeping the database safe in a system is analyzed in the results.

### B. *Presenting Results*

The results shown at Table III are organised to highlight the important aspect of SQLIA. The overview of aspect in the results table include execution time, vulnerability impact, and the effect on database integrity. Execution time measures the time taken for the database to process the query and return the results. The purpose for these measures is to evaluates the performance impact and shows the efficiency of the attack. Vulnerability impact refers to the effect of exploiting a vulnerability. It is categorized as High, Medium, or Low based on the level of damage it can cause to the database. The effect on database integrity describes where the attack modifies or damage the database. The effect is categorized by destructive attacks and non-destructive attacks. Destructive attacks such as piggy-backed queries can delete or alter data, while non-destructive attacks only read data without altering. Finally, error message details reveal the database's response to specific SQLI attacks. The purpose of error message details is to determines whether the attack reveals sensitive information about the database.

TABLE III. IMPORTANT ASPECT OF SQLIA

| **Types of SQL Injection** | **Execution Time (MS)** | **Vulnerability Impact** | **Effect on Database Integrity** | **Error Message Details** |
|---|---|---|---|---|
| Tautology | 0.002 | High | Non-Destructive | Login bypass |
| Logical Incorrect Queries | 0.001 | Low | Non-Destructive | Syntax Error |
| Union Queries | 0.032 | High | Non-Destructive | Return additional rows from tables |
| Piggy-backed Queries | 0.027 | High | Destructive | Database table altered |
| Stored procedure | 0.028 | High | Destructive | Database table deleted or system shut down |
| Boolean-based | 0.005-0.007 | Medium | Non-Destructive | Return TRUE or FALSE |
| Time-based | TRUE-Longer time FALSE-Shorter time | Medium | Non-Destructive | Execution time being observed by attackers |
| Alternative encodings | 0.002 | High | Destructive and Non-destructive | Bypass security measures |

Moreover, Table V shown are the findings of SQLDetc effective in high-concurrency environment was set up using Oracle Database 19c. The evaluation strategy for SQLDetc aimed to see its ability to detect a wide variety of SQLIA types and measure its impact on the database. The test was done involving 500 concurrent connections executing 2,000 queries per minute over a 3-day testing period. The system was exposed to both benign and malicious queries, to test its ability to maintain high detection rates while ensuring that legitimate traffic was unaffected.

TABLE IV. SQLDETC DETECTION ACCURACY ON SQLIA TYPES

| **Attack Type** | **Detection Rate** | **False Positives** |
|---|---|---|
| Union Based | 97.8% | 1.2% |
| Tautology | 96.5% | 1.5% |
| Time Based | 94.3% | 1.8% |
| Piggy-Back | 95.4% | 1.3% |
| Boolean-based | 92.1% | 2.0% |
| Time-based | 93.5% | 1.9% |
| Alternate Encodings | 91.7% | 2.3% |
| Stored Procedures | 96.3% | 1.6% |

TABLE V. TESTING RESULTS OF SQLDETC

| **Metric** | **Value** |
|---|---|
| Test Environment | Oracle Database 19c |

| | |
|---|---|
| Concurrent Connections | 500 |
| Queries per Minute | 2,000 |
| Total Attacks Attempted | 1,200 |
| Successfully Detected | 1,150 |
| Detection Rate | 95.8% |
| False Positives | 25 |
| Average Response Time | 4.5ms |
| Blocked Attacks | 1,150 |
| Prevention Rate | 100% of detected |
| False Blocks | 7 |
| System Downtime | 0% |
| Testing Duration | 3 days |

### C. *Comparative Analysis and Discussion*

Several evaluation metrics such as execution time, vulnerability impact, effect on database integrity, and error message details have been evaluated on SQLIA. Union query has the most longer execution time compared to other SQLIA because it involves merging data from multiple queries. Second and third to union would be piggy-backed queries and stored procedure as both involve complex logic execution such as DROP TABLE. Besides, tautology, logical incorrect query and alternative indicates the shortest execution time. The execution time for Boolean-based and Time-based is based on the condition results either TRUE or FALSE in Time-based SQLIA. Moreover, the vulnerability impact seen from logical incorrect query is low. Other than that, vulnerability impact for Boolean-based and Time-based is medium because the focus is on inferencing the database. Lastly, vulnerability impact that is high are tautology, union query, piggy-backed query, stored procedure, and alternative encodings. On the effect on database integrity, Piggy backed queries, and Stored Procedures are classified as destructive because they alter or delete database tables. Non-destructive attacks like tautologies and time-based queries do not harm the database directly. While alternative encodings attacks have both non-destructive and destructive impact. If logical incorrect queries were inputted, the system will just generate detailed syntax errors that identified vulnerabilities while Boolean-based queries is less informative as it returns either a TRUE or FALSE.

Based on the results, shorter execution times in SQLIA indicate fewer operations and not much damage while longer times suggest complex operations. Piggy-backed queries and stored procedures pose higher risks, leading to data loss or corruption and significantly harming database integrity. Lastly, the type of error messages depends on the specific SQLIA used. As for SQLDetc performed well with a detection rate of 95.8% over 3 days. This proves that SQLDetc is an effective tool.

### D. *Overall Discussion*

In summary, the findings highlight that piggy-backed queries and stored procedures pose the highest vulnerability on database integrity while tautologies and union queries primarily focus on data extraction with minimal harm. Although logically incorrect queries exploit syntax errors, Boolean-based and time-based attacks focus on inferring schema information and alternate encodings aim to evade detection mechanisms. This implies the significance in securing DBMS. Proper prevention methods such as query parameterization and input sanitization should be enforced as discuss. With SQLDetc developed by the team, demonstrated high efficacy during testing, achieving a 95.8% detection rate and blocking 1,150 of 1,200 attempted attacks while maintaining zero downtime with an average response time of 4.5ms. There are also limitations occurring in this research study such as the use of a controlled environment for SQLIAs and a simulation of the real-world for SQLDetc, it lacks the diversity and unpredictability of real-world application. While promising, future research should focus on diverse and real-world application.

### E. *Visual Representation of Results*

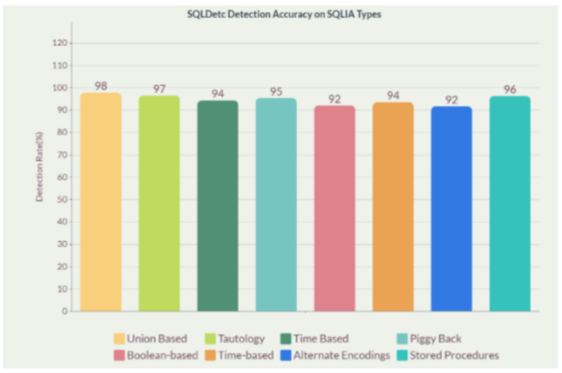


Fig. 34. Detection Accuracy on SQLIA for SQL Detc Tool

### F. *Critical Evaluation*

The study aims to give a thorough understanding of the vulnerabilities presented by SQLIAs and provide a solution to enhance the security for DBMS which is SQLDetc developed by the team. The findings and results of this study support the goals of this research study. According to the results of this study, there is a relationship between the execution time and scalability. For example, piggy-backed query and stored procedure are more destructive and have longer execution time which will impact the performance of a database system. As for the solution, SQLDetc not only demonstrated high efficacy of 95.8% in the test. SQLDetc is built with 157 common SQLI patterns which helps it to block SQLIA. Overall, the findings and results are aligned with the initial objectives.

### G. *Linking to Literature Review*

Many of results and discussion presented are directly linked to the findings in the literatures. The types of SQLIA evaluated in this study such as tautology, piggy-backed queries, and stored procedures which align with the vulnerabilities mentioned in [1], [3], [6]. The high execution times observed for union-based queries and piggy-backed queries reflect the limitations discussed in [2], [7], [15]. Additionally, the impact on database integrity for destructive attacks like stored procedures and piggy-backed queries can be found in [4], [10], [16].

SQLDetc's detection accuracy is supported by research into SQLI prevention mechanisms like pattern recognition and anomaly detection as shown in [5], [8], [12]. The high detection rate (95.8%) and small false positives with similar tools described in [9], [11], [13], demonstrating SQLDetc's effectiveness in handling SQLIA in a high-concurrency environment.

However, simulations may lack the diversity of real-world applications, a limitation discussed in [19], [20]. By linking these findings to the literature, this study validates the relevance of SQLDetc as a solution for enhancing database security and mitigating the risks associated with SQLIA.

### H. *Conclude the Results and Discussion*

In conclusion, this study compared SQLIA types and from that piggy-backed queries and stored procedures have the highest vulnerability impact on database integrity. Tautologies and union queries also posed high risks by bypassing authentication and extracting data. Other SQLIA types exploited vulnerabilities to gather information with less direct harm. The results show the importance of prevention measures in DBMS and validate SQLDetc's efficacy in preventing such SQLIA from happening with a high detection rate and minimal false positives.

To ensure fellow researchers to test and validate the regression models, the demonstration and prevention code has been made publicly available at the following repository: https://github.com/bohe1823/DBMS.git

## V. Conclusion and Future Work

This study demonstrates how SQLIA is a significant database threat. This is because it provides the ability for attackers to manipulate queries and gain unauthorized access to sensitive data. From the research, it is showed that certain SQLIAs such as tautologies and union queries can easily bypass authentication to extract data, piggy-backed queries and stored procedures can alter and delete while attacks such as Boolean-based and time-based queries exploit vulnerabilities to reveal database details. The study highlights the effectiveness of SQLDetc in preventing specific types of SQLIA and have achieve a detection rate of 91.7%-97.8% corresponding to the SQLIA. By observing various SQLIA and implementing countermeasures, the research gives a clear picture of the risks involved and has offered effective solutions to secure DBMS.

This research makes several important contributions to SQLIA prevention. It provided an analysis of various SQLIA types by showing their impact on database security and integrity. The study also introduces SQLDetc, a tool developed to effectively detect and prevent SQLIAs, achieving high detection rates ranging from 91.7% to 97.8% corresponding to the SQLIA types. Additionally, the research validates SQLDetc's performance through a simulated real-world testing. Finally, this study establishes a methodological framework for evaluating and comparing SQLIA prevention which can be used as a foundation for future research in database security.

The findings of this research have significant practical for SQLIA prevention. By demonstrating the effectiveness of SQLDetc in detecting and preventing various SQLIAs, the study offers insights into how organizations can enhance their database security. The high detection rates and low false positive rates achieved by SQLDetc suggest that it can be a for actual real-time protection of databases against SQLIA. Furthermore, the study emphasizes the importance of preventative measure, such as query parameterization to prevent SQLIAs even without SQLDetc. Organizations can apply these findings to strengthen their security infrastructure and reduce the likelihood of successful SQLIA.

This research study encountered various limitations such as the use of synthetic data in simulating the test scenarios for SQLIAs as it may be inaccurate and unable to reflect the complex structure of real-world DBMS since it is based on the predetermined patterns in which would limit the variety inputs that can be tested. Besides, the simulation for SQLIAs by using PHP-based code and MySQL is conducted in a controlled and testing environment that does not consider other external factors such as third-party integration. As for the test of SQLDetc, it was made a simulating real-world, but it is still somewhat controlled. Hence, the data would not be the most accurate. To overcome the limitations of this study, real-world data should be used to demonstrate the SQLIAs and SQLDetc on DBMS. The uses of real-world data can accurately demonstrate how an attackers will attack a system by using SQLIAs. A more authentic illustration of real-world environments can be reached by using real-world data in which can capture the complexity and anomalies of the database system. In addition, future research should focus on producing and proposing more effective and accurate prevention methods to stop the SQLIAs on DBMS.

In conclusion, the findings and results that were presented in this research study reveal a better understanding of how SQLIAs will demonstrate on a database system and proposed a tool, SQLDetc to detect and prevent SQLIAs. Although the findings and results provide important details on how to prevent SQLIAs, future research has to improve the implementation of the prevention methods in order to enhance the integrity and security of database systems.

## Acknowledgment

Chong, Lee, and Tong sincerely appreciate and grateful to course lecturer Dr Sathishkumar Veerappampalayam Easwaramoorthy for his guidance and encouragement on this subject which is Database Management System. He also provides detailed and suggested recommendations about the research study. His guidance has carried the team through all the stages of writing this research study. Lastly, the team also grateful to Sunway University about the unimaginable resources they have. We hope that the findings that being produced in this research paper will contribute to the field of computer science in the future.

## References

[1] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," *International Symposium on Signals, Systems, and Electronics (ISSSE 2006)*, March 2006.

[2] D. A. Kindy and A. K. Pathan, "A Survey on SQL Injection: Vulnerabilities, Attacks, and Prevention Techniques," *2011 IEEE 15th International Symposium on Consumer Electronics (ISCE '11)*, Singapore, June 2011, DOI: 10.1109/ISCE.2011.5973873.

[3] I. Jenal, O. Cheikhrouhou, H. Hamam, and A. Mahfoudhi, "SQL Injection Attack Detection and Prevention Techniques Using Machine Learning," *International Journal of Applied Engineering Research*, vol. 15, no. 6, pp. 569-580, July 2020, ISBN: 0973-4562, Available: https://www.researchgate.net/publication/342734749.

[4] A. Paul, V. Sharma, and O. Olukoya. "SQL injection attack: Detection, prioritization & prevention," *Journal of Information Security and Applications*, vol. 85, p. 103871, Sep. 2024, DOI: 10.1016/j.jisa.2024.103871.

[5] A. Orso, W. Lee, and A. Shostack, "Preventing SQL Code Injection by Combining Static and Runtime Analysis," *Defense Technical Information Center*, May 2008. DOI: 10.21236/ADA482932.

[6] Z. S. Alwan and M. F. Younis, "Detection and Prevention of SQL Injection Attack: A Survey," *International Journal of Computer Science and Mobile Computing*, vol. 6, no. 8, pp. 5-17, Aug. 2017, ISSN: 2320-088X. Available: https://www.researchgate.net/publication/320108029.

[7] M. A. Azman, M. F. Marhusin, and R. Sulaiman, "Machine Learning-Based Technique to Detect SQL Injection Attack," *Journal of Computer Science*, vol. 17, no. 3, pp. 296-303, March 2021, DOI: 10.3844/jcssp.2021.296.303.

[8] H. Shahriar and M. Zulkernine, "Information-Theoretic Detection of SQL Injection Attacks," *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*, pp. 40-47, Oct. 2012, DOI: 10.1109/HASE.2012.31.

[9] A. Falor, M. Hirani, H. Vedant, P. Mehta, and D. Krishnan, "A Deep Learning Approach for Detection of SQL Injection Attacks Using Convolutional Neural Networks" in *Proceedings of Data Analytics and Management*, Singapore, vol. 91, pp. 293-304, Jan. 2022, DOI: 10.1007/978-981-16-6285-0_24.

[10] M. Nasereddin, A. ALKhamaiseh, M. Qasaimeh, and R. Al-Qassas, "A systematic review of detection and prevention techniques of SQL injection attacks," *Information Security Journal: A Global Perspective*, vol. 32, no. 4, pp. 252-265, Oct. 2021, DOI: 10.1080/19393555.2021.1995537.

[11] S. Singh and A. Kumar, "Detection and Prevention of SQL Injection," *International Journal of Scientific Research & Engineering Trends*, vol. 6, no. 3, pp. 1642-1645, May-June 2020, ISSN: 2395-566X.

[12] R. Jahanshahi, A. Doupé, and M. Egele, "You shall not pass: Mitigating SQL Injection Attacks on Legacy Web Applications," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (ASIA CCS '20)*, Taipei, Taiwan, Oct. 2020, DOI: 10.1145/3320269.3384760.

[13] A. Joy and R. M. Daniel, "A Survey on SQL Injection Attack: Detection and Prevention," *International Journal of Science & Engineering Development Research (www.ijrti.org)*, vol. 7, no. 5, pp. 774-779, May 2022, ISSN:2455-2631, Available: https://www.ijrti.org/papers/IJRTI2205126.

[14] A. Abdullah, Muhammad, and M. Malik, "A Survey on SQL Injection Attacks: Detection and Prevention," June 2022, Available: https://www.researchgate.net/publication/361444044.

[15] Y. Haixia and N. Zhihong, "A Database Security Testing Scheme of Web Application," in *Proceedings of 2009 4th International Conference on Computer Science & Education (ICCSE '09)*, Nanning, China, pp. 953-955, July 2009, DOI: 10.1109/ICCSE.2009.5228560.

[16] O. C. Abikoye, A. Abubakar, A. H. Dokoro, O. N. Akande, and A. A. Kayode, "A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm," *EURASIP Journal on Information Security*, Aug. 2020, DOI: 10.1186/s13635-020-00113-y.

[17] F. G. Deriba, A. O. SALAU, S. H. Mohammed, T. M. Kassa, and W. B. Demilie, "Development of a Compressive Framework Using Machine Learning Approaches for SQL Injection Attacks," *Przeglad Elektrotechniczny*, vol. 1, no. 7, pp. 183-189, July 2022, ISSN: 0033-2097, DOI: 10.15199/48.2022.07.30.

[18] A. Mishra, N. Mehre, and S. Dubey, "A Review on SQL Injection, Detection and Prevention Techniques," *Journal of Pharmaceutical Negative Results*, vol .14, no. 1, pp. 1068-1073, Feb. 2023, DOI: 10.47750/pnr.2023.14.S01.148. (DOI Invalid)

[19] B. Nagpal, N. Chauhan, and N. Singh, "A Survey on Detection of SQL Injection Attacks and Their Countermeasures," *Journal of Information Processing Systems*, vol. 13, no. 4, pp. 689-702, Aug. 2017, DOI: 10.3745/JIPS.03.0024.

[20] H. Zhang and X. Zhang, "SQL Injection Attack Principles and Preventive Techniques for PHP Site," in *Proceedings of the 2nd International Conference on Computer Science and Application Engineering (CSAE '18)*, pp. 1-9, October 2018, DOI: 10.1145/3207677.3277958

[21] R. Chandel, "Beginner Guide to SQL Injection Boolean Based (Part 2)," *Hacking Articles*, July 2017, Available: https://www.hackingarticles.in/beginner-guide-sql-injection-boolean-based-part-2/

[22] OWASP, "OWASP Top 10: 2021," *OWASP*, 2021. https://owasp.org/Top10/

## 8. Group Member Contribution

| **Student Name (ID)** | **Percentage of contribution in each section (1-7)** | **Percentage of the overall participation** |
|---|---|---|
| Chong Ting Fung (23101686) | 1(0%), 2(0%) 3(0%), 4(33%), 5(40%), 6(33%), 7(0%), Code SQLDetc | 100% |
| Lee Shu Ann (23099740) | 1 (0%), 2 (70%), 3 (50%), 4 (33%), 5 (30%), 6 (33%), 7 (20%) | 100% |
| Tong Zi Qian (22087852) | 1 (100%), 2 (30%), 3 (50%), 4 (33%), 5 (30%), 6 (33%), 7 (80%) | 100% |