

# SQL Injection and Strategies for Prevention

Tim Cabaza, Jae Chung, Nicholas Mueller, Nicholas Sager  
Southern Methodist University

**Abstract**—Database systems often serve as vulnerable points in websites, making them susceptible to SQL injection attacks. This research paper aims to provide an understanding of SQL injection, demonstrate common attacks using MySQL and Python, and explore preventive measures against such attacks. The study proposes a three-part approach, including an explanation of SQL injection, a demonstration of attacks and their consequences, and an overview of SQL injection prevention techniques. By conducting focused research, building a sample database, and analyzing previous academic literature, the authors aim to contribute to the knowledge base on SQL injection prevention.

**Index Terms**—SQL injection prevention; database security

## I. INTRODUCTION

IN today's digital landscape, websites and online applications heavily rely on database systems to store and manage vast amounts of data. However, these databases can become vulnerable to various security threats, including SQL injection attacks. SQL injection occurs when an attacker maliciously inserts unauthorized SQL code into a website's input fields, tricking the system into executing unintended commands. This can lead to unauthorized access, data breaches, and manipulation of sensitive information. As SQL injection attacks continue to pose a significant risk to the integrity and security of database systems, it becomes crucial to explore preventive measures to mitigate this threat effectively.<sup>1</sup>

The primary objective of this research is to provide a comprehensive understanding of SQL injection attacks and develop effective strategies to prevent them. By examining the principles and techniques behind SQL injection, this study aims to equip database administrators, web developers, and security professionals with the knowledge and tools to safeguard their systems against such attacks. The research will delve into the underlying concepts of SQL injection and investigate preventive measures that can be implemented to mitigate this security risk.

<sup>1</sup>This paper was submitted to Dr. Sohail Rafiqi as part of DS 7330 - File Organization and Database Management at Southern Methodist University on August 8, 2023. T. Cabaza is with Southern Methodist University, Dallas, TX 75205 USA. (e-mail: tcabaza@smu.edu) J. Chung is with Southern Methodist University, Dallas, TX 75205 USA. (e-mail: jgchung@smu.edu) Nicholas Mueller is with Southern Methodist University, Dallas, TX 75205 USA. (e-mail: nmuel@smu.edu) Nicholas Sager is with Southern Methodist University, Dallas, TX 75205 USA. (e-mail: nsager@smu.edu)

The significance of this study lies in its potential to enhance the understanding and awareness of SQL injection attacks among individuals responsible for managing and securing database systems. By identifying the vulnerabilities that make systems prone to SQL injection attacks and providing preventive measures, this research can contribute to the development of robust security practices in the field of database management and web application development. Ultimately, the findings and recommendations of this study aim to mitigate the risks posed by SQL injection attacks, enhance data security, and safeguard the integrity of database systems.

## II. BACKGROUND

The History of SQL injections dates back to the mid-late 1990's, first documented by a cyber security researcher Jeff Forristal. 1990s: As websites and technology became more effective and database-driven, software developers often found little success when trying to properly approve user inputs, which led to exploitable vulnerabilities.

Late 1990s and early 2000s: SQL injection developers became more aggressive with the use of these attacks by developing tools that automated the SQL injection process of exploiting vulnerable website. With the high success rate of these cyber security attacks it took little time for SQL injections to become world wide and highly volatile. 2000s to 2010s: SQL injection continues to be a major threat, leading to several high-profile data breaches. These data breaches were targeted towards large organizations and government information that lacked proper coding practices and poorly designed security code.

Showing Awareness and Security Improvements: As time moved on, developers, increase security awareness and started to perform best practices in order to prevent SQL injection vulnerabilities. Developers used precautions like framework, libraries, and other tools developed specifically to help assist developers write secure code that would leave little to no vulnerability.

Ongoing Threat: Even with present day advancements in website development and code, SQL injections are still a threat. New processes, variations, and techniques of SQL injection continue to be developed and adapted.

## III. THEORY OF SQL INJECTION

SQL Injection is a prevalent cybersecurity threat, with a broad impact on the integrity and security of database-driven applications. This section will delve into the underlying principles of SQL injection, its mechanisms, various types, and potential consequences. [1] [2]

### A. Definition

SQL injection is a code injection technique used by cyber attackers to exploit a vulnerability in a web application's database layer. This method involves inserting malicious SQL statements into an entry field for execution, typically to manipulate the application's interactions with its back-end database. These manipulations can provide unauthorized access to data, leading to potential data breaches, system compromises, and other security incidents.

### B. How it works

SQL injection works by taking advantage of poorly designed or inadequately secured web applications that do not correctly sanitize user input. This allows attackers to input SQL code into fields such as usernames, passwords, or search boxes. Once entered, the malicious code is then executed by the application, leading to unexpected and often detrimental behavior. For example, suppose an application does not properly filter out SQL control characters. In that case, an attacker could use a SQL injection attack to alter a query's logic, bypass login algorithms, or return additional records. This can result in unauthorized access to sensitive information, such as user credentials, personal information, and proprietary data.

### C. Types of SQL Injection

There are several types of SQL injection attacks, including: Union-Based SQL Injection: This type of attack leverages the UNION SQL operator to combine the results of the original query with results from the attacker's injected query, often revealing sensitive information. Error-Based SQL Injection: This method relies on error messages thrown by the database to obtain information about its structure, such as table names and column details. Time-Based Blind SQL Injection: In this form of attack, the attacker sends an SQL query to the database and then measures the time it takes to respond. If the response is delayed, the attacker deduces specific details about the database. Boolean-Based Blind SQL Injection: In this technique, an attacker sends an SQL query to the database and makes deductions based on the application's response (true or false). This type of attack can be time-consuming as it often requires sending many queries to map out the database.

### D. Consequences

The consequences of SQL Injection attacks can be severe, ranging from unauthorized viewing of user data to full system compromise. For businesses, a successful SQL Injection attack can lead to data breaches, loss of proprietary information, and a damaged reputation. In certain cases, attackers can even use the exploited system as a launchpad for further attacks on other systems, escalating the extent of the breach. Moreover, for individuals, a successful SQL injection attack can lead to privacy violations, identity theft, financial loss, and more. Hence, the importance

of proper security measures in preventing such attacks cannot be overstated.

## IV. DEMONSTRATION OF SQL INJECTION USING MYSQL AND PYTHON

### A. Sample Database

The database we are using for demonstration purposes is the World Wide Importers (WWI) database[3]. The database is a long-running demonstration tool developed by Microsoft for their MSSQL and Azure database platforms. The database is provided under the MIT license, meaning anyone is free to use or modify it. World Wide Importers is a fictitious wholesaler company located in San Francisco, CA. Microsoft's documentation describes WWI as follows:

Wide World Importers (WWI) is a wholesale novelty goods importer and distributor operating from the San Francisco bay area.

As a wholesaler, WWI's customers are mostly companies who resell to individuals. WWI sells to retail customers across the United States including specialty stores, supermarkets, computing stores, tourist attraction shops, and some individuals. WWI also sells to other wholesalers via a network of agents who promote the products on WWI's behalf. While all of WWI's customers are currently based in the United States, the company is intending to push for expansion into other countries/regions. WWI buys goods from suppliers including novelty and toy manufacturers, and other novelty wholesalers. They stock the goods in their WWI warehouse and reorder from suppliers as needed to fulfill customer orders. They also purchase large volumes of packaging materials, and sell these in smaller quantities as a convenience for the customers.

[4] For our purposes, we have modified the source database for use with MySQL rather than MSSQL. We have also limited the records to 100 rows per table, as large amounts of data are not necessary to demonstrate SQL Injection.

1) *Schema*: The version of the World Wide Importer's database we are using consists of four schemas. They are Application, Purchasing, Sales, and Warehouse. The schemas are designed to simulate a fully functioning wholesale import / export business and are complex. All tables have single-column primary keys, and one sequence of TransactionID is used for all tables involving transactions (CustomerTransactions, SupplierTransactions, StockItemTransactions). More details can be found in the documentation[4]. For demonstration purposes, we combine the schema into a single database called WWI and append the original schema as a prefix to the table name.

The application schema contains information on people within and outside the organization. It also contains reference tables with information such as cities and states. The

People table is noteworthy because it contains sensitive employee and customer info and could be a target for SQL Injection. The Purchasing schema contains details about transactions and suppliers. The Sales schema has details about customers, orders, and stock of items. This is another target for SQL injection, as the Customer table contains sensitive banking information. The Warehouse schema contains details about inventory.

2) *Potential Attack Vectors:* The WWI database is not intended to be accessed directly for security reasons. In the original MSSQL implementation, it is designed to be accessed through a combination of applications, views, and PowerBI. For demonstration purposes, we will show attacks on the database through direct access via python as well as how attacks can circumvent a view. This should be interpreted as serves as stand in for malicious code entered into a application or field on a website. One area of further research could be attacking a view that has restricted (non-administrator) access to the database as this would be a more realistic scenario.

#### B. Attacking the sample database using Python

The first attack vector we will demonstrate is how the database can be modified using SQL statements embedded in a query.

For this SQL Injection example, we will DROP the table Applications.People (boxed in yellow in fig. 1) with our query to show all tables from our database to demonstrate how SQL injection is simply multiple commands that are embedded in a data structure to alter or extract data from the database.

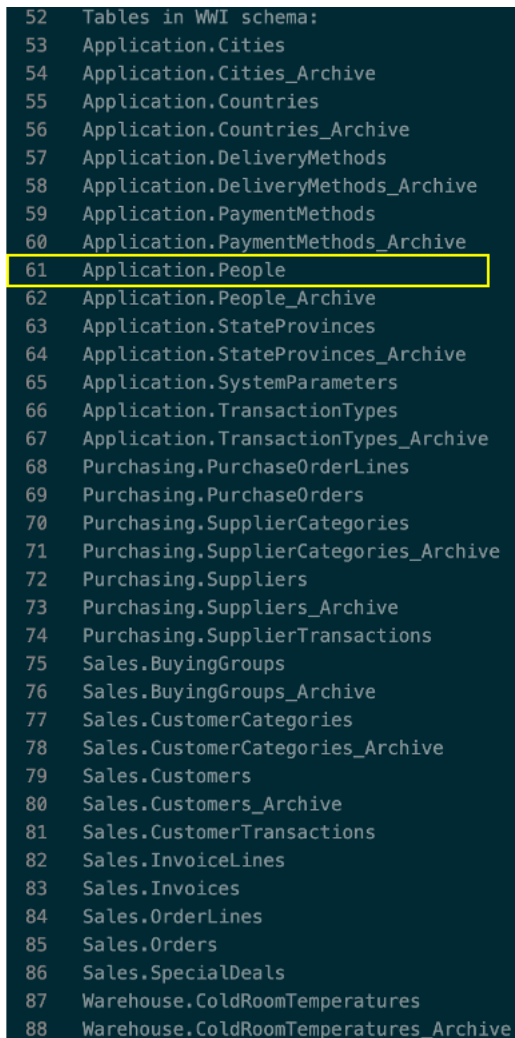
In this example, the user does not practice proper parameterization to prevent an SQL injection attack but instead enters the table name embedded directly into the f-string. The user has constructed a query that allows for the query's dynamic variable (table\_name) to be easily changed without altering the rest of the query itself. The hacker is able able to take control of the variable table\_name and change the value to maliciously alter the database by dropping the table Application.People.

##### # Example 1 - SQL INJECTION ATTACK

```
from getpass import getpass
from mysql.connector import connect, Error

# Establish the connection
try:
    connection = connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: ")
    )
    print("Connection established successfully.")

    try:
        with connection:
```



```
52 Tables in WWI schema:
53 Application.Cities
54 Application.Cities_Archive
55 Application.Countries
56 Application.Countries_Archive
57 Application.DeliveryMethods
58 Application.DeliveryMethods_Archive
59 Application.PaymentMethods
60 Application.PaymentMethods_Archive
61 Application.People
62 Application.People_Archive
63 Application.StateProvinces
64 Application.StateProvinces_Archive
65 Application.SystemParameters
66 Application.TransactionTypes
67 Application.TransactionTypes_Archive
68 Purchasing.PurchaseOrderLines
69 Purchasing.PurchaseOrders
70 Purchasing.SupplierCategories
71 Purchasing.SupplierCategories_Archive
72 Purchasing.Suppliers
73 Purchasing.Suppliers_Archive
74 Purchasing.SupplierTransactions
75 Sales.BuyingGroups
76 Sales.BuyingGroups_Archive
77 Sales.CustomerCategories
78 Sales.CustomerCategories_Archive
79 Sales.Customers
80 Sales.Customers_Archive
81 Sales.CustomerTransactions
82 Sales.InvoiceLines
83 Sales.Invoices
84 Sales.OrderLines
85 Sales.Orders
86 Sales.SpecialDeals
87 Warehouse.ColdRoomTemperatures
88 Warehouse.ColdRoomTemperatures_Archive
```

Figure 1: Target of Attack

```
cursor = connection.cursor()

cursor.execute("USE WWI")
# cursor.execute("SHOW TABLES")
#Example1
table_name = 'Application.People'
sql_injection_that_alters_database =
    "; DROP TABLE `Application.People`; --"
table_name
    =sql_injection_that_alters_database
query = f"SHOW TABLES LIKE '{table_name}';"
cursor.execute(query)
result = cursor.fetchall()

except Error as e:
    print(e)

except Error as e:
    print("Error connecting to MySQL:", e)
```

```

Connection established successfully.
('Application.Cities',)
('Application.Cities_Archive',)
('Application.Countries',)
('Application.Countries_Archive',)
('Application.DeliveryMethods',)
('Application.DeliveryMethods_Archive',)
('Application.PaymentMethods',)
('Application.PaymentMethods_Archive',)
('Application.People_Archive',)
('Application.StateProvinces',)
('Application.StateProvinces_Archive',)
('Application.SystemParameters',)
('Application.TransactionTypes',)
('Application.TransactionTypes_Archive',)
('Purchasing.PurchaseOrderLines',)
('Purchasing.PurchaseOrders',)
('Purchasing.SupplierCategories',)
('Purchasing.SupplierCategories_Archive',)
('Purchasing.Suppliers',)
('Purchasing.Suppliers_Archive',)
('Purchasing.SupplierTransactions',)
('Sales.BuyingGroups',)
('Sales.BuyingGroups_Archive',)
('Sales.CustomerCategories',)

```

Figure 2: Database after Attack

### C. Results

The hackers' attempt is successful and the table `Application.People` has been maliciously dropped (See fig. 2). We can verify the result in MySQL Workbench as well. The vulnerability in the code lies in how the user directly inputs the SQL query string and thus by mixing the input with the SQL query has opened the door to an attacker to modify or execute any arbitrary SQL Statements. If the user had separated the input from the SQL query, it would have helped mitigate the SQL Injection Vulnerability.

There are other actions that can be done with SQL Injection such as Drop, Insert, and Modify. For example, a user without login rights in the `People` table could be modified to have access to the database and website by changing that value from 0 to 1. There are many other attacks possible that don't involve changing the database schema. For example, the same type of SQL injection could be used to extract or modify banking data for customers from a view that is intended for tracking product inventory.

## V. SQL INJECTION PREVENTION

The final section of the research will delve into the theory and implementation of preventive measures against SQL injection attacks. It will explore various strategies and best practices that can be employed to harden database systems and protect against SQL injection vulnerabilities. The research will cover techniques such as input validation and sanitization, prepared statements, and parameterized

queries. The authors will explain the principles behind each preventive measure and demonstrate their effectiveness in mitigating SQL injection risks. While this section will provide an overview of SQL injection prevention, it is important to note that it may not encompass an exhaustive treatment of all prevention techniques but will serve as a foundation for implementing basic measures to enhance database security.

### A. Theory

Because of the wide variety of databases in use today, there is no one effective strategy for preventing SQL Injection attacks. Defending against SQL Injection requires a multi-pronged approach that includes many facets. Most commonly used today are defensive coding techniques, systems for SQLIA detection, and systems for runtime SQLIA prevention. [5] The latter two options are primarily third party programs that run on top of the database systems. In this section, we will focus on defensive coding techniques that can be implemented by developers without any additional resources to prevent SQLIA.

### B. Common Prevention Methods

According to the Open Web Application Security Project, there are four primary methods for preventing SQL Injection attacks. [6] They are: Use of prepared statements (with parameterized queries), use of properly constructed stored procedures, allow-list input validation, and escaping all user-supplied input. Additionally, there are two defensive coding principles that should be used to mitigate damage from successful SQLIA. The first is the principle of least privilege. By limiting permissions of each application or user to the least amount required to function, any compromise of that application or user is limited in scope. This includes the use of views to limit the data visible to each user. If stored procedures are going to be used, the individual users should not be granted access to the database. It is also best practice to make as many specific user accounts as possible with limited access, rather than few accounts with broad privileges. The second principle is secure error handling. While not a prevention method in and of itself, effective error handling can prevent the leakage of valuable information to an attacker. When an error occurs, the information that gets sent to the user should be minimal, avoiding the disclosure of any details about the database or the SQL queries used. A system that fails securely can deter attackers and prevent them from gaining more knowledge about the system's structure. [7]

Use of prepared statements is the most effective method for preventing SQLIA. Prepared statements are precompiled SQL statements that are stored in the database. They are created by the developer and contain placeholders for user input. When the application needs to execute a query, it sends the user input to the database along with the prepared statement. The database then executes the prepared statement with the user input. This method

prevents SQLIA because the user input is never directly concatenated with the SQL query. Instead, it is sent to the database separately from the query. This prevents the user input from being interpreted as SQL commands. [6] Effectively, this treats the user input like a string rather than allowing it to be concatenated with the SQL statement. For example the database would search the students table for a student literally named "Robert'); DROP TABLE students; --" rather than executing the malicious code.

Stored procedures allow developers to create a set of SQL statements that can be executed by the database. They are similar to prepared statements in that they are precompiled, but are stored in the database rather than in the application. Stored procedures can have the same defensive capability as prepared statements, but must be implemented properly. It is important that the stored procedures do not dynamically generate any SQL code, as this can expose the system to the same vulnerabilities as if the input were not sanitized at all. Additionally, stored procedures run with elevated privileges on some database systems (MSSQL for example) and can thus violate the principle of least privilege.[6] If constructed properly, stored procedures can force the database to treat user input as a string rather than as SQL code. Much like prepared statements, this can greatly reduce the vulnerability of the system to SQLIA.

Allow-list input validation can be used in cases where a query requires the user to input the name of a column or table, or a parameter such as ASC or DESC. In these cases, the application can check the user input against a list of valid inputs. If the user input is not on the list, the application can reject the input and prevent the query from being executed. This technique is recommended as a secondary defense in cases where prepared statements or stored procedures are not feasible. [6]

Escaping all user input is a last resort option for defensive coding. This technique is generally only used to retrofit existing systems where it might be too difficult to implement the previous techniques. [6] Escaping user input involves replacing certain characters with escape characters. For example, the single quote character ' is replaced with \'. This prevents the database from interpreting the input as SQL code. This technique is not recommended as the implementation will be complex, specific to each database system, and it is possible to miss characters that have the potential for malicious use.

### C. Implementation on the Sample Database

To neutralize the demonstration attack, we will use a prepared statement which implements a parameterized query.

Here in Example 2, the user has optimized the query by parameterizing the query and making use of parameter place holder "%s". By making use of the parameter placeholder

```

52 Tables in WWI schema:
53 Application.Cities
54 Application.Cities_Archive
55 Application.Countries
56 Application.Countries_Archive
57 Application.DeliveryMethods
58 Application.DeliveryMethods_Archive
59 Application.PaymentMethods
60 Application.PaymentMethods_Archive
61 Application.People
62 Application.People_Archive
63 Application.StateProvinces
64 Application.StateProvinces_Archive
65 Application.SystemParameters
66 Application.TransactionTypes
67 Application.TransactionTypes_Archive
68 Purchasing.PurchaseOrderLines
69 Purchasing.PurchaseOrders
70 Purchasing.SupplierCategories
71 Purchasing.SupplierCategories_Archive
72 Purchasing.Suppliers
73 Purchasing.Suppliers_Archive
74 Purchasing.SupplierTransactions
75 Sales.BuyingGroups
76 Sales.BuyingGroups_Archive
77 Sales.CustomerCategories
78 Sales.CustomerCategories_Archive
79 Sales.Customers
80 Sales.Customers_Archive
81 Sales.CustomerTransactions
82 Sales.InvoiceLines
83 Sales.Invoices
84 Sales.OrderLines
85 Sales.Orders
86 Sales.SpecialDeals
87 Warehouse.ColdRoomTemperatures
88 Warehouse.ColdRoomTemperatures_Archive

```

Figure 3: Target of Attack

"%s" (which is specific to MySQL), the user ensures that the library properly escapes and handles the values in the query.

*# Example 2 - Correctly Parameterized -  
# showing how we corrected the code*

```

from getpass import getpass
from mysql.connector import connect, Error

# Establish the connection
try:
    connection = connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: ")
    )
    print("Connection established successfully.")

    try:
        with connection:

```



```

        cursor = connection.cursor()
        cursor.execute("USE WWI")
        table_name = 'Application.People'
        query = f"SHOW TABLES LIKE %s"
        cursor.execute(query, (table_name,))
        result = cursor.fetchall()
        print(result)
    except Error as e:
        print(e)

except Error as e:
    print("Error connecting to MySQL:", e)

```

```

Connection established successfully.
[('Application.People',)]

```

Figure 4: Database after Example 2

As before in Example 1, Example 3 below will attempt an SQL Injection attack to DROP the table Application.People (boxed in yellow in fig. 3).

Here in Example 3, the user's properly parameterized query comes under attack but the hackers attempt to drop the table fails. The user's properly parameterized query avoids SQL injection vulnerabilities by pre-compiling the query and using MySQL's parameter placeholder in the query separate from the input data/variable (table\_name) that the hacker took control of. The database remains unaltered.

```

# Example 3 - Correctly Parameterized
# with failed SQL Injection Attack
from getpass import getpass
from mysql.connector import connect, Error

# Establish the connection
try:
    connection = connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: ")
    )
    print("Connection established successfully.")

    try:
        with connection:
            cursor = connection.cursor()
            cursor.execute("USE WWI")
            table_name = 'Application.People'
            sql_injection =
                """; DROP TABLE
                `Application.People`; --"""
            table_name=sql_injection
            query = f"SHOW TABLES LIKE %s"
            cursor.execute(query, (table_name,))
            result = cursor.fetchall()

```

```

    except Error as e:
        print(e)

except Error as e:
    print("Error connecting to MySQL:", e)

```

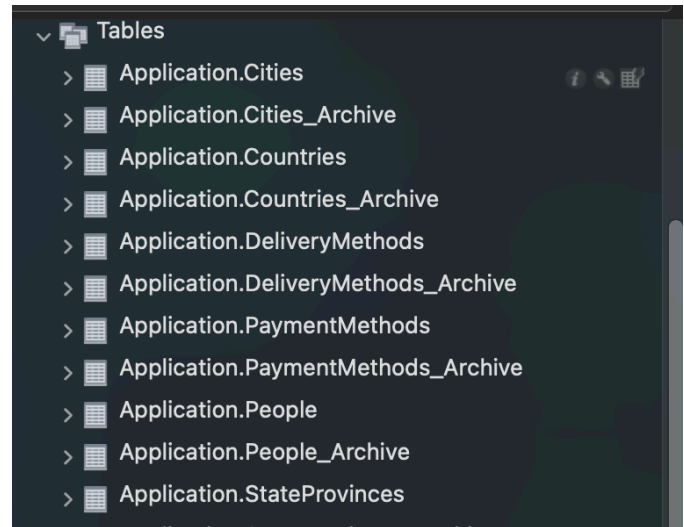


Figure 5: Database after Example 3

#### D. Comparison of Results

Using parameterization in the application code prevents the database from executing the malicious SQL code. We can see above that the database has not been altered by the attack (See fig. 6). Separating the user input from the SQL query prevents the database from interpreting the input as SQL code, so the query doesn't return anything.

```

Connection established successfully.
Table exists: [('Application.People',)]
('Application.People',)

```

Figure 6: Database after Attack

Because of its effectiveness, simplicity and ease of implementation, parameterization is the recommended first line of defense for preventing SQLIA.

## VI. CONCLUSION

#### A. Summary of findings

SQL Injection is a common and dangerous vulnerability that can be exploited to gain access to a database. It is a type of injection attack that allows an attacker to execute arbitrary SQL code on a database. SQLIA can be used to extract data from a database, modify data in a database, or even delete entire tables. It is a serious threat to the security of any database system and remains among the top five most critical web application security risks [8] decades after its discovery.

SQLIA can be prevented by using defensive coding techniques such as parameterized queries, stored procedures,

allow-list input validation, and escaping user input. Parameterized queries are the most effective method for preventing SQLIA. They are easy to implement and can be used in most cases where user input is required. Stored procedures can also be used to prevent SQLIA, but must be implemented properly to be effective. Allow-list input validation and escaping user input are less effective methods that should only be used when the other methods are not feasible. Additionally other layers of defense can and should be used on production databases. These include systems for detecting SQLIA and systems for preventing SQLIA at runtime and are specific to the type of database being used.

### *B. Limitations of the research*

The research was limited to the use of SQL injection attacks on MySQL databases using Python. While the principles of SQLIA are applicable to other database systems, the implementation of preventive measures may vary.

### *C. Recommendations for further research*

Further research in this topic could go in many directions. SQLIA is an important topic in the field of cybersecurity and there is still much to be learned about it. The research could be expanded to include other database systems such as MSSQL or Oracle. Additionally, the research could be expanded to include other programming languages such as Java or C# or other types of injection attacks such as LDAP injection or XML injection. Another option would be to research the strategies and techniques for SQLIA detection.

## REFERENCES

- [1] Invicti Security Team, “14 years of SQL injection history and still the most dangerous vulnerability.” 2013.
- [2] Yasar, Kinza and Terrell Hanna, Katie and Lewis, Sarah, “SQL injection (SQLi).”
- [3] Microsoft Learn, “Wide world importers sample databases for microsoft SQL.” 2023.
- [4] Microsoft Learn, “WideWorldImporters database catalog.” 2023.
- [5] L. K. Shar and H. B. K. Tan, “Defeating SQL injection,” *Computer*, vol. 46, no. 3, pp. 69–77, 2012.
- [6] Open Web Application Security Project, “SQL injection prevention cheat sheet.” 2019.
- [7] W. G. Halfond, J. Viegas, A. Orso, and others, “A classification of SQL-injection attacks and countermeasures,” in *Proceedings of the IEEE international symposium on secure software engineering*, 2006, vol. 1, pp. 13–15.
- [8] Open Web Application Security Project, “A03:2021 – injection.” 2021.

[9] Chaitanya Baweja, “Python and MySQL database: A practical introduction.” 2023.

[10] Haki Benita, “Preventing SQL injection attacks with python.” 2023.

[11] Hacksplaining.com, “SQL injection prevention.”