|  |  |
| --- | --- |
|  | Lab 2 – SQL INJECTION |
|  |  |
|  | James Cassidy  Secure Software Development  11/3/22 |

**Table of Contents**

[**Executive Summary** 2](#_Toc98674617)

[**Proposal** 2](#_Toc98674618)

[**Risk** 2](#_Toc98674619)

[**Recommendation** 2](#_Toc98674620)

[**Testing Performed** 3](#_Toc98674621)

[**Vulnerabilities Detected** 3](#_Toc98674622)

[**Mitigation Techniques** 3](#_Toc98674623)

[**Least privilege** 3](#_Toc98674624)

[**Encrypt sensitive data** 4](#_Toc98674625)

[**Error Messages** 4](#_Toc98674626)

[**Preventative Measures** 4](#_Toc98674627)

[**Input Validation** 4](#_Toc98674628)

[**Prepared Statements** 4](#_Toc98674629)

[**Action Taken** 5](#_Toc98674630)

[**Conclusion** 5](#_Toc98674631)

[**Appendices** 6](#_Toc98674632)

[**Testing performed** 6](#_Toc98674633)

[**Always True Statement** 6](#_Toc98674634)

[**ORDER BY** 7](#_Toc98674635)

[**UNION SELECT** 7](#_Toc98674636)

[**Action taken** 8](#_Toc98674637)

[**Prepared statement** 8](#_Toc98674638)

[**Generic Error Messages** 8](#_Toc98674639)

[**Bibliography** 9](#_Toc98674640)

# **Executive Summary**

Extensive penetration testing in DVWA shows that a SQL Injection vulnerability exists, allowing an attacker to steal all passwords from the database. The business will suffer catastrophic ramifications should this vulnerability remain open.

Eliminating this vulnerability will ensure customer and business assets remain intact.

## **Proposal**

The following proposals have been made to address the vulnerability:

* **Prepared Statements**
* **Input Validation**
* **Principle of Least Privilege**
* **Encrypt sensitive data**
* **Displayed errors directly from the database**

These preventive measures and mitigations techniques are explained under their respective headings.

## **Risk**

There is a risk involved with this:

* **Delayed-Release *–*** To implement the proposed solutions effectively, the applications' release needs delaying. Failure to do so will result in far more significant risks in the product's life cycle.

## **Recommendation**

Fixing this vulnerability without disrupting the rest of the business's current operations, an increase in the IT budget and deadline extension will need to be approved to support the time spent on fixing this issue.

# **Testing Performed**

Penetration testing was performed on DVWA SQL Injection on Low difficulty.

The objective of this attack is to steal all five passwords of the users on the system. Entering an apostrophe into the application displays an error message alongside the database being used, allowing an attacker to tailor their injection attack.

The following attack string was entered:

* **1' or '1' = '1**

This statement is always true and returned all five users alongside their surnames, confirming SQL injection is possible.

Following this, the **ORDER BY** clause was utilized to determine how many columns existed in the database.

Successfully finding the number of columns, the UNION operator can then be utilized, with the following attack string entered:

* **' UNION SELECT user , password FROM users#**

The application returned all five user IDs including admin alongside each of their hashed passwords. System integrity has now been compromised. More detail can be found under '[Testing Performed](#_Testing_performed)' in Appendices.

# **Vulnerabilities Detected**

Several vulnerabilities were found that allowed this attack to become possible.

Input into the database is not sanitized. As a result, an attacker can send any SQL command to this database that will be executed without possessing the necessary permissions.

Once an apostrophe is entered into the application, the error message states the SQL server used, allowing an attacker to tailor their input to successfully exploit the SQL injection vulnerability successfully.

# **Mitigation Techniques**

Several mitigation techniques need to be implemented to lessen the impact of SQL injection if it occurs.

## **Least privilege**

The principle of least privilege will minimize the potential damage caused by an SQL injection attack. Therefore, every database account on the application must strictly have the permissions required to fulfil their job.

Database accounts should only be granted read or write access to the table that they only need access to.

It is imperative that admin type access rights are not given to any accounts. The first account on this database is an admin (sa). This account needs removed immediately to pervert an attacker gaining root access to the entire application.

Limited privileges for accounts will limit the scope of damages caused and therefore reduce the threat level to the business to a certain degree.

## **Encrypt sensitive data**

Sensitive data needs to be encrypted on the database should SQL Injection occur. An attacker will be unable to benefit from encrypted data, reducing the threat posed by this attack.

## **Error Messages**

Error messages allows an attacker to refine their injection technique. This application's error message displayed MariaDB, allowing an attacker to craft their injection around that database.

Generic errors message must be displayed when an incorrect or malicious string in input into an SQL query. This increases the time it takes for an attacker to exploit an SQL Injection and gives the business time to react, potentially minimizing damages caused.

# **Preventative Measures**

Several preventative measures need to be implemented during the application's development life cycle to prevent this attack from occurring in future.

## **Input Validation**

This validation process is crucial in preventing an SQL Injection attack. Any input that a user provides needs to be validated and cleansed.

This process will check the user input for correct length, invalid characters, and other anomalies. Therefore, input validation should occur early in the data flow when an external party proceeds to make an input.

Whitelist Validation checks user input amongst a set of known safe values. If the input detected is invalid, the application rejects it instead of trying to fix it.

Input validation reduces the threat level of this attack significantly. Correct implementation reduces the likelihood an attacker exploits this vulnerability as particular characters cannot be parsed as malicious SQL queries that trigger an SQL Injection.

## **Prepared Statements**

Prepared Statements are parameterized and reusable SQL queries that separate query structure into query code and data. (Ablon, 2020)

User input, therefore, is not directly embedded into the SQL query. As a result, malicious queries can still be passed through but aren't executed like before, allowing an SQL command to run safely preventing an SQL injection.

The threat level is reduced significantly with this preventative measure. An attacker cannot execute always true statements or other malicious queries, resulting in a failed SQL Injection attack.

# **Action Taken**

Several actions have already been taken to resolve the vulnerabilities mentioned

* **Prepared Statement**
* **Generic Error Message**

From the testing that was performed for this application, these changes are necessary to eliminate the vulnerabilities found. The threat level to the business will reduce significantly following this. These changes can be found under ['Action Taken'](#_Action_taken) in Appendices.

# **Conclusion**

SQL Injection continues to be a major vulnerability with no industry wide resolution. The business needs to be aware of how this can put sensitive data at risk.

Input validation ensures input is cleaned before it can be processed. In addition, prepared statements ensure malicious queries that bypass input validation cannot be executed within the query. Generic error messages ensure injection attacks cannot be tailored to a specific database and least privilege ensures an attacker can't gain much control of a compromised database.

Maintaining these measures throughout the software development lifecycle will eliminate the threat to the business almost entirely.

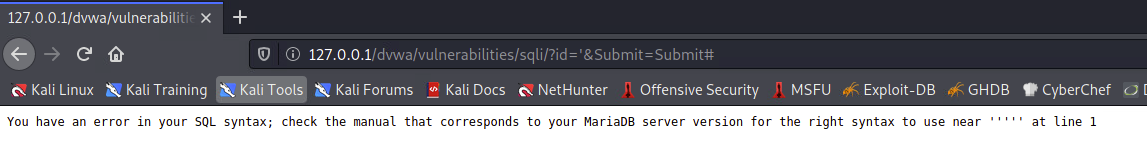
# **Appendices**

## **Testing performed**

The objective of this attack is to steal the passwords of the five users within the database. **Text

Description automatically generated with medium confidence**

From the menu, an attacker is able to enter what user ID they wish from 1 to 5. Entering an apostrophe force the database to consider any characters following the apostrophe as a string and not as SQL code, forcing a syntax error. This displays an error message:

****

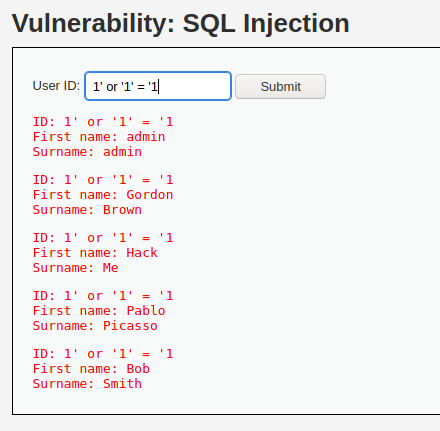
The message displayed tells an attacker MariaDB is the SQL server being used by the application. This is a major vulnerability and alerts an attacker on how to construct their malicious queries accordingly.

### **Always True Statement**

The following attack string is then entered:

* **1' or '1' = '1**

This will confirm if a SQL attack is possible. Once submitted, the first and surname of all users within the database are displayed as 1 = 1 is always equal to true. This confirms SQL injection is possible. Application integrity is now compromised.

****

As the vulnerability is now known, the operator 'UNION' allows an attacker to execute one or more SELECT queries and add those results to the original query.

### **ORDER BY**

For UNION SELECT to be successful, the number of columns must be known. The following ORDER BY clause was used:

* **' ORDER BY 3#**

An error is returned with this input. When this is decremented to:

* **' ORDER BY 2#**

No error is returned with this input. The attacker now knows how many columns are returned by the original query. (Portswigger, n.d.)

The symbol '#' is used to comment out the remainder of the original query following the injection point.

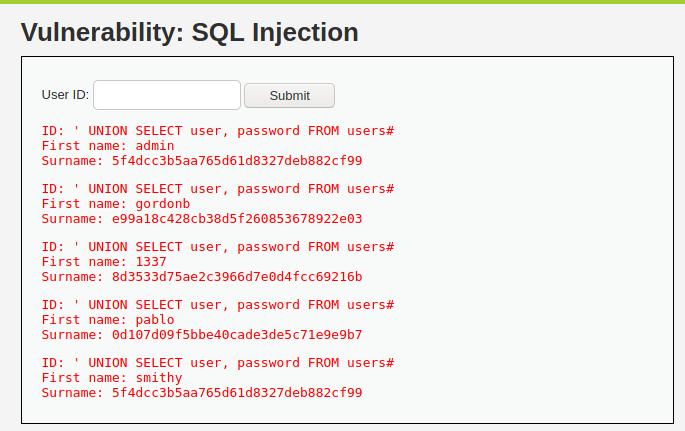
Graphical user interface, application

Description automatically generated

### **UNION SELECT**

The passwords of each user can now be accessed from the users' table by submitting the following UNION SELECT query:

* **' UNION SELECT user , password FROM users#** (CryptoCat, 2021)



This will select all data from columns' user' and 'password' and return their contents. An attacker now has access to all five users and their password hashes. Attackers can use applications like 'Jack the Ripper' to crack these password hashes and gain access to all five accounts, including admin.

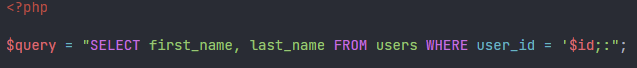
## **Action taken**

### **Prepared statement**

With this prepared statement written in PHP, the user-supplied input is not directly embedded into the SQL query. Instead of the user-supplied query, there is a '?' symbol. This question mark acts as a placeholder and takes place temporarily instead of the data itself. If an attacker inserts **1' or '1' = '1,** the initial query won't be changed. Instead, the database will look for the user\_id that is **1' or '1' = '1.** (Ablon, 2020)

The prepared statement has done its job in preventing the attacker from executing a malicious SQL query, reducing the threat level to the business significantly.

**Original Statement**



**Prepared Statement**

Text

Description automatically generated

### **Generic Error Messages**

With this error message, the attacker is told to contact support if they continue to experience problems with their input. The attacker, in this case, has attempted to enter an apostrophe into the data. Before, this would return an error displaying the database used, allowing the attacker to tailor their SQL Injection attack.

Now an attacker will need to find other ways to exploit SQL Injection successfully.

Graphical user interface, text, application

Description automatically generated

# **Bibliography**

Ablon, J. (2020, February 11). *How to prevent SQL Injection vulnerabilities: How Prepared Statements Work*. Retrieved from Medium: https://jaredablon-31568.medium.com/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-work-f492c369614f

CryptoCat. (2021, February 27). *SQL Injection (low/med/high) - Damn Vulnerable Web Application (DVWA)*. Retrieved from Youtube: https://www.youtube.com/watch?v=5bj1pFmyyBA

Portswigger. (n.d.). *SQL injection UNION attacks*. Retrieved from Portswigger: https://portswigger.net/web-security/sql-injection/union-attacks