MITIGATION OF SQL INJECTION ATTACKS

Pallavi Shinde[1], Manasi Wavade[2], Kalyani Mishra[3]

Siddharth Mukhia[4] , Mugdha Parande[5]

Information Technology Department, Mumbai University
Atharva College of Engineering,Malad,Mumbai,India

pallavi.shinde087@gmail.com, wavademanasi@gmail.com , mishra.kalyani48@gmail.com, sid.mukhia@gmail.com
parandemugdha@gmail.com

***Abstract*— Internet technologies has played a very important role in easing the lives of humans in numerous ways, but the drawbacks like the intrusions that are attached with the Internet applications, sustains the growth of these applications. SQLIA contributes 25% of the total Internet attacks. In this paper a method is proposed to detect the SQL injection; where a Reverse proxy is used to mitigate SQL Injection Attack using the cleansing algorithm.**

***Keywords*— SQL Injection , SQL attack, Security threats, Web application vulnerability**

I. INTRODUCTION

In the recent years, web applications have tended to become common place. Nowadays there is a plethora of web applications that cover a wide range of daily needs. A large number of electronic transactions, including e-commerce, e-banking, e-voting, e-learning, and e-health among others, can be conducted online at any time and from any place. However, in all these Internet applications exposed to hacking attempts, security-related problems are a major issue. SQL injection represents today the most common indirect attack technique against web-powered databases and can disassemble effectively the secrecy, integrity and availability of web applications. SQL injection occurs when an attacker inserts malicious SQL code into an SQL query by manipulating data input into an application. This kind of vulnerability is a serious threat to any web application that reads input from users and uses it to build and execute SQL queries to an underlying database. With SQL injection, the attacker can run arbitrary SQL queries, extracting sensitive customer and order information from e-commerce applications, or s/he can bypass strong security mechanisms compromising the back-end databases and the data server file system.

II. ILLUSTRATION OF A SIMPLE SQL ATTACK

The requirement of Safeguarding the Web Application can be met by filtering all the requests before any transaction in the database takes place or the user is able to access any sort of sensitive data.

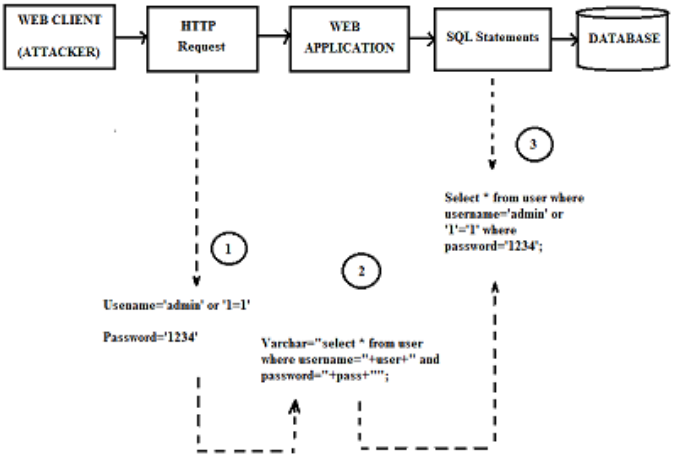

Fig. 1 Illustration of sample of SQL attack

SQL injection vulnerabilities allow attackers to insert SQL commands as a part of user input [1]. When an SQL query is constructed dynamically with maliciously-devised user input containing SQL keywords, attackers can gain access or modify critical information such as a credit card number in a database without proper authorization.
However, an attacker can enter the input for the values of login ID and password through a web form.
It would generate the following query:

SELECT info FROM account WHERE id = '1' OR '1' = '1' AND password = '1' OR '1' = '1';

Because the given input makes the WHERE clause in the SQL statement always true (a tautology), the database returns all of the user information in the table. Therefore, the malicious user has been authenticated without a valid login ID and password. Most Web applications used on the Internet or within enterprise systems work this way and could therefore be vulnerable to SQL injection. The cause of SQL injection vulnerabilities is relatively simple and well understood: insufficient validation of user input.

III. TYPES OF ATTACKS

The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific goals of the attacker [13].

A. Tautologies

1) Attack Intent: Bypass authentication, identifying inject able parameters, extracting data.

2) Description: The general goal of a tautology-based attack is to inject code in one or more conditional statement's so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. An attacker exploits an injectable field that is used in a query's WHERE conditional.
The most common usages are to bypass authentication pages and extract data thereby attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned.

B. Illegal/Logically Incorrect Queries

1) Attack Intent: Identify inject able parameters, performing database finger- printing, extracting data.

2) Description: This attack lets an attacker gather important information about the type and structure of the back-end database of a Web application. It is a preliminary, information gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. Errors messages are generated can often reveal vulnerable/inject able parameters to an attacker. Syntax errors can be used to identify injects able parameters.

C. Union Query

1) Attack Intent: Bypass Authentication, extracting data

2) Description: In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. In this, an attacker trick the application into returning data from a table different from the one that was intended by the developer. Attackers do this by injecting a statement of the form:
UNION SELECT <rest of injected query>.
The attacker can use that query to retrieve information from a specified table as it can completely control the second/injected query resulting into execution of the injected second query on dataset.

D. Stored Procedures

1) Attack Intent: Perform privilege escalation, performing denial of service, executing remote commands.

2) Description: SQLIAs of this type try to execute stored procedures present in the database. Most databases have standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Hence, an attacker can craft SQLIAs to execute stored procedures provided by that specific database, including procedures that interact with the operating system once backend has been determined.

E. Piggybacked Queries

1) Attack Intent: Extract data, adding or modifying data, performing denial of service, executing remote commands.

2) Description: In this, attacker tries to inject additional queries into the original query. In this attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that "piggy-back" on the original query resulting into database receiving multiple SQL queries. Both the queries are executed one after another. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string.

F. Alternate Encodings

1) Attack Intent: Evade detection.

2) Description: In this attack, the injected text is modified so as to avoid detection by defensive coding practices and also many automated prevention techniques. This attack type is used in conjunction with other attacks. It is not a unique way to attack an application; they are simply an enabling technique that allows attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known "bad characters".

IV. PROBLEM STATEMENT

The Existing System assumes that user input consists only of values (numbers and strings) that are not meant to be interpreted as SQL tokens. This technique cannot handle cases in which the user input is legitimately supposed to add SQL tokens to the query. Applications that allow the user to do so would cause this technique to generate false positives because we would recognize the user-introduced SQL tokens and operators as an injection. Also the Existing Systems does not check the URL signatures which can be changed by Attacker during the SQLIA.
Thus, the Proposed System has the capability to mitigate both the possible threats using Filter Application located at reverse Proxy Server.

V. PROPOSED ARCHITECTURE

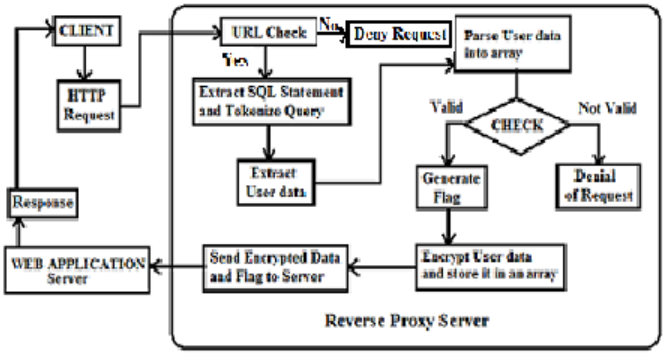

Fig. 2 Proposed System Architecture

Working of System described as follow [2]:

Step1:
The client sends the request to the Reverse proxy server.

Step 2:
The sanitizing application in the Reverse proxy server extracts the URL Query String from the HTTP request and the user data from the SQL statement.

The filtering Application checks the URL signature
If signature is benign
Then
Continue
Set *flag* to true

Else
Discard

Step 3:
The Application checks the authorisation of SQL statement in prototype model.
If the Statement is authorised
Then
Continue
Else
Discard

Step 5:
User Input data is checked for reserved SQL keywords.
If the Input data is benign
Then
Input Data is hashed
Else
Discard
\
Step 6:
Encrypt the Input Data using hashing algorithm. Forward the request and the flag to Web Application Server.

Step 7: If the hashed user data matches the stored hash value in the database, then the data is retrieved and the user gains access to the account.

Step 8:
Else the user is denied access.

VI. SCOPE

The filtering application installed on the Reverse Proxy Server provides basic solution for preventing unauthorised access to the server thereby preventing any loss sensitive data. Thus, it provides a secure gateway for Online Transaction for Customers.
By blocking the unauthorised access it provides a way to prevent the loss of business by securing the sensitive data about the Clients which was vulnerable prior to the implementation of security application.

VII. SANITIZING APPLICATION

Cleaning Algorithm:

Step 1:
- Extract the URL Query String from HTTP;
- Parse the URL into Tokens-*toks;*
- While (not empty of *toks*)
- Check if (URL = Benign using the signature check)
- Set the *flag* to *continue*;
- Else
- Set the *flag* to *deny*;

Step 2:
- Extract the SQL statement from the Query String;
- Tokenize the SQL Statement-*q(array)*;
- While (not empty of *q*)
  Change Character Encoding to UTF;
  Add token to Array-*sqlarr*
- For (every data in prototype array)
- Check if (*sqlarr*= prototype model in document)
- Extract the user input data;

Step 3:
- Parse the user data into array-*usrarr;*
- While (not empty of *usrarr*)
- Check if every element in *usrarr* ≠ reserved SQL Keyword
  Else
  Deny Request;

Step 4:
- For (every data in usrarr)
- Perform appropriate data encryption and store the data in a *UDA (User Defined Array)*.

Step 5:
- Send the encrypted user data and *flag* to Web application Server;

VIII. FLOW CHART OF THE SYSTEM

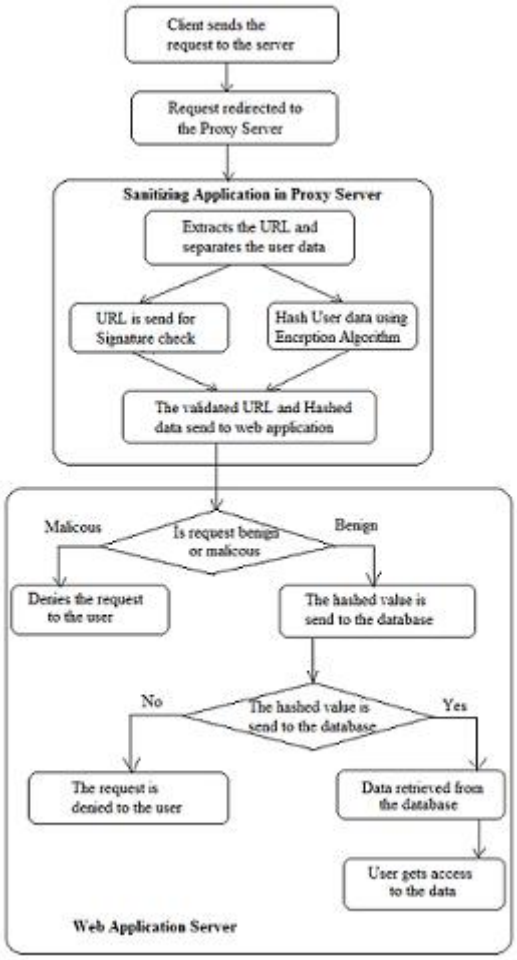

Fig. 3 Flowchart of the System

IX. PERFORMANCE OF OUR SOLUTION

TAUTOLOGY BASED SQL INJECTION	YES
STATEMENT INJECTION	YES
UNION QUERY	YES
STORED PROCEDURES	YES
ALTERNATE ENCODING	NO
ILLOGICAL/INCORRECT QUERIES	YES
INFERENCE	NO

Fig. 4 Output Result after Defence Mechanism

X. CONCLUSION

In this paper, we have presented a survey on different types of SQLIA and some of the important approaches for detection and preventing of SQLIA. Our proposed technique will able to suitably classify the attacks that performed on the applications without blocking legitimate accesses to the database.

REFERENCES

[1] Y.Huang, F.Yu, C. Hang,C.H. Tsai, D.T.Lee and S.Y.Kuo, (2004) "Securing Web Application Codeby Static Analysis and Runtime Protection", Proc. International World Wide Web Conference '04, pp.

[2] Chip Andrews, "SQL Injection FAQs", http://www.sqlsecurity.com/FAQs/SQLInjectionFAQ/tabid/56/Default.aspx

[3] W. G. Halfond and A. Orso, (2005) "AMNESIA: Analysis and Monitoring for NEutralizing SQLInjection Attacks", Proc. ACM International Conference on Automated Software Engineering '05, pp.

[4] Zhendong Su and Gary Wassermann, (2006) "The Essence of Command Injection Attacks in Web Applications", Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages '06, pp.372-382.

[5] D.Scott and R.Sharps, (2002) "Abstracting Application-level Web Security", Proc. International Conference on the World Wide Web '02, pp. 396-407.

[6] S.W. Boyd and A.D. Keromytis, (2004) "SQLrand: Preventing SQL Injection Attacks", Proc. 2nd Applied Cryptography and Network Security (ACNS) Conference, pp. 292-302.

[7] W. Halfond, J. Vigeas and A.Orso, (2006) "A Classification of SQL Injection Attacks and Counter Measures", Proc. International Symposium on Secure Software Engineering '06.

[8] C.Gould, Z.Su and P.Devanbu, (2004) "JDBC Checker: A Static Analysis Tool for SQL/JDBCApplication", Proc. International Conference on Software Engineering '04, pp.697-698.

[9] Konstantinos Kemalis and Theodoros Tzouramanis, (2008) "SQL-IDS: a specification-based approach for SQL-injection detection", Proc. 2008 ACM symposium on Applied computing, pp.2153 - 2158.

