SQL INJECTION DETECTION AND PREVENTION SYSTEM

Ankit Raj¹ and Akshay Kaushal²

Abstract—At Present time where we all are in the moment of doing our work easily using technology on the other hand we also getting challenge about security of that technology. SQL injection attacks pose a serious security threat to Web applications: they allow attackers to obtain unrestricted access to the databases underlying the applications and to the potentially sensitive information these databases contain. Although researchers and practitioners have proposed various methods to address the SQL injection problem, current approaches either fail to address the full scope of the problem or have limitations that prevent their use and adoption. In this paper we present a detail on numerous types of SQL injection attacks and prevention technique for web application. We also describes the technique to prevent injections attacks occurring due to dynamic SQL statements in database stored procedures, which are often used in e-commerce applications.

Index Terms—Sql injection, sql injection vulnerability, web application, security, threat, virus.

I. INTRODUCTION

A Database is the heart of many, if not all, web-applications and is used to store information needed by the application, such as, credit card information, customer demographics, customer orders, client preferences, etc. Consequently, databases have become attractive and very lucrative targets for hackers to hack into. SQL Injections happen when a developer accepts user input that is directly placed into a SQL Statement and doesn’t properly validate and filter out dangerous characters. This can allow an attacker to alter SQL statements passed to the database as parameters and enable her to not only steal data from your database, but also modify and delete it. Web applications are often vulnerable for attackers that can be easily accessed to the application's underlying database. SQL injection attacks occurs only when a malicious user causes a web application to generate and send a query that functions differently than the programmer intended.

A SQL injection takes place when the application fails to properly sanitizing the user supplied input used in SQL queries. An attacker can manipulate the SQL statement which is passed at the backend of database management system.

What Is Sql Injection Attack?

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

Figure 1

SQL injection exploits security vulnerabilities at the database layer. By exploiting the SQL injection fault, attackers can create, read, modify, or delete sensitive data.

An Attacker provides specially designed input data to the SQL interpreter and traps the interpreter to execute unintended commands.
II. PROBLEM DEFINITION

There are different kinds of SQL Injection attacks and each attack is performed for some specific purposes. These purposes are as follows:

- **Identifying Injectable Parameters**
  The attacker wants to know what kind of parameters and user input fields are vulnerable to SQL Insecurity (SQLIA) in a web application.

- **Performing Database Finger Printing**
  The attacker wants to know what type and version of database is being used by the Web application. Different types of databases will be responding differently to different queries and attacks, and this information can be used to “finger print” the database. If the attacker knows the type and version of the database used by a Web application then it allows the attacker to design database specific attacks.

- **Determining Database Schema**
  The attacker wants to discover database schema information such as table names, column names, and column data types in order to correctly clipping data from a database.

- **Extracting Data**
  These types of attacks employ techniques that will clip valuable data values from the database.

- **Adding or Modifying Data**
  The main motive of these attacks is to add or modify information in a database.

- **Performing Denial of Service**
  These attacks are used to shut down the database of a Web application, thus denying service to other users even to logical ones.

- **Evading Detection**
  This type of attacks refers to those which are employed to avoid verification and detection by system protection mechanisms.

- **Bypassing Authentication**
  The motive of these types of attacks is to let the attacker to bypass authentication Mechanisms of application and database. By passing such mechanisms could let the attacker to assume the rights and privileges associated with another application.

- **Executing Remote Commands**
  These types of attacks try to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.

III. OBJECTIVES

1. To design technique for effective dynamic detection and prevention of SQL Insecurity (SQLIA) without access to the application source code.
2. To implement a heuristic approach to searching a valid query structure in our database to minimize the response time.
3. To compare computational overhead of string matching algorithm for matching tokens by converting the token to some integer values and checking those integer values instead of the tokens.

IV. RESULTS AND DISCUSSION

This research investigates what are SQL injection attacks in Real Time web Application with example and how that can be prevented. SQL injection means injecting some SQL commands in SQL statements to hack the data or delete data or change contents in tables via web page input.

When the application is not secured it can easily hacked and even destroy all database using SQL injection.

When we run code we will get throughout like as shown in figure 2.

![Figure 2](image)

This SQL code is designed to pull up the records of the specified username from its table of users. However, if the "user Name" variable is crafted in a specific way by a malicious user, the SQL statement may do more than the code author intended. For example, setting the "user Name" variable as ' OR '1'='1
- SELECT * FROM users WHERE name = " OR '1'='1';
- DROP TABLE users; SELECT * FROM user_info WHERE 't' = 't'.

LOGIC: 'a'='a' that is always true.

Example: SELECT * FROM `table` WHERE 'a'='a'

SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrators of the database server. After sql injection hacker get the access of application as shown in following figure.

As the hacker access the database of application he can easily hack all the details of user. He can easily see all the records and alter the data as in above he got the email id of user and its password as well.

V. SQL INJECTION PREVENTION

There are many architectures which can manage and organize any data-driven systems, but the most common architecture which is used is the three-tier architecture that depends on dividing the system into three tiers as follows:

1. Presentation Tier (a Web browser or rendering engine).
2. Logic Tier (a server code, such as C#, ASP, .NET, PHP, JSP, etc ...).
3. Storage Tier (a database such as Microsoft SQL Server, MySQL, Oracle, etc.)

This paper focused on providing clear, simple, actionable guidance for preventing SQL Injection flaws in your applications. SQL Injection attacks are unfortunately very common, and this is due to two factors:

1. the significant prevalence of SQL Injection vulnerabilities, and
2. the attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).

It’s somewhat shameful that there are so many successful SQL Injection attacks occurring, because it is EXTREMELY simple to avoid SQL Injection vulnerabilities in your code.

SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input. To avoid SQL injection flaws is simple. Developers need to either: a) stop writing dynamic queries; and/or b) prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

VI. SUGGESTED APPROACH

The suggested approach is based on different steps to reject any malicious query from being passed through the database engine before its execution course, and those steps could be listed as follows:

1. Prepared Statements (with Parameterized Queries)

The use of prepared statements with variable binding (aka parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then
pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied. Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. In the safe example below, if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1.

Language specific recommendations:
- **Java EE** – use PreparedStatement() with bind variables
- **.NET** – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
- **PHP** – use PDO with strongly typed parameterized queries (using bindParam())
- **Hibernate** - use createQuery() with bind variables (called named parameters in Hibernate)
- **SQLite** - use sqlite3_prepare() to create a statement object

2. Parsing And Pattern Matching
For each database for which we want to be secured from SQLIAs, there should be a new replication database and it must contain a small amount of sample data.

Creating “database Behaviors” database The suggested approach must have a separate database called “database Behaviors” that will be containing all system database queries and their familiar behaviors that have resulted from SQL queries execution in normal cases. This database is located in the replicated instances.

3. Mailing Alert (Sqlia Detection)
This stage is the most important stage in the suggested technique, its purpose is to detect whether the received SQL query is valid or not. The idea here is to catch the object that was affected by the current SQL query whatever the type of such objects and create a list of these objects to use them in the upcoming step of this stage. The resulted list of affected objects will be matched with the “database_Behaviors”. If there is a query which is handling the entire listed objects with the previous step then this behavior query will be added to a new list (Expected Queries). Any resulted behavior that is detected as a suspicious must be rejected and deleted from the actual database instance execution queue; otherwise the query will be transferred to the actual database instance for being executed. Algorithm for detecting and preventing SQL Injection Attack.

VII. CONCLUSION
The suggested compound technique will be done in two main phases: runtime analysis, and static analysis. The first phase is a dynamic/runtime analysis method which depends upon applying tracking methods to process and monitor the execution processes of all received queries. The resultant of affected objects of this monitoring will be matched with a planned set of expected changes that the developer had created before, and the output of this comparison process will decide if there is an existence of any type of SQLIA and if so that will be forwarded to the next phase. The next phase is a static analysis phase that is performing a string matching between the received SQL queries and previous
expected SQL queries to stop any query that is described as a cautious query.

REFERENCES


