Scientific Journal of Impact Factor(SJIF): 3.134

e-ISSN(O): 2348-4470
p-ISSN(P): 2348-6406

International Journal of Advance Engineering and Research Development

Volume 2,Issue 5, May -2015

Methods of Detection and Prevention of SQL Injection with Some Java Code

JinalPatel

Computer And Science Department,Saffrony Institute of Technology ,Linch,Mehsana

Abstract:_*In this paper, we are presenting a fundamentals of SQL Injection. Also we will discuss the types of SQL Injection. Then we will detect the SQL injection and provide the detection algorithm includes these steps: lexicalanalysis of source code, parsing of source code, constructingabstract syntax tree of source code, defining rules of SQL injection attack. And the methods to prevent the SQL injection.*

Keywords: *SQL Injection, SQL Injection Detection, SQL Injection Prevention, Prepared statement, SQL Queries.*

I. INTRODUCTION

What is SQL Injection?

A SQL injection attack consists of insertion or "injection" in a SQL query using 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.

Threat Modeling:

- 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.
- SQL Injection is very common with PHP and ASP applications due to the prevalence of older functional interfaces. Due to the nature of programmatic interfaces available, J2EE and ASP.NET applications are less likely to have easily exploited SQL injections.
- The severity of SQL Injection attacks is limited by the attacker's skill and imagination, and to a lesser extent, defense in depth countermeasures, such as low privilege connections to the database server and so on. In general, consider SQL Injection a high impact severity.

SQL injection errors occur when:

1. Data enters a program from an untrusted source.
2. The data used to dynamically construct a SQL query

The main consequences are:

- **Confidentiality**: Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.
- **Authentication**: If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- **Authorization**: If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL Injectionvulnerability.
- **Integrity**: Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL Injection attack.

II. METHODS

There are many methods of SQL injection some of them which are simply understandable are as below:

- **Incorrectly filtered escape characters:**

This form of SQL injection occurs when user input is not filtered for escape characters and is then passed into a SQL statement. This results in the potential manipulation of the statements performed on the database by the end-user of the application.
The following line of code illustrates this vulnerability:
statement = "SELECT*FROM users WHERE name ='" + userName + "';"
This SQL code is designed to pull up the records of the specified username from its table of users. However, if the "userName" 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 "userName" variable as:
‘ or ‘1’=’1
or using comments to even block the rest of the query (there are three types of SQL comments). All three lines have a space at the end:
' or '1'='1' –
' or '1'='1' ({
' or '1'='1' /*
renders one of the following SQL statements by the parent language:

```
SELECT*FROM users WHERE name ='OR'1'='1';
SELECT*FROM users WHERE name ='OR'1'='1'-- ';
```

If this code were to be used in an authentication procedure then this example could be used to force the selection of a valid username because the evaluation of '1'='1' is always true.

- **Incorrectly Type Handling:**

This form of SQL injection occurs when a user-supplied field is not strongly typed or is not checked for type constraints. This could take place when a numeric field is to be used in a SQL statement, but the programmer makes no checks to validate that the user supplied input is numeric.
For example:
statement := "SELECT*FROMuserinfoWHERE id =" + a_variable + ";"
It is clear from this statement that the author intended a_variable to be a number correlating to the "id" field. However, if it is in fact a string then the end-user may manipulate the statement as they choose, thereby bypassing the need for escape characters. For example, setting a_variable to
1;DROP TABLE users
will drop (delete) the "users" table from the database, since the SQL becomes:

```
SELECT*FROMuserinfoWHERE id=1;DROPTABLE users;
```

- **Blind SQL Injection :**

Blind SQL Injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page. This type of attack can become time-intensive because a new statement must be crafted for each bit recovered. There are several tools that can automate these attacks once the location of the vulnerability and the target information has been established.

Conditional responses

One type of blind SQL injection forces the database to evaluate a logical statement on an ordinary application screen. As an example, a book review website uses a query stringto determine which book review to display. So the URL **http://books.example.com/showReview.php?ID=5** would cause the server to run the querySELECT*FROMbookreviewsWHERE ID ='Value(ID)';from which it would populate the review page with data from the review with ID 5, stored in the table bookreviews. The query happens completely on the server; the user does not know the names of the database, table, or fields, nor does the user know the query string. The user only sees that the above URL returns a book review. A hacker can load the URLs http://books.example.com/showReview.php?ID=5 OR 1=1 and

http://books.example.com/showReview.php?ID=5 AND 1=2, which may result in queries

```
SELECT*FROMbookreviewsWHERE ID ='5'OR'1'='1';
SELECT*FROMbookreviewsWHERE ID ='5'AND'1'='2';
```

respectively. If the original review loads with the "1=1" URL and a blank or error page is returned from the "1=2" URL, and the returned page has not been created to alert the user the input is invalid, or in other words, has been caught by an input test script, the site is likely vulnerable to a SQL injection attack as the query will likely have passed through successfully in both cases. The hacker may proceed with this query string designed to reveal the version number of MySQL running on the server:
http://books.example.com/showReview.php?ID=5 AND substring(@@version,1,1)=4,
which would show the book review on a server running MySQL 4 and a blank or error page otherwise. The hacker can continue to use code within query strings to glean more information from the server until another avenue of attack is discovered or his or her goals are achieved

- **Second Order SQL Injection :**

Second order SQL injection occurs when submitted values contain malicious commands that are stored rather than executed immediately. In some cases, the application may correctly encode a SQL statement and store it as valid SQL. Then, another part of that application without controls to protect against SQL injection might execute that stored SQL statement. This attack requires more knowledge of how submitted values are later used. Automated web application security scanners would not easily detect this type of SQL injection and may need to be manually instructed where to check for evidence that it is being attempted.

III. DETECTION

ORDINARY SQL INJECTION ATTACK DETECTION ATTACK AND DEFENSE

Ordinary SQL injection attack main detection techniques are as follows:

(1) White box detection: It is to check static web pagein order to search for all paths where SQL injection attack maygenerate by static analysis and to inspect the quality of webpage that being published before. For example, Fortify .

(2) Black box detection: It is to use a rule library tosimulate hacker attacking application program and to pinpointthe problems by analyzing the results that the applicationprogram perform. For example, AppScan.

PRINCIPLE OF STATIC ANALYSIS SCANNING

Programming static analysis is not to execute the programand to find potential safety hazard in program with thealgorithm automatically scanning. It has the followingadvantages: quickly performing and high effectively etc.Source code with static analysis employs lexical analysis and
parsing of compiling technology. The principle of staticanalysis is illustrated in Figure 1.

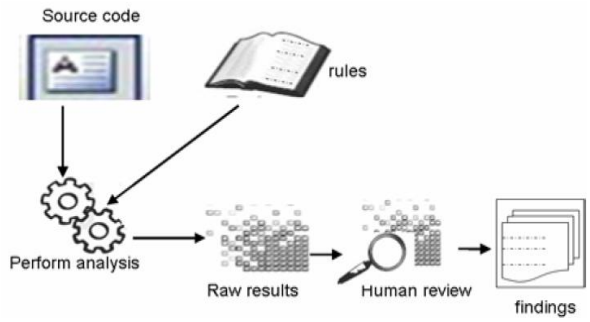

Figure1. The principle of source code with static analysis

Algorithm:

The proposed algorithm of Java source-code SQLinjection attack detection.

The grammar definition of version Java1.6 is pointed out.Based on the principle of static analysis, with lexical analysisof source code and parsing of source code, construct abstract
syntax tree of source code according to program structure.Different nodes on the abstract syntax tree are marked byspecial signs in order to improve the efficiency of the proposedalgorithm. The proposed SQL injection attack detectionalgorithm includes the following steps:

Step 1. Abstract syntaxtree traversal .
To search for allnodes named executeQuery , executeUpdate, execute orexecuteBatch under the branch node of METHOD_DECL andto store them in the Hash keyNode table.

Step 2. To confirm whether each of all nodes in the HashkeyNode table is at- risk or not .
Step 2.1 Firstly, to gain the former expression of thekeyNode and to obtain types of return value about the formerexpression.
Step 2.2 If the type of return value about the formerexpression.is java.sql.Statement, this node can be validated.
Then go to next step; Othewise, go to Setp 2 and analyze thenext node.

Step 3. Obtain the first parameter as the path of node aboutconfirmed nodes, analyze and track data flow of nodes on thepath:
Step 3.1. Analyze and track nodes that are variablesexpression on the path
Step 3.2. Analyze and track nodes that are methodological expression on the path.
Step3.3. End the condition of tracking:
(1) If all the tracking variables are constant, there is notSQL injection attack on the path and go to Step 2.
(2) Track the methodological nodes that are APIdefined ,and go to Step 4.

Step 4. Record and store paths of QL injection attack andgo to Setp2,analyze the next the node. The algorithm stopswhen all nodes are detected in the keyNode table.

```
importjava.sql.Statement;
importjava.sql.ResultSet;
importjavax.servlet.http.HttpServletRequest;
public class Test
{
        Statement statement = new Statement();
        public void testMethod(HttpServletRequest request);
        {
                StringBuffersqlStatement = newStringBuffer("select * from employee where userid=");
                String id = request.getParameter("userid");
                if (id != null)
                        sqlStatement.append(id);
                else
                        sqlStatement.append("");
                        ResultSet results =statement.executeQuery(sqlStatement.toString());
        }
}
```

The Above algorithm includes these steps such as lexical analysis of sourcecodes, parsing of source codes, constructing abstract syntaxtree of source code, defining rules, abstract syntax tree traversal,etc. Finally, Use experiments to evaluate the performance ofthe algorithm. Test results show the proposed algorithmperforms perfectly. Nevertheless, because of various means of
SQL injection attack and technological updating of SQLinjection attack, hence language rules must be studied furtherso as to greatly raise recognition rate and reduce misstatementrate.

IV. PREVENTION

To 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.
This article provides a set of simple techniques for preventing SQL Injection vulnerabilities by avoiding these two problems. These techniques can be used with practically any kind of programming language with any type of database. There are other types of databases, like XML databases, which can have similar problems (e.g., XPath and XQuery injection) and these techniques can be used to protect them as well.

At present, there are main two means about SQL injection attack defense, such as platform level defense and code level defense. The common defense techniques are as follows:

Primary Defenses:

- Use of Prepared Statements (Parameterized Queries)
- Use of Stored Procedures
- Server filtering
- Restrict database permission
- Data Fitering

Additional Defenses:

- Also Enforce: Least Privilege
- Also Perform: White List Input Validation

Unsafe Example
SQL injection flaws typically look like this:
The following (Java) example is UNSAFE, and would allow an attacker to inject code into the query that would be executed by the database. The unvalidated "customerName" parameter that is simply appended to the query allows an attacker to inject any SQL code they want. Unfortunately, this method for accessing databases is all too common.

```
String query = "SELECT account_balance FROM user_data WHERE user_name= "
  + request.getParameter("customerName");

try {
        Statement statement = connection.createStatement( … );
        ResultSet results = statement.executeQuery( query );
}
```

V. PRIMARY DEFENSE

- **Use of Prepared Statements (Parameterized Queries)**

The use of prepared statements (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

In rare circumstances, prepared statements can harm performance. When confronted with this situation, it is best to either a) strongly validate all data or b) escape all user supplied input using an escaping routine specific to your database vendor as described below, rather than using a prepared statement. Another option which might solve your performance issue is to use a stored procedure instead.

Safe Java Prepared Statement Example

The following code example uses a PreparedStatement, Java's implementation of a parameterized query, to execute the same database query.

```
String custname = request.getParameter("customerName"); // This should REALLY be validated too
 // perform input validation to detect attacks
 String query = "SELECT account_balance FROM user_data WHERE user_name= ? ";

PreparedStatementpstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);

ResultSet results = pstmt.executeQuery( );
```

- **Use of Stored Procedures**

Stored procedures have the same effect as the use of prepared statements when implemented safely*. They require the developer to define the SQL code first, and then pass in the parameters after. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.

Safe Java Stored Procedure Example
The following code example uses a CallableStatement, Java's implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

```
 String custname = request.getParameter("customerName"); // This should REALLY be validated
try {
        CallableStatementcs = connection.prepareCall("{call sp_getAccountBalance(?)}");
        cs.setString(1, custname);
        ResultSet results = cs.executeQuery();
        // … result set handling
 } catch (SQLException se) {
        // … logging and error handling
 }
```


- **Server filtering**:
  It is to embed filtering modulesweb servers. Filtering modules find malicious codes andprevent web server from SQL injection attack, according tofiltering rules analyzing Http input request. Its advantage lies indefending SQL injection attack.

- **Restrict database permission:**
  Database permission islimited in some application program, which may cut downvarious kinds of hackers attacking database.

- **Data filtering:**
  It is to validate user input data and filterdata by using white list as far as possible, or by using blacklist.or example, filtering dangerous single quote character etc.

VI. ADDITIONAL DEFENSES

- **Least Privilege:**

To minimize the potential damage of a successful SQL injection attack, you should minimize the privileges assigned to every database account in your environment. Do not assign DBA or admin type access rights to your application accounts. We understand that this is easy, and everything just 'works' when you do it this way, but it is very dangerous. Start from the ground up to determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access to the tables they need access to. If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data and assigning the account access to the view instead, rather than the underlying table. Rarely, if ever, grant create or delete access to database accounts.
If you adopt a policy where you use stored procedures everywhere, and don't allow application accounts to directly execute their own queries, then restrict those accounts to only be able to execute the stored procedures they need. Don't grant them any rights directly to the tables in the database.
SQL injection is not the only threat to your database data. Attackers can simply change the parameter values from one of the legal values they are presented with, to a value that is unauthorized for them, but the application itself might be authorized to access. As such, minimizing the privileges granted to your application will reduce the likelihood of such unauthorized access attempts, even when an attacker is not trying to use SQL injection as part of their exploit.
While you are at it, you should minimize the privileges of the operating system account that the DBMS runs under. Don't run your DBMS as root or system! Most DBMSs run out of the box with a very powerful system account. For example, MySQL runs as system on Windows by default! Change the DBMS's OS account to something more appropriate, with restricted privileges.

- **White Space Input Validation:**

Input validation can be used to detect unauthorized input before it is passed to the SQL query.

VII. REFERENCES

[1] Wang Tian, Wei Lihao, Zou Hong," A Java Source-code SQL Injection Attack DetectionAlgorithm Based on Static Analysis ", National Conference on Information Technology and Computer Science (CITCS 2012)
[2] William G J, Viegas H J, Orso A. A Classification of SQL InjectionAttacks and Countermeasures[C]//Proc. of International Symposium onSecure Software Engineering. Arlington, USA: IEEE Press, 2006.
[3] Su Zhendong, Wassermann G. The Essence of Command InjectionAttacks in Web Applications[C]//Proc. of Annual Symposium onPrinciples of Programming Languages. Charleston, USA: [s. n.],2006.
[4] Stuttard D, Pinto M. The Web Application Hacker's Hand book:Discovering and Exploiting Security Flaws [M]. Beijing: People'sTelecon Publishing House, 2009.
[5] Zhang zhuo, SQL injection attack techniques and countermeasuresanalysis [D].Shanghai Jiao tong university.2007, 50-51

