IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 1 Issue 3, May 2014.
www.ijiset.com
ISSN 2348 – 7968

Multidimensional Analysis of SQL Injection Attacks in Web Application

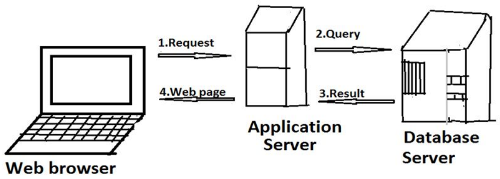
A.VANITHA, Dr.N.RADHIKA
Assistant professor/CSE, Sri venkateshwara college of arts and science, Peravurani.Thanjavur(Dt)
Associate professor/CSE, Amrita vishwa vidyapeedham, Amrita Nager, Coimbatore.

INTRODUCTION

In recent years, wide spread adoption of the internet has resulted in to rapid advancement in information technologies. The internet is used by the general population for the such as financial transactions, educational endeavor, and countless other activities. The use of internet for accomplishing important task, such as transfer a balance from a bank account, always comes with a security risk. The database system behind this secure websites store non-critical data along with sensitive information, in a way that allows the information owners quick access while blocking break - in attempts from unauthorized users.A common break -in strategy is to try to access sensitive information from a database by generating a query that will cause the database parse to malfunction, followed by applying this query to the desired database .Such an approach to gaining access to private information is called SQL injection.

To get a better understanding of SQL injection, we need to have a good understanding of the kinds of communications that takes place during a typical session between user and a web application. The following figure shows the typical communication exchange between all the components in a typical web application system.

WEB APPLICATION ARCHITECTURE



A web application, based on the above model, takes text as input from users to retrieve information from a database. some web applications assume that the input is legitimate and use it to build SQL queries to access a database. Since these web applications do not validate user queries before submit them to retrieve data, they become more susceptible to SQL injection attacks. For example, attackers, posing as normal users, use maliciously crafted input text containing SQL instructions to produce SQL instructions to produce SQL queries on the web application end. Once processed by the web application, the accepted malicious query may break the security policies of the underlying database architecture because the result of the query might cause the database parser to malfunction and release sensitive information.

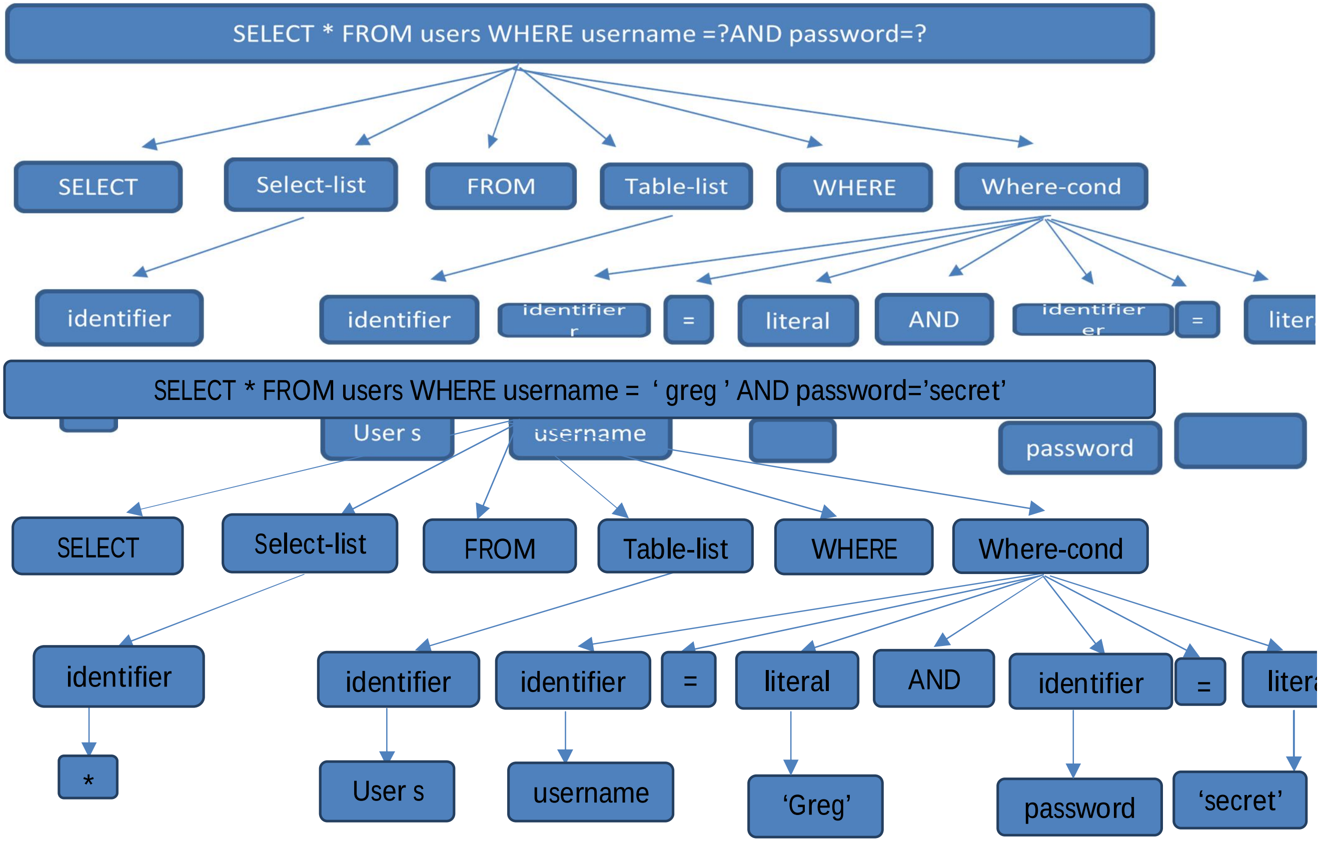
SQL INJECTION DISCOVERY TCHNIQUE

It is not compulsory for an attacker to visit the web pages using a browser to find if SQL injection is possible on the site. Generally attackers build a web crawler to collect all URLs available on each and every web page of the site. Web crawler is also used to insert illegal characters into the query string of a URL and check for any error result sent by the server. If the server sends any error message as a result, it is a strong positive indication that the illegal special Meta character will pass as a part of the SQL query and hence the site is open to SQL injection attack.

SQL PARSE TREE VALIDATION

A parse tree is nothing but the data structure built by the developer for the parsed representation of a statement. To parse the statement, the grammar of that parse statement’s language is needed. In this method, by parsing two statements and comparing their parse trees, we can check if the two queries are equal. When attacker successfully injects SQL into a database query, the parse tree of the intended SQL query and the resulting SQL query generated after attacker input do not match. The following figure shows the representation of a parse tree.

A SELECT QUERY WITH TWO USER INPUTS



In the above parse tree the programmer- supplied portion is hard-coded, and the user-supplied portion is represented as a vacant leaf node in the above parse tree. A leaf node must be the value of a literal, and it must be in the position where vacant space is located. The SQL query for the above parse tree is as below.SELECT * FROM user WHERE username=? AND password=? The question marks are place holders for leaf nodes.

APPROACH FOR SQL CHECK:

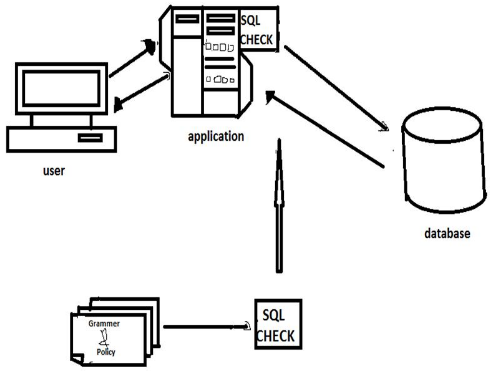
A JSP PAGE FOR RETRIEVING CREDIT CARD NUMBERS.

```
<%!
        // database connection
    info
String dbdriver=”com.mysql.jdbc.driver”;
String strconn=”jdbc:mysql://”
+”sport4sale.com/sport”;
String dbuser = “manager”;
String dbpassword = ‘’atnltpass’’;
//generate query to send
String sanitized name=
Replace(request.getparameter(“name”),”’”,”
””);
String sanitized card type=
Replace(request.getparameter(“cardtype”),”’
”,”””);
String query=’SELECT cardnum FROM
accounts”
+”WHERE
uname=’”+sanitizedcardtype+”;”
        try{// connect to
    database and send query
java.sql.drivermanager.registerdriver(
(java.sql.driver)
(class.forName(dbdriver).newInstance()));
Java .sql.connection conn=
Java.sql.drivermanager.getconnection(
Strconn, dbuser, dbpassword);
Java.sql.statement stmt=
Conn.create statement();
Java.sql.resultset rs=
Stmt.executeQuery(query)
        //generate html output
        Out.println(“<html><
    body><table>”)
        While(rs.next()){out.p
    rintln(“<tr><td>”);
        Out.println(rs.getstrin
    g(1))
        Out.println(“</td></tr
    >”)
        }if(rs!=null)
    {rs.close();
        }out.println(“</table>
    </body></html>”);
        }catch(Exception e)
        {out.println(e.tostring
    ());}%>
```

Web applications have SQL injection vulnerabilities because they do not sanitize the inputs they use to construct structured output. Consider the snippet shown in the above program.

The Code is for an online store. The website provides user input field to allow the user to keep their credit card information which user can use for future purchase. Replace method is used to escape the quotes so that any single quote characters in the input considered as a literal and not a string delimiters. Replace method is intended to block attacks by preventing an attacker from ending the string and adding SQL injection code. Although, card type is a numeric column, if an attacker passes 2 OR 1=1.As the card type, all account numbers in the database will be returned and displayed.

SYSTEM ARCHITECTURE OF SQLCHECK



In this approach they track through the program, the substrings receive from user input and sanitize that substrings syntactically. The aim behind this program is to block the queries in which the input substrings changes the syntactic structure of the rest of the query. They use the meta-data to watch users input, displayed as ‘_’ and ‘_,’ to mark the end and beginning of the each user input string. This meta-data pass the string through an assignments, and concatenations, so that when a query is ready to be sent to the database, it has a matching pairs of markers that identify the substring from the input. These annotated queries called an augmented query. To build a parser for the augmented grammar and attempt to parse each augmented query steve use a parse generator. Query meets the syntactic constrains and considered legitimate if it parses successfully. Else, it fails the syntactic constrains and interprets it as an SQL injection attack. In spite of the inputs source, each input which is to be passed into some query, gets augmented with the meta-characters ‘_’and’_,’. Finally application creates augmented queries, which SQLCHECK attempts to parse, and if a query parses successfully, SQLCHECK sends it the meta-data to the database, else the query get rejected.

BACKGROUND FOR SQL STATEMENT

This section gives a brief idea about the SQL injection vulnerability and a related SQL injection attacks. SQL injection vulnerability means the combination of dynamic SQL statement combination compilation and a weak in input validation. This input validation forces input to change the structure of a SQL query. Such combinations are generally found in java. Following examples shows the code that initially have plain text SQL statement which dynamically produces the SQL query based on a variable input (user ISBN). Moreover, without any input verification it creates the SQL query with use of string concatenation

“Statement stmt =’’

“conn.createStatement();’’

“ResultSet rs =’’
‘

“stmt.executeQuery(“select amount from’’
“books where isbn=’’’+userISBN+’’’’’);’’

SQL INJECTION ATTACKS PREVENTION APPROACH

Model-based guard constructor prevention is an efficient method in preventing an SQL injection attack. This method is established on breaking the suitable conjunction of input, code, data, and database access situation that would employ an SQL injection attack. Spontaneously inserting appropriate guards before allowing the access to the database, we can avoid an SQL injection attack. As shown in the figure 6, initially instrument the PHP string to collect the samples of query which authentically used at database application program interface call point. These queries called as a set of trusted test cases. From the flow of the diagram, we can easily understand the prevention of an SQL injection attack. Instrumentation is nothing but to add an output instruction database application interface calls, as below.

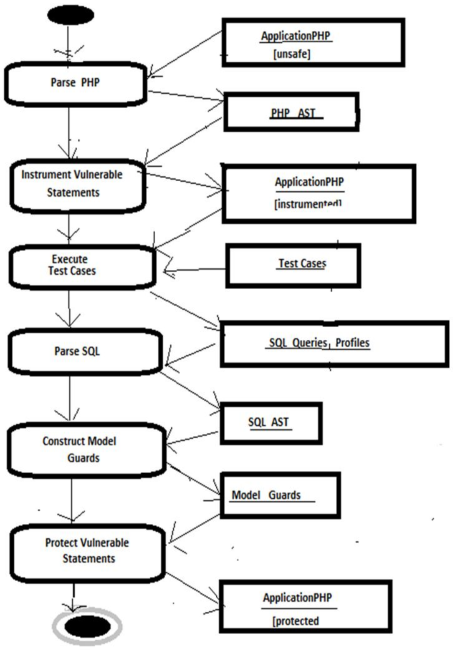
```
Sql_query(…EXPRESSION…);
```

After passing this expression through automated approach it becomes:

```
$string = Expression;
fRead($file handle; $string));
$result = sql_query($string);
```

After running the trusted test cases to gather the plain text strings that are produced dynamically at various call sites matching to trusted queries. It is a straight forward to create model guards from sets of ASTs leading to legitimate queries.

AUTOMATED PROTECTION



ASTs are generalized by type rather than image, because constants, strings and additional types of data are also stored in the ASTs. On the other hand, application dependent identifiers, such as the names of the tables, number of columns, rows, are counted as a part of syntactic structure of the SQL query which plays crucial role to prevent malicious substitution of table or column names in the valid queries. Therefore this method permits number of queries with same syntactic structure, but with different values of data. Using special call site, model guard invokes the SQL parser on the database, where we are working currently to and obtains the matching SQL AST. Generally "ASTs" are stored as token strings containing token types where an application table names and file names have become keywords.

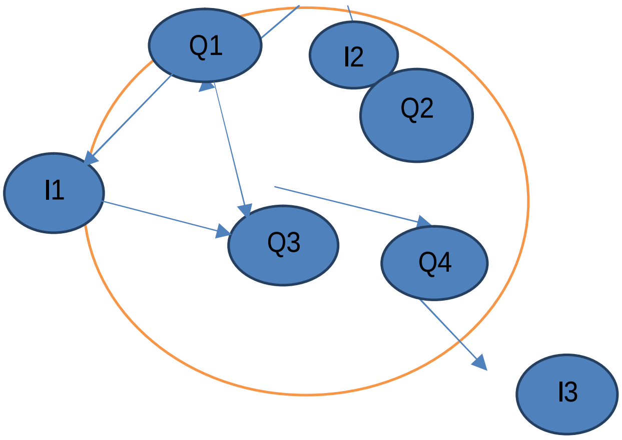
PREVENTING SQL INJECTION METHOD

STATIC ANALYSIS:

In static analysis authors provides the parser called stored procedure parser which is used to extracts the "CONTROL FLOW GRAPH" from the saved procedures, we can see in detail about the control graph in following section. At the start, we label every execution statement in the control flow graph and then use the backtracking method to verify all statements participated in the formation of the SQL statements in the control flow graph. In the SQL graph, statements which are depended on the users input are screened and flags are sent on it to monitor their behavior at run time. In this method, using finite state automaton, we compare the statement with dynamically created SQL statement of user inputs which tries to change the original SQL statement.

The statement created users input which tries to change the original pattern of the parser will indicated by flag as dangerous statement and provides the related information. following figure gives a clear understanding of static analysis. Four different SQL queries Q1, Q2, Q3,and Q4 are in the stored procedure shown as nodes within a boundary displayed in dotted circle. Suppose a user enter the input I in the SQL query Q and the relationship between input I and query Q is represented by R. D represents the dependencies in SQL diagram that links the one SQL query to another. The user input 'I' accepted by previous query is transfer to another query through the dependency link.

SQL CONTROL GRAPH



Q1,Q2,Q3,Q4 I1,I2,I3	SQL QUERIES USER Inputs
R1	I1 € Q1
R2	I2 € Q1
R3	I1 € Q3
R4	I2 € Q2
R5	I3 € Q4
D1	Q3 (Q1
D2	Q2 (Q1

ADVANTAGES OF STATIC ANALYSIS

1) SQL graph representation used to reduce the runtime scanning overhead of program by preventing the number of queries that are not require to execute in stored procedure.
2) SQL control graph does not include the query which does not take an input from user.
3) The queries which includes input from user to access the database information are counted towards SQL control graph representation.

Conclusion

SQL Injection attacks are one of the most dangerous types of threats to web applications. Many solutions to these attacks have been proposed over years. But almost none of them provide security to the full extent of this attack. Also very little emphasis is laid on preventing SQLIA in stored procedures. The proposed solutions for preventing or detecting SQLIA provide security to either application layer or database layer but not to both. We have proposed a technique that provides security to both application layer as well as database layer via frontend phase and backend phase. As a result, industry is paying increased attention to the security of the web applications themselves in addition to the security of the underlying computer network and operating systems Limiting the permissions on the database logon used by the web application to only what is needed may help reduce the effectiveness of any SQL injection attacks that exploit any bugs in the web application.

REFERENCES

[1]. William G.J. Halfond, T. Viegas and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," in Proc. of ISSSE06, 2006.

[2]. Dr. M. Amutha Prabakar, M.KarthiKeyan, Prof.K. Marimuthu, "An Efficient Technique for Preventing SQL Injection Attack Using Pattern Matching Algorithm," in Proc. of ICECCN, 2013.

[3]. Gao Jiao, Chang-Ming XU and Jing Maohua, "SQLIMW: a new mechanism against SQL-Injection," in Proc. of CSSS, 2012.

[4]. William G.J. Halfond and Alessandro Orso, "AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks," in Proc. of ASE, 2005, p. 174–183.

[5]. Sruthi Bandhakavi, Bisht, P. Madhusudan, V.N. Venkatakrishnan, "CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations," in Proc. of CCS'07, 2007.

[6]. S. W. Boyd and A. D. Keromytis, "SQLRand: Preventing SQL injection attacks," in Proc. of ACNS, 2004.

[7]. Ke Wei, M. Muthuprasanna and Suraj Kothari, "Preventing SQL Injection attacks in stored procedures," in proceedings of ASWEC, 2006.

[8]. Amit Kukreti. (2005) SQL Injection Attacks homepage on codeproject. [Online]. Available: http://www.codeproject.com/Articles/11020/SQL-injection-attacks/

[9]. (2013) Malicious SQL Injection: an introduction homepage on hackmac. [Online]. Available: http://www.hackmac.org/tutorials/malicious-sql-injection-an-introduction/