Volume 2, Issue 6, June 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

SQL INJECTIONS – A HAZARD TO WEB APPLICATIONS

Neha Singh
Guru Nanak Dev Co-ed Polytechnic,
New Delhi, India
neha.s02dec@gmail.com

Ravindra Kumar Purwar
University School of Information Technology, Guru Gobind Singh Indraprastha University,
New Delhi, India

Abstract — ***With changing times, our dependence on the web applications for the fulfilment of our daily needs (like online shopping, banking, share trading, ticket booking, payment of bills etc.) has increased. Because of this, our confidential data is present in the databases of various applications on Web. The security of this myriad amount of data is a matter of major concern. In recent times, SQL Injection attacks have emerged as a major threat to database security. In this paper we define SQL Injections, illustrate how SQL Injections are performed. In addition we have also surveyed the various SQL Injection detection and Prevention tools and well-known attack methods. Finally, we have provided our solution to the problem and have assessed its performance.***

Keywords — ***SQLIA, Database, Web application, SQL query, attacker.***

I. INTRODUCTION

The numerous advances in technology have simplified many of our daily tasks. With multiple services available via a single click through various Web Applications, we don't have to stand in long queues at the banks or have to go to the markets to shop for the latest trends. The gaining popularity of the Web Applications has drawn the attention of many hackers. With so much personal data scattered over the Web in various databases, hackers can certainly harm many lives by gaining access to it or by making changes to it. For instance, if a hacker obtains the bank account details of an individual, he can misuse this information (like account number, account balance, loan amount, etc.) and can also alter the data to cause harm to the concerned individual.
SQL (Structured Query Language) is a common language used to insert, retrieve, update and delete data from the databases. When we enter our information (like login credentials etc.) in the input fields provided on the web form of a Web Application, it forms the part of the SQL query written at the backend, to be executed on the database. For instance, when we login in our mailbox, we provide username and password. The username and password form the part of the internal SQL query. Then the SQL query is executed on the database to check whether the login credentials provided match with those present in the tables on the database. The attacker, who is not aware of the login credentials but, wants to gain access to the mailbox by unfair means, provides SQL code instead of correct input in the test fields of the web form. This malicious code changes the structure of the original SQL query and consequently, allows the attacker to gain access to the information it was not authorized for. This type of attacks, which allow the attacker to change the meaning of the original SQL query by inputting

illegitimate SQL code from the front end of the Web application are termed as SQLIAs (SQL Injection Attacks).
SQLIAs pose a serious threat to the data security of the Web applications. These are the most popular and harmful attacks used by the hackers to attack databases. In SQLIAs, the attacker provides SQL code instead of the legitimate input in the input fields of the web form in order to change the meaning of the original SQL query issued by the database at the backend. Once the attacker gains access to the database, it can alter any data.

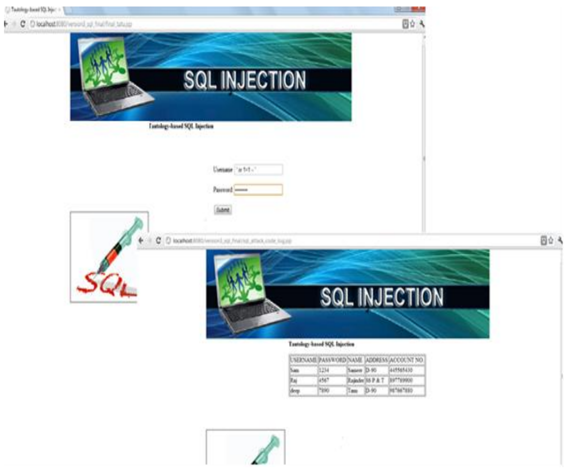
II. DEFINITION

SQL Injections are attacks by which an attacker alters the structure of the original SQL query by injecting SQL code in the input fields of the web form in order to gain unauthorized access to the database.

III. ILLUSTRATION OF A SIMPLE SQL ATTACK

There are several methods which an attacker can use to gain access to the database of a Web Application. We have developed a tool using Apache Tomcat and Oracle to illustrate SQLIAs.
Tautology is a conditional statement which always evaluates to true. This type of method is used to attack the SQL queries which use the 'WHERE' clause. Here the intention of the attacker is to make the condition in the query as true. To do so, the attacker adds the most common tautology condition '1=1' in the input field of the web form. Tautology attacks are illustrated in fig.1 below:

FIGURE I
ILLUSTRATION OF SQL ATTACK



USERNAME	PASSWORD	NAME	ADDRESS	ACCOUNT NO
Sam	1234	Sameer	D-90	[illegible]
Raj	4567	Rajinder	58 P & T	[illegible]
deep	7890	[illegible]	D-90	[illegible]

The SQL query being executed at the backend is **'Select * from login where Username = ' ' or 1 = 1 -- ' and Password= 'xxxxx''**

The single quote entered by the attacker closes the Username field and the double dashes comments out everything after the dashes. Therefore, the query retrieves all the records in the LOGIN table and returns them back to the attacker.

IV. BACKGROUND

Research has been going since long to mitigate the effects of SQLIAs and consequently many detection and prevention tools have been developed over the years. We've surveyed some of these tools and have summarized them in the table 1 below. ([3],[5],[8]-[15]).

TABLE II
SQL INJECTION DETECTION AND PREVENTION TOOLS

CANDID	It is a tool developed to guard Web applications in Java language against SQL Injection attacks. It uses candidate inputs to dynamically infer about the programmer intended query structure. Candid consists of two components: an offline Java program transformer and an online SQL parse tree checker.
AMNESIA	Detection and prevention technique, which uses static and dynamic analysis in combination. During static analysis, it predicts the legitimate queries that can be generated by the application. During dynamic analysis, it uses runtime monitoring to check the queries generated in static analysis against the actual set of generated queries.
Positive Tainting	Dynamic approach to detect and prevent SQL injections by performing dynamic tainting. Firstly, it finds and highlights the trusted data. Then it performs accurate taint propagation by highlighting the trusted data at character level. Finally, it performs syntax-aware evaluation of the queries, where all queries containing untrusted characters are blocked from execution on the database.
SQL Rand	The concept of Instruction-Set randomization is applied to the SQL language to detect and abort queries which contain injected code. Here, each SQL keyword is joined with a random integer to mislead the attacker.
SQL DOM	Object oriented model in which SQL queries are generated by manipulating objects which are strongly-typed to the database. It inspects the dynamically generated queries at of compile time.
Viper	A tool used for Web Application penetration testing which uses heuristic approach for detecting SQL Injections.
SQL-Prob	SQL Proxy-based Blocker which fetches the user input from the SQL query of the application and checks it against the syntactic structure of the query.
ADMIRE	It is a threat risk model which provides a thorough and step-by-step technique to identify and moderate the effect of SQL Injections.
WAVES	A Black box technique which searches for vulnerable locations in a Web application using a Web Crawler and then builds attacks which target these locations. Finally, it watches the responses of the Web application to these attacks using machine learning techniques.
JDBC-Checker	It is a static checking technique which checks for the correctness of the dynamically-generated SQL queries.

V. SQL INJECTION ATTACK METHODS

There are multiple methods by which a Web application can be attacked. We've discussed each of these methods in detail to illustrate how each of them is used to attack the database of the application. ([1],[2],[4],[6],[9]).

Tautology based SQL Injections - It is used to inject a tautology statement (e.g. "1=1") to the conditional SQL query so that it evaluates to true always. SQL query containing 'WHERE' clause is vulnerable to this kind of attack.
Example -
Original Query- ' Select accountno, balance from accounts where loginid='abc' and pwd='xxxxx''
Injected Query- 'Select accountno, balance from accounts where loginid=' ' or 1=1 --' and pwd='not required''
Result- It shows the account number and balance amount for all values of loginid from the accounts table.

Statement Injection - It is used to inject a new SQL query to the original SQL query, through the front-end of an application using query delimiter ";".
Example -
Original Query- 'Select * from employee where empid='1234' and password='xxxx''
Injected Query- 'Select * from employee where empid=' '; Delete from employee where empid= '1234' --' and password='not required''
Result- It deletes the record for empid=1234 from employee table.

Stored Procedures - A Stored Procedure consists of precompiled set of SQL statements referred by a single name. As a stored procedure is coded by the programmer, it is as vulnerable to SQL injections as the other Web Applications.
Example -
Consider the stored procedure below:

```
CREATE PROCEDURE new_dept(new IN varchar2, old IN varchar2)
IS
        line varchar2(8000);
BEGIN
        line:='begin
                update department set dept=''' || new || ''' where dept= '''|| old || ''';' || 'END;';
        DBMS_OUTPUT.PUT_LINE('line: ' || line);
        EXECUTE IMMEDIATE line;
END;
```

This procedure has two input fields, old department name and new department name and replaces old name with the new one. The attacker injects the code [' ';SHUTDOWN;--] in either of the two fields. This injection generates the following query:
Update department Set dept ='abc';SHUTDOWN;-- where dept='aaa'
At this stage, the attack behaves like the statement injection attack where the injected query is made to execute with the original query using query delimiter ';'.

Illogical/Incorrect queries - The attacker deliberately inputs incorrect information in order to gather information about the internal database structure of the application, through the displayed error messages.
Example - Consider the stored procedure below:

```
CREATE PROCEDURE sales
  (tot OUT number,dt in date)
  IS
  line CONSTANT VARCHAR2(4000) :='select sum(price) from orders where odate+30>'''||dt||'''';
   BEGIN
      DBMS_OUTPUT.PUT_LINE('line: ' || line);
      EXECUTE IMMEDIATE line into tot;
      DBMS_OUTPUT.PUT_LINE('total sales:'||tot);
   END;
```

It requires odate as input from the user. The attacker inputs date as '5a', which is an incorrect date format. The error displayed is
javax.servlet.ServletException: ORA-06550: line 1, column 7:
PLS-00306: wrong number or types of arguments in call to 'SALES'
ORA-06550: line 1, column 7:
PL/SQL: Statement ignore….
Two important pieces of information can be inferred from the above error. First, the name of the procedure is SALES and second, the name of the database is Oracle.

Union query - The original SQL query is concatenated to the injected query by using the SQL keyword 'UNION', to gather information from other tables related to the application.
Example -
Original Query- 'Select salary from employee where empid='1234''
Injected Query- 'Select salary from employee where empid=' ' Union select * from employee'
Result- It will return all the records from the employee table.

Alternate Encodings - The attacker uses Alternate Encodings like, ASCII, Unicode, EBCDIC and Hexadecimal to inject code so that it can bypass the validations on the input, if any.
Example -
Original Query- 'Select * from login where username = 'a123' and pwd='xxx''
Injected Query- 'Select * from login where username = ' '; exec(char(0x73687574646f776e)) --' and pwd='not required''
Result- The value passed to the char() function is the hexadecimal encoding for SHUTDOWN. Thus as the above injection uses hexadecimal encoding instead of actual characters, it will bypass the input validations and will cause the SHUTDOWN command to be executed.

Inference - These types of attacks are framed where the applications have input validations and thus it is relatively difficult for the attacker to deduce information about the vulnerable parameters and structure of the database. Still, attacker tries to gain information from the application by changing the behaviour of the queries. These attacks are bifurcated into two major types: Blind Injection and Timing Attacks.
Blind Injection: The query is transformed into a group of true/false questions in order to fetch information related to the behavior of the application.
Example -
Original URL -
http://www.myarea.com/value/value.asp?value=111
Injected URL-
http://www.myarea.com/value/value.asp?value=111 and '1=0'
Result- If there is no input validation present on the application; the following error message will be produced.
Select *from users where userid=111 and '1=0'
From the message, we can infer that table name is 'users' and field name is 'userid'. But if the application has incorporated input validations, then no error will be returned by the application. This piece of information

will be useful for the attacker to sight the vulnerable parameters in the application.

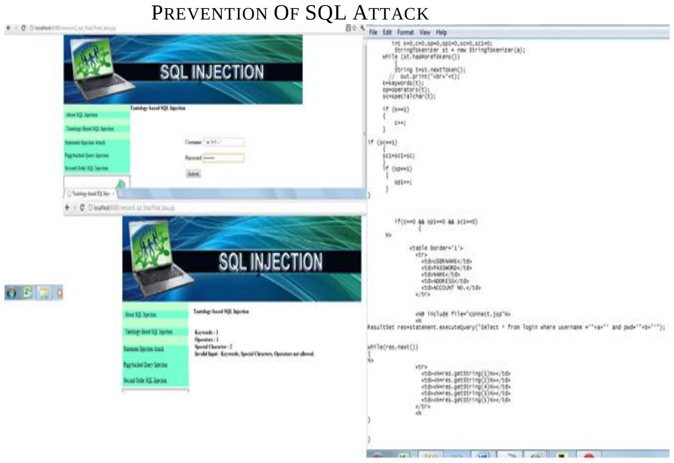
Timing Attacks: They examine the timing delays in responses from the database to fetch information from the application (e.g. WAITFOR keyword is used to execute a SQL statement after some specified delay). The attacker asks questions through the queries and sets time delay for a condition in the query. If the condition is true, the delay takes place, which allows the attacker to gain information about the application.
Example -
Original Query- 'Select * from users where uid='c12' and pwd='xxx''
Injected Query- 'Select * from users where uid=' c12' and ascii(substring(pwd,1,1)) > Z waitfor delay '0:0:7' --'and pwd='not required''
Result- If the ascii value of the first character of pwd is greater than the value z, the query will generate a delay for seven seconds.

VI. Proposed Solution

The gaining popularity of SQL Injection attacks is attributed to the fact that many Web Applications do not perform any validations on the data input by the user. This enhances the chances of the attacker to gain unauthorized access to the database of the application.

Our proposed solution to prevent SQL Injection Attacks suggests performing validations on the data entered by the user through the web form. We create three tables at the database, one which contains the list of all the SQL operators, second which contains the list of all the SQL keywords and third which contains special characters (like ;, ',',--, etc.). The input given by the user through the web form is tested against the data present in these tables. If the data contains SQL operators, SQL Keywords or special characters, the SQL query is terminated and is not allowed to be executed on the database.

FIGURE II
PREVENTION OF SQL ATTACK



SQL INJECTION
Tautology based SQL Injection
About SQL Injection
Tautology Based SQL Injection
Statement Injection Attack
Piggybacked Query Injection
Second Order SQL Injection
Username
Password
Submit
File Edit Format View Help

Keywords : 1
Operators : 1
Special Character : 2
Invalid Input : Keywords, Special Characters, Operators not allowed.

```
        int k=0,c=0,op=0,op1=0,sc=0,sc1=0;
         StringTokenizer st = new StringTokenizer(a);
     while (st.hasMoreTokens())
         {
         String t=st.nextToken();
       //  out.print("<br>"+t);
     k=keywords(t);
     op=operators(t);
     sc=specialchar(t);

     if (k==1)
     {
         c++;
     }
if (sc==1)
     {
     sc1=sc1+sc;
     }
     if (op==1)
      {
         op1++;
      }
}

         if(c==0 && op1==0 && sc1==0)
             {
    %>

           <table border='1'>
             <tr>
               <td>USERNAME</td>
               <td>PASSWORD</td>
               <td>NAME</td>
               <td>ADDRESS</td>
               <td>ACCOUNT NO.</td>
             </tr>

             <%@ include file="connect.jsp"%>
             <%
ResultSet res=statement.executeQuery("Select * from login where username ='"+a+"' and pwd='"+b+"'");

while(res.next())
{
%>
             <tr>
               <td><%=res.getString(1)%></td>
               <td><%=res.getString(2)%></td>
               <td><%=res.getString(4)%></td>
               <td><%=res.getString(3)%></td>
               <td><%=res.getString(5)%></td>
             </tr>
             <%
}

}
```

We've implemented this prevention approach in our Apache Tomcat and Oracle tool. The results are as shown in the FigureII below.
Here, the input containing a tautology is supplied as input. The application prevents the resultant query from executing it on the database (as the data input by the user contains SQL code) and consequently the number of keywords, operators and special characters present in the user input are returned.

The performance our solution depending upon its ability to counter various attack methods is given below in tableII.

TABLE II
PERFORMANCE OF OUR SOLUTION

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

VII. Conclusion and Future Work

SQL injection Attacks are a serious hazard to the growing popularity of these applications. The main target of these attacks is the database of the Web application and attackers have devised various methods for the same. We have surveyed all the common attack methods and have provided simple illustrations for each of them. Also, we have formulated a new solution to counter the problem of SQL Injection Attacks but, it is not fool proof against every well-known attack method. In future we would like to improvise our solution so that it can counter all types of attacks.

References

[1] Evaluation of SQL Injection Detection and Prevention Techniques By Atefeh Tajpour , Centre for Advanced Software Engineering (CASE) ,University Technology Malaysia ,Kuala Lumpur, Malaysia ; Mohammad JorJor zade Shooshtari ,Centre for Advanced Software Engineering (CASE) ,University Technology Malaysia,Kuala Lumpur, Malaysia

[2] A Survey On Sql Injection: Vulnerabilities, Attacks, And Prevention Techniques By Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan Department of Computer Science, International Islamic University Malaysia, Malaysia

[3] AMNESIA: Analysis and Monitoring for NEutralizing SQLInjection Attacks by William G.J. Halfond and Alessandro Orso ,College of Computing Georgia Institute of Technology

[4] SQL Injection Detection and Prevention Tools Assessment By Atefeh Tajpour CASE Center University Technology Malaysia Kuala ,Lumpur, Malaysia ; Mohammad Zaman Heydari ,IT & Management Dep UCSI University Kuala Lumpur, Malaysia ; Maslin

Masrom ,CASE Center University Technology Malaysia Kuala ,Lumpur, Malaysia , Suhaimi Ibrahim ,CASE Center UTM University Kuala Lumpur, Malaysia

[5] Shielding Against SQL Injection Attacks Using ADMIRE Model By Prof (Dr.) Sushila Madan Supriya Madan ,Department of Computer Science ,Lady Shri Ram College ,University of Delhi ; Supriya Madan ,Department of Information Technology,Vivekananda Institute of Professional Studies

[6] A Survey of SQL Injection Defense Mechanisms By Kasra Amirtahmasebi, Seyed Reza Jalalinia and Saghar Khadem, Chalmers University of Technology, Sweden

[7] An Authentication Mechanism against SQL Injection on Web Platform process. By Kamal Kumar1 Sandeep Jain2 ,1Assistant Professor, M.M.E.C., M. M. University, Mullana (Ambala, India) kumarkamal78@yahoo.com ; 2M. Tech Student , M. M. University, Mullana (Ambala, India), 26sand@gmail.com

[8] CANDID : Preventing SQL Injection Attacks Using Dynamic Candidate Evaluations By PRITHVI BISHT ,University of Illinois, Chicago ;P. MADHUSUDAN,University of Illinois, Urbana-Champaign and V. N. VENKATAKRISHNAN, University of Illinois, Chicago

[9] A Classification of SQL Injection Attacks and Countermeasures By William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, College of Computing, Georgia Institute of Technology {whalfond|jeremyv|orso}@cc.gatech.edu

[10] Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks By William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios College of Computing – Georgia Institute of Technology {whalfond|orso | manolios}@cc.gatech.edu

[11] SQLrand: Preventing SQL Injection Attacks By Stephen W. Boyd and Angelos D. Keromytis, Department of Computer Science, Columbia University,{ fswb48,angelosg}@cs.columbia.edu

[12] SQL DOM: Compile Time Checking of Dynamic SQL Statements By Russell A. McClure and Ingolf H. Krüger, University of California, San Diego, Department of Computer Science and Engineering 9500 Gilman Drive, La Jolla, CA 92093-0114,USA {rmcclure, ikrueger} @ cs.ucsd.edu

[13] A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications By Angelo CiampaUniv. Of Sannio, ItalyCorrado Aaron VisaggioUniv. Of Sannio, ItalyMassimiliano Di PentaUniv. Of Sannio, Italy

[14] SQLProb: A Proxy-based Architecture towards Preventing SQL Injection Attacks By Anyi Liu, Yi Yuan, Duminda Wijesekera and Angelos Stavrou, Department of Computer Science, George Mason University { aliu1, yyuan3,wijesekera,astavrou}@gmu.edu

[15] ADMiRe: An Algebraic Approach to System Performance Analysis Using Data Mining Techniques By Kien A. Hua, Ning Jiang, Roy Villafane and Duc Tran, University of Central Florida { kienhua, njiang, villafan, dtran}@ cs.ucf.edu