ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 14, May 2016

SQL INJECTION ATTEMPTS: DEFENSE MECHANISM IN ORACLE DATABASE

J. KARTHIKEYAN
M.E-CSE, II Year
Ranippettai Engineering College
Vellore, Tamil Nadu, India
Email Id: karthiknj6@gmail.com

ABSTRACT

SQL injection is a technique to maliciously exploit applications that use client-supplied data in SQL statements. Attackers trick the SQL engine into executing unintended commands by supplying specially crafted string input, thereby gaining unauthorized access to a database in order to view or manipulate restricted data. 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 a way to attack the data in a database through a firewall protecting it. It is a method by which the parameters of a Web-based application are modified in order to change the SQL statements that are passed to a database to return data. The aim of this paper is to create awareness among application developers or database administrators about the urgent need for database security. Our ultimate objective is to totally eradicate the whole concept of SQL injection and to avoid this technique becoming a plaything in hands of exploiters.

Keywords: **SQLIA - SQL Injection Attacks, Order of Attacks, Blind SQL Injection, Lateral SQL Injection, Stored Procedure, Bind Arguments, Dynamic SQL, DBMS_ASSERT, AUTHID CURRENT_USER**

I. INTRODUCTION

SQL is a textual relational database language. There are many varieties of SQL; however, the differences among the various dialects are minor. SQL, also known as Structured Query Language, is a special-purpose programming language used to communicate with databases. SQL can insert data, retrieve data, and update and delete data. The SQL injection technique tricks the target into passing malicious SQL code to a database by embedding portions of code with user input. The concept of the malicious code with the user input is known as 'code injection'.

A. SQL Injection Attack

Many web applications take user input from a form, often this user input is used literally in the construction of a SQL query submitted to a database. A SQL injection attack involves placing SQL statements in the user input.

SQL Injection Attacks (SQLIA) is one of the top threats for web application security, and SQL injections are one of the most serious vulnerability types. SQLIA are easy to learn and exploitable, so this method of attack is easily used by attackers. SQLIA techniques have become more common, more ambitious, easy to learn/implement, and increasingly sophisticated, so there is a need to find an effective and feasible solution for this problem in the computer security community.

Structured Query Language Injection Attack (SQLIA) is the most exposed to attack on the Internet. From this attack, the attacker can take control of the database therefore be able to interpolate the data from the database server for the website. Hence, the big challenge became to secure such website against attack via the Internet. We have presented different types of attack methods and prevention techniques of SQLIA which were used to aid the design and implementation of our model.

The first aims to put SQLIA into perspective by outlining some of the materials and researches that have already been completed. The section suggesting methods of mitigating SQLIA aims to clarify some misconceptions about SQLIA prevention and provides some useful tips to software developers and database administrators.

II. LITERATURE SURVEY

A. Impacts on SQL Injection Attacks

The impact of SQL injection attacks may vary from gathering of sensitive data to manipulating database information, and from executing system-level commands to denial of service of the application. The impact also depends on the database on the target machine and the roles and privileges the SQL statement runs with.

Researchers have proposed various solutions and techniques to address the SQL injection problems. However, there is no one solution that can guarantee complete safety. Many current solutions often cannot address all of the problems. For example, many techniques proposed are based on the assumption that only the SQL statements that receive user input are vulnerable to SQL injection attacks. However, there is a new type of attack called Lateral SQL injection, which does not require a vulnerable SQL statement to have user input parameters. A comprehensive survey can help developers and researchers better understand the various forms of SQL injection attacks, as well as the strengths and weaknesses of excising countermeasures for the attacks. This research will presents

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 14, May 2016

a survey of SQL injection attacks and various techniques used to counter them.

B. SQL Attacks Injection Are Performed

There are number of SQL injection techniques available and they differ from attacker to attacker; however, the functionality or malfunctioning they exploit is the same. They find out the vulnerability in SQL queries using the web URL or the error messages generated. Often developers use dynamic SQL statements made up of strings that are concatenated or query parameters directly specified along with input keywords.

For Example:

Select * from MyLoginAccounts where loginname='karthi' and loginID='123' and permission='admin'

When there is a SQL statement like above, there are great chances of exploit as the developer is passing the values directly into the SQL statements and which can be hacked and manipulated to give all the login details including the password and the database permission to the hacker if he /she tries to manipulate the above statement to cause a SQL injection attack. So here the attacker tries to manipulate the above SQL statement string as below:

Select * from MyLoginAccounts where loginname='**karthi' or '1'='1'--** and loginID='123' and permission='admin'

By passing one more parameter such as **"or '1'='1'"** which is always true, the user tries to capture all the records from the system. Also, to restrict the other condition to be executed from the system, attacker uses **'--'** to make the keywords following it look like a comment statement. This way, when an attacker passes a flawed string to the database query, it will return all the records to the attacker regardless of the original query. Thus the attacker can access sensitive information from the database even though he/she is not a legitimate user.

III. RELATED WORK

A. SQL Injection Strategies

1) Finding SQL Injection

There are three key aspects for finding SQL injection vulnerabilities:

1) Identifying the data entry accepted by the application,
2) Modifying the value of the entry including hazardous strings.
3) Detecting the anomalies returned by the server.

A response of the server which includes a database error or that is an HTTP error code usually eases the identification of the existence of SQL injection vulnerability. However, blind SQL injection is something that can also be exploited, even if the application doesn't return an obvious error.

2) Blind SQL Injection

Blind SQL injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response. This Inferential SQL injection attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection. Sometimes developers hide the error details which help attackers to compromise the database. In this situation attacker face to a generic page provided by developer, instead of an error message. So the SQLIA would be more difficult but not impossible. An attacker can still steal data by asking a series of True/False questions through SQL statements.

Consider two possible injections into the login field:

SELECT accounts FROM users WHERE login='KARTHI' and **1=0** -- AND pass= AND pin=0

SELECT accounts FROM users WHERE login='KARTHI' and **1=1** -- AND pass= AND pin=0

If the application is secured, both queries would be unsuccessful, because of input validation. But if there is no input validation, the attacker can try the chance. First the attacker submit the first query and receives an error message because of **"1=0"**. So the attacker does not understand the error is for input validation or for logical error in query. Then the attacker submits the second query which always true. If there is no login error message, then the attacker finds the login field vulnerable to injection. SQL injection is no more exception to the golden rule 'ALL INPUT IS EVIL'. SQL injection is nothing but, using the CRUD operation against the database in a way that it no more fulfills the desired results but give the attacker an opportunity to run his own SQL command against the database that too using the front end of your web site.

3) Steps for retrieving 'USER NAME':

1. Blind SQL injection for extracting the length of user name, if length is matched with the value (1, 2, 3...) then it will WAIT FOR 10 seconds and confirms the length of the user name.
2. Extracting the user name by matching first letter with ASCII values, it responses after 10 seconds if matched and every letter of ASCII value is matched till the length of the user name.

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 14, May 2016

Blind SQL Injection Syntax for Extracting the USER Name Length
==========
3 - Total Characters

http://[site]/page.asp?id=1; IF(LEN(USER)=1) WAITFOR DELAY '00:00:10'--
Valid page returns immediately

http://[site]/page.asp?id=1; IF(LEN(USER)=2) WAITFOR DELAY '00:00:10'--
Valid page returns immediately

http://[site]/page.asp?id=1; IF(LEN(USER)=3) WAITFOR DELAY '00:00:10'--
Valid page returns after 10 second delay

Blind SQL Injection Syntax for Extracting the USER Name
==========
d-1st Character it works

http://[site]/page.asp?id=1; IF(ASCII(lower(substring(USER),1,1)))>97 WAITFOR DELAY
Valid page returns immediately

http://[site]/page.asp?id=1; IF(ASCII(lower(substring(USER),1,1)))>98 WAITFOR DELAY
Valid page returns immediately

http://[site]/page.asp?id=1; IF(ASCII(lower(substring(USER),1,1)))>99 WAITFOR DELAY
Valid page returns immediately

http://[site]/page.asp?id=1; IF(ASCII(lower(substring(USER),1,1)))>100 WAITFOR DELA
Valid page returns after 10 second delay

Fig. 3.1 Finding User Name

4) Blind SQL Injection versus SQL Injection

In normal SQL injection hackers rely on error messages or Error-based SQL injection returned from the database in order to give them some clues on how to proceed with their SQL injection attack. But with blind SQL injection the hacker does not need to see any error messages in order to run his/her attack on the database and that is exactly why it is called blind SQL injection. So, even if the database error messages are turned off a hacker can still run a blind SQL injection attack.

B. First Order SQL Injection Attack

The attacker can simply hits the database with a malicious string attached to an input field and cause the modified code to be executed immediately. This attack type is used when the application is very insecure in terms of exposing the detailed error messages to end users, not validating input and output.

Following are some examples of first order attack:

- Existing SQL short-circuited to bring back all the data (for example, adding a query condition such as OR 1=1).
- Subquery added to an existing statement.
- UNIONS added to an existing statement to execute a second statement.

Let's assume that we have a web application with a URL as shown below which displays the product details based on the query string parameter (id) value.

1) Original URL:

http://[site]/product/view.aspx?**ID=101**

In query string (ID) value is assigned to a variable directly and not checking the type of user supplied value. Application expects value of "ID" to be an Integer, but user can supply a string also as shown below.

2) SQL Injection:

http://[site]/Product/view.aspx?**ID=101 OR 1=1**

"OR 1=1" is always true condition and returns all the rows of in a table. With that URL structure, SQL Statement generated will be look like as shown below.

3) SQL Statement Creation:

SELECT * FROM tblProduct
WHERE ProductId = 101 OR 1=1

When the above statement is executed, all the rows from tblProduct table is returned and displayed to the user. Now, the user confirms that, there is no input validation done in the server and can execute whatever he supply in the query string.

4) Tautologies: This type of attack injects SQL tokens to the conditional query statement to be evaluated always true. This type of attack used to bypass authentication control and access to data by exploiting vulnerable input field which use WHERE clause. Well, mostly text box input is the best friend of the attacker. However any input going neatly to the database is not safe.

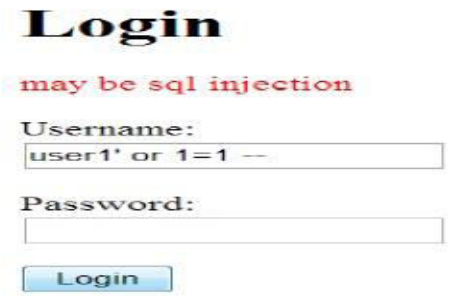

Fig. 3.2 SQL Injection in Login Form

SELECT * FROM employee
WHERE userid = **'user1' or 1 = 1 --** and password ='"

This injection is successful as it logged on to the account of a client of the company; so even this simple code can cause trouble. As the tautology statement **(1=1)** has been added to the query statement so it is always true. Attacker uses **'--'** to make the keywords following it look like a comment statement.

5) Finding Number of Columns in a table:

After locating a vulnerable site, you need to figure out how many columns are in the SQL database and how many of those columns are able to accept queries from you. Append an **"order by"** statement to the URL like this:

http://[site]/index.php?catid=1 **order by 1**

Continue to increase the number after **"order by"** until you get an error. The number of columns in the SQL database is the highest number before you receive an error.

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 14, May 2016

6) Union Query: By this technique, attackers join injected query to the safe query by the word **UNION** and then can get data about other tables from the application. Suppose for our example that the query executed from the server is the following:
SELECT Name, Phone FROM Users WHERE Id=$id

By SQL injecting the following Id value:

$id=1 **UNION ALL SELECT creditCardNumber, 1 FROM CreditCardTable**

We will have the following query with multiple statements:

SELECT Name, Phone FROM Users WHERE Id=**1 UNION ALL SELECT creditCardNumber, 1 FROM CreditCardTable**

This will join the result of the original query with all the credit card users. Union-based SQLi is an in-band SQL injection technique that leverages the UNION SQL operator to combine the results of two or more SELECT statements into a single result which is then returned as part of the HTTP response.

C. Second Order SQL Injection Attack

The attacker injects into persistent storage (such as a table row) which is deemed as a trusted source. An attack is subsequently executed by another activity. Attacker hits the database with a SQL statement to insert malicious query in a table. Later, the attacker will execute.

1) Piggy-backed Queries

Think of what the following input can cause to the application.

http://[site]**/Product/view.aspx?ID=101;DELETE FROM tblProduct**

Using the following SQL statement to select the product, based on ID.

SELECT * FROM tblProduct Where ProductId = '101'

What could happen if the seller has entered the product ID as **101'; DELETE FROM tblProduct**. Query for selecting the product will look like as shown below and will delete all the records from tblProduct table.

SELECT * FROM tblProduct Where ProductId = '101'; DELETE FROM tblProduct

In this type of attack, intruders exploit database by the query delimiter, such as **";"**, to append extra query to the original query. With a successful attack database receives and execute a multiple distinct queries. Normally the first query is legitimate query, whereas following queries could be illegitimate. So attacker can inject any SQL command to the database. In the following example, attacker inject **" 0; drop table user "** into the pin input field instead of logical value. Because of **";"** character, database accepts both queries and executes them. It is noticeable that some databases do not need special separation character in multiple distinct queries, so for detecting this type of attack, scanning for a special character is not impressive solution. I hope those examples had helped you to understand what the SQL Injection can do for an application.

D. Lateral SQL Injection Attack

The attacker can manipulate the implicit function To_Char () by changing the values of the environment variables, NLS_Date_Format or NLS_Numeric_Characters. A new type of attack that could give a hacker access to an Oracle database, called a Lateral SQL injection, could be used to gain database administrator privileges on an Oracle server in order to change or delete data or even install software.

Examples of Lateral SQL Injection Attacks

Using Lateral SQL Injection, an attacker can exploit a PL/SQL procedure that does not even take user input.

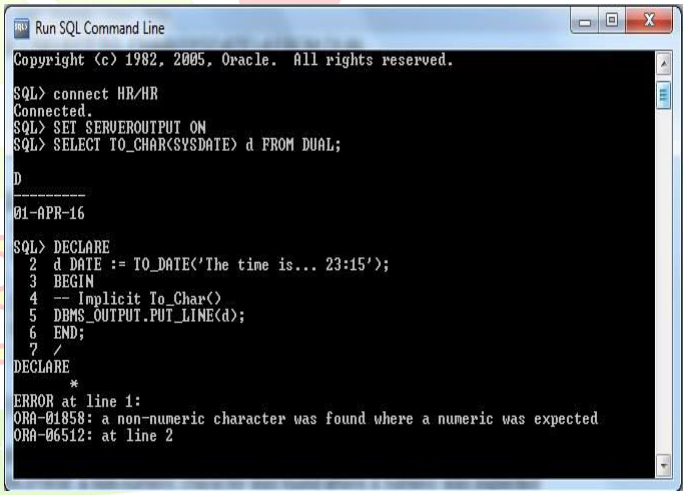

Fig. 3.3 before Lateral SQL Injection

When a variable whose data type is date or number is concatenated into the text of a SQL statement, then, contrary to popular belief, there still is a risk of injection. The implicit function TO_CHAR() can be manipulated by using NLS_Date_Format or NLS_Numeric_Characters, respectively.

You can include arbitrary text in the format model, and you do not need to include any of the "structured" elements such as Mon, hh24, and so on. Here's the "normal" use of that flexibility:

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 14, May 2016

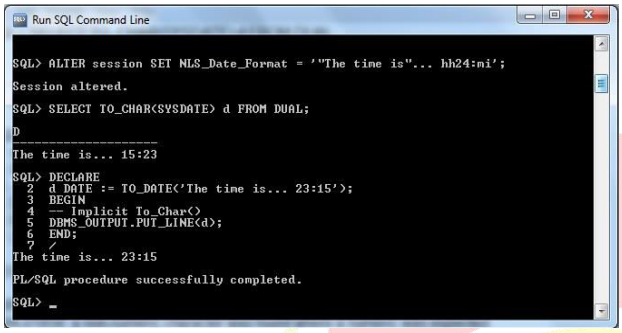

Fig. 3.4 after Lateral SQL Injection

SQL injection attacks do not have to return data directly to the user to be useful. "Blind" attacks (for example, that create a database user, but otherwise return no data) can still be very useful to an attacker. In addition, hackers are known to use timing or other performance indicators, and even error messages to deduce the success or results of an attack.

E. SQL Injection Attacks Stored Procedures

They are statements which are stored in DBs. The main problem with using these procedures is that an attacker may be able to execute them and damage database as well as the operating system and even other network components. Usually attackers know system stored procedures that come with different and almost easily can execute them.

```
CREATE OR REPLACE PROCEDURE TAB_COLS
(VCOLS VARCHAR2, VTABS VARCHAR2)
AS
   STMT VARCHAR2(4000);
   TYPE VTYPE IS VARRAY (250) OF
VARCHAR2(250);
   RESULTS VTYPE;

BEGIN
   STMT: = 'SELECT '||VCOLS ||' FROM '||VTABS;
   DBMS_OUTPUT.PUT_LINE(STMT);
   EXECUTE IMMEDIATE STMT BULK COLLECT
INTO RESULTS;
   FOR J IN 1..RESULTS.COUNT() LOOP
        DBMS_OUTPUT.PUT_LINE (RESULTS(J));
   END LOOP;
END TAB_COLS;
```

1. Executing Procedure:

EXEC **TAB_COLS ('FIRST_NAME','HR.EMPL');**

This executes to retrieve the records of Column: "FIRST_NAME" from Table: "EMPL"

2. Execution Procedure with SQL Injection:

EXEC TAB_COLS ('FIRST_NAME','HR.EMPL **WHERE 1=2 UNION SELECT TABLE_NAME FROM ALL_TABLES WHERE TABLESPACE_NAME="USERS"**');

Now **"1=2"** neglects the EMPL table, using UNION operation retrieves all TABLE_NAME's from ALL_TABLES of specific users while executing the SQL injected query instead of column FIRST_NAME.

F. Avoiding Dynamic SQL

Because SQL injection is a feature of SQL statements dynamically constructed via user inputs, it follows that designing your application to be based on static SQL reduces the chances of attack.

1) Static SQL Syntax Template

Consider the following statement:

```
Stmt constant varchar2(32767) :=
'SELECT Email FROM EMPLOYEES WHERE v_EMPID =
:B;
```

Although the above statement is not a compile-time-fixed SQL statement text, the SQL syntax template of the above statement, however, is frozen at compile time. It is clear that the SQL statement will extract Email of the employee whose email has been specified by the bind variable B. This kind of statement is a run-time static SQL statement template.

2) Dynamic SQL Syntax Template

Now consider the following statement:

```
Stmt constant varchar2(32767) :=
'SELECT ' || Col_List() || ' Report from "My
Table" where v_EMPID =: B;
```

In the above statement, the SQL syntax template is unresolved at compile time. Col_List () is invoked at run time, until then it is not clear how many columns the SQL statement will extract. This kind of statement is a dynamic SQL statement template.

G. Fixing Dynamic SQL Injection using Oracle Database Code

A stored procedure is a logical set of SQL statements, performing a specific task; it is compiled once and stored on a database server for all clients to execute; they are used very commonly for the many benefits that they provide. Often times, stored procedures are blindly considered secure; however, it is not so always. SQL Injection is a concern when dynamic SQL is handled incorrectly in a stored procedure.

In Oracle, Dynamic SQL can be used in

1. EXECUTE IMMEDIATE statements
2. DBMS_SQL package and
3. Cursors.

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 14, May 2016

1) Execute Immediate Statement
a) Secure Usage
(Execute Immediate - named parameter)
1 Stmt: = 'SELECT emp_id FROM employees WHERE emp_email =: email';
2 EXECUTE IMMEDIATE Stmt USING email;

(Execute Immediate - positional parameter)
1 Stmt: = 'SELECT emp_id FROM employees WHERE emp_email =:1 and emp_name =:2';
2 EXECUTE IMMEDIATE Stmt USING email, name;

Here, bind variables are used to set data in the query, hence SQL injection proof.

b) Vulnerable Usage
1 Stmt: = 'SELECT emp_id FROM employees WHERE emp_email = '" || email || "";
2 EXECUTE IMMEDIATE Stmt INTO empId;

Here, the input variable "email" is used directly in the query using concatenation; opening up the possibility to manipulate the "where" clause.

2) DBMS_SQL package
a) Secure Usage
1 Stmt: = 'SELECT emp_id FROM employees WHERE emp_email =: email';
2 empcur: = DBMS_SQL.OPEN_CURSOR;
3 DBMS_SQL.PARSE (empcur, Stmt, DBMS_SQL.NATIVE);
4 DBMS_SQL.BIND_VARIABLE (empcur, ':email', email);
5 DBMS_SQL.EXECUTE (empcur);

Here, bind variable is used to set data to query, hence SQL injection proof.

b) Vulnerable Usage
1 Stmt: = 'SELECT emp_id FROM employees WHERE emp_email = '" || email || "";
2 empcur: = DBMS_SQL.OPEN_CURSOR;
3 DBMS_SQL.PARSE (empcur, Stmt, DBMS_SQL.NATIVE);
4 DBMS_SQL.EXECUTE (empcur);

Here, the input variable "email" is used directly in the query using concatenation; opening up the possibility to manipulate the "where" clause.

3) Cursor with dynamic query
a) Secure Usage
1 Stmt: = 'SELECT emp_id FROM employees WHERE emp_email =: email';
2 OPEN empcur FOR Stmt USING email;

Here, bind variable is used to set data to the query, hence SQL injection proof.

b) Vulnerable Usage
1 Stmt: = 'SELECT emp_id FROM employees WHERE emp_email = '" || email || "";
2 OPEN empcur FOR Stmt;
Here, the input variable "email" is used directly in the query using concatenation; opening up the possibility to manipulate the "where" clause.

Dynamic SQL may be unavoidable in the following types of situations:
•You do not know the full text of the SQL statements that must be executed in a PL/SQL procedure. For example, a SELECT statement that includes an identifier (such as table name) that is unknown at compile time or a WHERE clause in which the number of sub clauses is unknown at compile time.
• You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
• You want to write a program that can handle changes in data definitions without the need to recompile.

If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments. If you cannot avoid input concatenation, you must validate input values, and also consider constraining user input to a predefined list of values, preferably numeric values.

H. Avoidance Strategies against SQL Injection Attacks

1) Using Bind Arguments

If you must use dynamic SQL, using bind arguments affords the next best protection against SQL injection attacks. Using bind arguments also enables cursor sharing, and thus improves application performance.

2) Use Bind Arguments with Dynamic SQL

You can use bind arguments in the WHERE clause, the VALUES clause, or the SET clause of any SQL statement, as long as the bind arguments are not used as Oracle identifiers (such as column names or table names), or key words. For example, you can rewrite this dynamic SQL with concatenated string value:

```
v_stmt:=
 'SELECT '||filter (p_column_list) ||' FROM
employees '||
 'WHERE department_name = '''||
p_department_name ||'''';

EXECUTE IMMEDIATE v_stmt;
```

into this dynamic SQL with a placeholder **(:1)** using a bind argument **(p_department_name)**:

```
v_stmt:=
 'SELECT '||filter (p_column_list) ||' FROM
employees '||
 'WHERE department_name =: 1';
```

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 14, May 2016

```
EXECUTE IMMEDIATE v_stmt
USING p_department_name;
```

So developers often use dynamic SQL to handle varying number of IN - list values or LIKE comparison operators in the query condition.

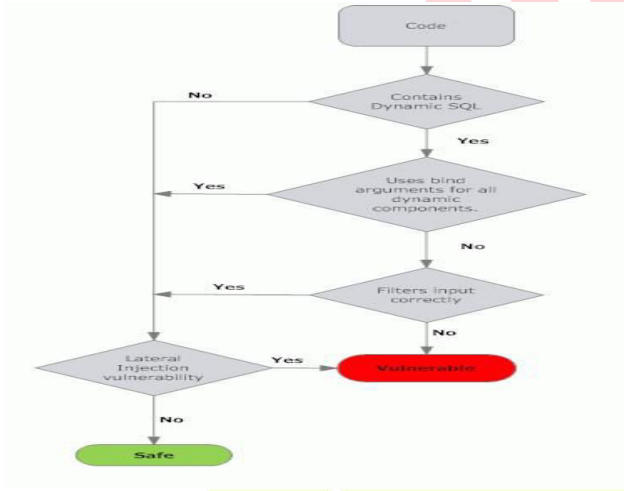

Fig. 3.5 Strategic Flowchart SQL Injection Attacks

3) Use Bind Arguments with Dynamic PL/SQL

As with dynamic SQL, you should avoid constructing dynamic PL/SQL with string concatenation. The impact of SQL injection vulnerabilities in dynamic PL/SQL is even more serious than in dynamic SQL because with dynamic PL/SQL, multiple statements (such DELETE or DROP) can be batched together and injected. If you must use dynamic PL/SQL, try to use bind arguments. For example, you can rewrite this dynamic PL/SQL with concatenated string values:

```
v_stmt:=
  'BEGIN
     get_phone (''' || p_fname ||''','''||
p_lname ||''');
   END;';

EXECUTE IMMEDIATE v_stmt;
```

into this dynamic PL/SQL with placeholders (:1,:2) using bind arguments (p_fname, p_lname):

```
v_stmt:=
  'BEGIN get_phone (:1,:2 );
END;';

EXECUTE IMMEDIATE v_stmt USING p_fname,
p_lname;
```

4) Limitations of Bind Arguments

Although you should strive to use bind arguments with all dynamic SQL and PL/SQL statements, there are instances where bind arguments cannot be used:

• DDL statements (such as CREATE, DROP, and ALTER)

• Oracle identifiers (such as names of columns, tables, schemas, database links, packages, procedures, and functions)

If bind arguments cannot be used with the dynamic SQL or PL/SQL, you must filter and sanitize all input concatenated to the dynamic statement.

I. DBMS_ASSERT Package

1) Filtering Input with DBMS_ASSERT

To guard against SQL injection in applications that do not use bind arguments with dynamic SQL, you must filter and sanitize concatenated strings. The primary use case for dynamic SQL with string concatenation is when an Oracle identifier (such as table name) is unknown at code compilation time.

The Oracle-supplied DBMS_ASSERT package contains a number of functions that can be used to filter and sanitize input strings, particularly those that are meant to be used as Oracle identifiers and help in guarding against SQL injection in applications that use dynamic SQL built with concatenated input values. Rather than relying on public synonyms, always specify the SYS schema when you call DBMS_ASSERT. In case your filtering requirements cannot be satisfied by the DBMS_ASSERT package, you may need to create your own filter.

```
CREATE OR REPLACE PROCEDURE TABS_COLS
(VCOLS VARCHAR2, VTABS VARCHAR2) AS

STMT VARCHAR2(400);
TYPE VTYPES IS VARRAY(250) OF VARCHAR2(250);
RESULTS  VTYPES;

BEGIN
   STMT: = 'SELECT'||
DBMS_ASSERT.simple_sql_name(VCOLS) ||
            '' FROM
'||DBMS_ASSERT.simple_sql_name(VTABS);
   DBMS_OUTPUT.PUT_LINE(STMT);

EXECUTE IMMEDIATE STMT BULK COLLECT INTO
RESULTS;
   FOR J IN 1..RESULTS.COUNT() LOOP
       DBMS_OUTPUT.PUT_LINE(RESULTS(J));
   END LOOP;
END TABS_COLS;
```

1) EXEC TAB_COLS ('FIRST_NAME','HR.EMPL');
Here 'FIRST_NAME' retrieved successfully

2) EXEC TAB_COLS ('FIRST_NAME','HR.EMPL
WHERE 1=2 UNION SELECT
DEPARTMENT_NAME FROM
HR.DEPARTMENTS');

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Special Issue 14, May 2016

Here trying to retrieve DEPARTMENT_NAME by SQL injection using UNION operator from DEPARTMENTS table, but **DBMS_ASSERT.simple_sql_name**

raised below error and stops SQL injection.

```
*
ERROR at line 1:
ORA-44003: invalid SQL name
ORA-06512: at "SYS.DBMS_ASSERT", line 146
ORA-06512: at "HR.TABS_COLSE", line 6
ORA-06512: at line 1
```

J) AUTHID CURRENT_USER

The **AUTHID CURRENT_USER** is used when you want a piece of code (PL/SQL) to execute with the privileges of the current user, and NOT the user ID that created the procedure. This is termed an "invoker rights", the opposite of "definer rights". The **AUTHID CURRENT_USER** is the opposite of **AUTHID DEFINER**.

In the same sense, the AUTHID CURRENT_USER is the reverse of the "grant execute" where the current user does not matter, the privileges of the creating user are used. PL/SQL, by default, runs with the privileges of the schema within which they are created no matter who invokes the procedure. In order for a PL/SQL package to run with invokers rights AUTHID CURRENT_USER has to be explicitly written into the package. The AUTHID CURRENT_USER clause tells the kernel that any methods that may be used in the type specification (in the above example, none) should execute with the privilege of the executing user, not the owner.

1) Code sample using AUTHID CURRENT_USER to execute code with invoker's rights:

1.1 Procedure that uses definer's rights

```
CREATE OR REPLACE PROCEDURE change_password
(p_username VARCHAR2 DEFAULT NULL,
p_new_password VARCHAR2 DEFAULT NULL)
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
   v_sql_stmt:= 'ALTER USER '||p_username ||'
IDENTIFIED BY'|| p_new_password;

EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
```

1.2 Procedure that uses invoker's rights

```
CREATE OR REPLACE
PROCEDURE change_password (p_username
VARCHAR2 DEFAULT NULL,
 p_new_password VARCHAR2 DEFAULT NULL)
AUTHID CURRENT_USER
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
   v_sql_stmt:= 'ALTER USER '||p_username ||'
IDENTIFIED BY'|| p_new_password;

   EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
/
```

K. Mitigation Strategies

1. Use bind arguments: Use SQL command parameters instead of directly passing the text value to input fields. This would eliminate attacks and will help in improving performance.

2. Avoid dynamic SQL with concatenated input: Try to avoid concatenated input as this attracts attackers and thus attacks.

3. Filter and sanitize input: Create query filters to only pass values which are intended ones, and filter out those which may cause or attract attacks. For example, the DBMS_ASSERT package contains a number of functions that can be used to sanitize user input and help in guarding against SQL injection in applications that use dynamic SQL built with concatenated input values.

4. Reduce the attack surface: Carry out a thorough analysis of the privileges granted to users versus the requirements. If found in excess, revoke those permissions and allow only intended ones.

IV. TESTING SQL INJECTION

A. Tools for Automatically Finding SQL Injection

1) SQLMAP

SQLMAP is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine, many niche features for the ultimate penetration tester and a broad range of switches lasting from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the operating system via out-of-band connections.

2) SQLNINJA

SQLNINJA is a tool targeted to exploit SQL Injection vulnerabilities on a web application that uses Microsoft SQL Server as its back-end. Its main goal is to provide a remote access on the vulnerable DB server, even in a very hostile environment. It should be used by penetration testers to help and automate the process of taking over a DB Server when SQL Injection vulnerability has been discovered.

V. CONCLUSION

In this paper, we have presented a study and comparison of different current techniques for detecting

and preventing SQLIAs. To perform this evaluation, we first identified the various types of SQLIAs known to date. We then evaluated the considered techniques in terms of their ability to detect and/or prevent such attacks.

Nowadays, many businesses and organizations use web applications to provide services to users. Web applications depend on the back-end database to supply with correct data. However, data stored in databases are often targets of attackers. SQL injection is a predominant technique that attackers use to compromise databases. I have conducted a survey of different types of SQL injection attacks, and have built applications and Oracle database environment to illustrate how they work. There are many types and forms of basic SQL injection attacks. The combination of them could come up with attacks that are more complicated. I mentioned some automatic testing tools for detecting SQL injection which may help DBAs. 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. Researchers have provided this two phase security because if security is compromised at one phase, the second phase can still provide security from attacks. Future work should focus on optimized and evaluating the techniques correctness and usefulness in practice.

REFERENCES

[1] SQL Injection Knowhow, Arpit Dubey, 6 June 2011 http://www.codeproject.com/Articles/206814/SQL-Injection-Knowhow

[2] D. Litchfield, "Lateral SQL Injection: A New Class of Vulnerability in Oracle," NGS Software Ltd., United Kingdom, Feb. 2008.

[3] Tutorial on Defending Against SQL Injection Attacks http://download.oracle.com/oll/tutorials/SQLInjection/index.htm

[4] Securing PL/SQL Applications with DBMS_ASSERT http://www.ngssoftware.com/papers/DBMS_ASSERT.pdf

[5] SQL Injection Attacks by Example http://www.unixwiz.net/techtips/sql-injection.html