Unwrapping Oracle PLSQL

Author: **G S Chapman**

Date: **October 2009**

Version: **1.0**

Location of Document:

DOCUMENT HISTORY

Version	Date	Changed By:	Remarks
1.0	10/10/09	G S Chapman	Initial Version
1.1	19/03/10	G S Chapman	Add some additional checks in unwrap package body.

DOCUMENT DISTRIBUTION

Copy No	Name	Role	Organisation

DOCUMENT REFERENCES

Document Name	Originator	Part Number	Version	Date
http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Finnigan.pdf	Pete Finnegan	N/A	N/A	2006
http://technology.amis.nl/blog/4753/unwrapping-10g-wrapped-plsql	Anton Scheffer	N/A	N/A	
The Oracle Hacker's Handbook	David Litchfield	N/A	N/A	
http://www.gzip.org/zlib/rfc-zlib.html	-	N/A	N/A	

TABLE OF CONTENTS

TABLE OF FIGURES

TABLES

Appendices

PURPOSE OF DOCUMENT

A description of how to unwrap Oracle 'wrapped' database code.

1 Introduction

1.1 History

This document started out as an investigation into how secure the database wrapped code is in practise as a means of securing account passwords for remote web service connection or accessing a Microsoft Active Directory LDAP service. Since the database in use was an Oracle 10g database it was decided to concentrate upon this and later database and not investigate Oracle 9 or earlier. The code developed has also been successfully tested on an Oracle 11g database.

Investigation on the web reveals that many people have tried to unwrap wrapped PL/SQL. Most people were unsuccessful but a presentation by Pete Finnegan's presentation at the 2006 Black Hat conference indicated that is possible. Additionally David Litchfield, in his book "The Oracle Hacker's Handbook", described a method to unwrap code on a 10G database. This described how the code is base64 decoded, and then, each byte is re-substituted with a second corresponding substitution table. Finally the text is decompressed, leaving the clear text of the PL/SQL.

The key to the mechanism is the substitution table, and this document describes how this was discovered and how a few procedures have been written to enable the unwrapping of both code stored in the database and also operating system flat files.

1.2 Wrapped code display

The first step is to look at the output of some simple wrapped procedures. The executable wrap.exe (Windows) is used to wrap the PL/SQL code in a file, alternatively the database procedure dbms_ddl.wrap can be used.

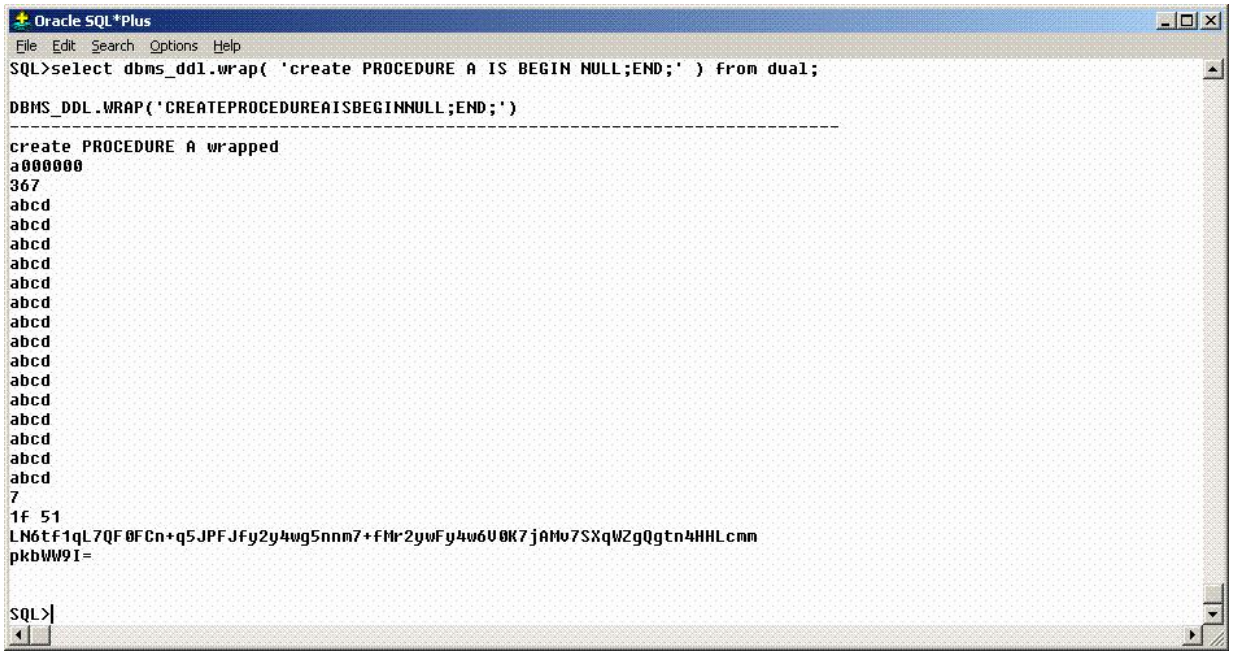

Figure 1- Unwrap display

In this output we see a line with a000000 and 15 lines with 'abcd'. This display is typical of all inspected wrapped code. The third line, in this example 367, is probably related with the database version. Upon an 10.2.0.4 database it seems to always be a value of 367, upon an 10.2.0.1 database is seems to be 2e. With an 11g release 1 database the values seems to be a 1.

The 19th line, appears to be and indication of the type of the wrapped object .

Table 1 - Wrap database type codes

Code	Database type
7	Procedure
8	Function
b	Package Body

The following line contains 2 hex numbers. The first is the length from the unwrapped text without the create + 1, and the second is the length of the base64-encoded text.

It is possible to use a shorter piece of code..

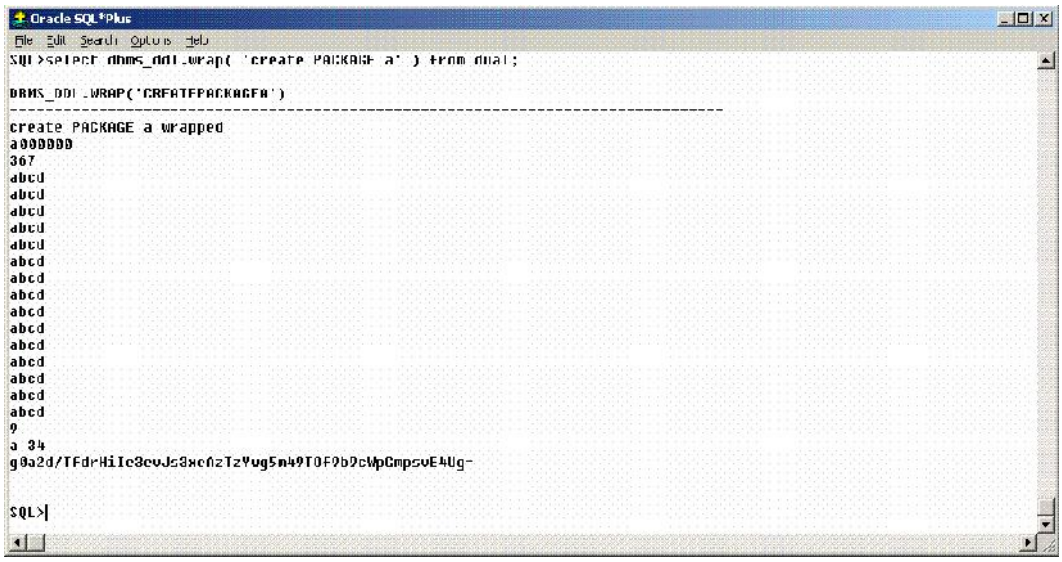

Figure 2 - Unwrap display 2

From Litchfieldss' book we can if we base64-decode this, skip the first 20 bytes (the size of a SHA1-hash) there are only 18 bytes left to decipher. For the base64-decoding one uses the package utl_encode. There is also an Oracle package utl_compress, which can be used for Lempel-Ziv compression/decompression.

There are 2 options:

- Find a substitution table, apply it, and decompress the result to get unwrapped PL/SQL code.
- Compress PL/SQL code, and compare this to the base64-decoded output to FIND (a part of) the substitution table.

Using the second option we can use different inputs to the dbms_ddl.wrap package.

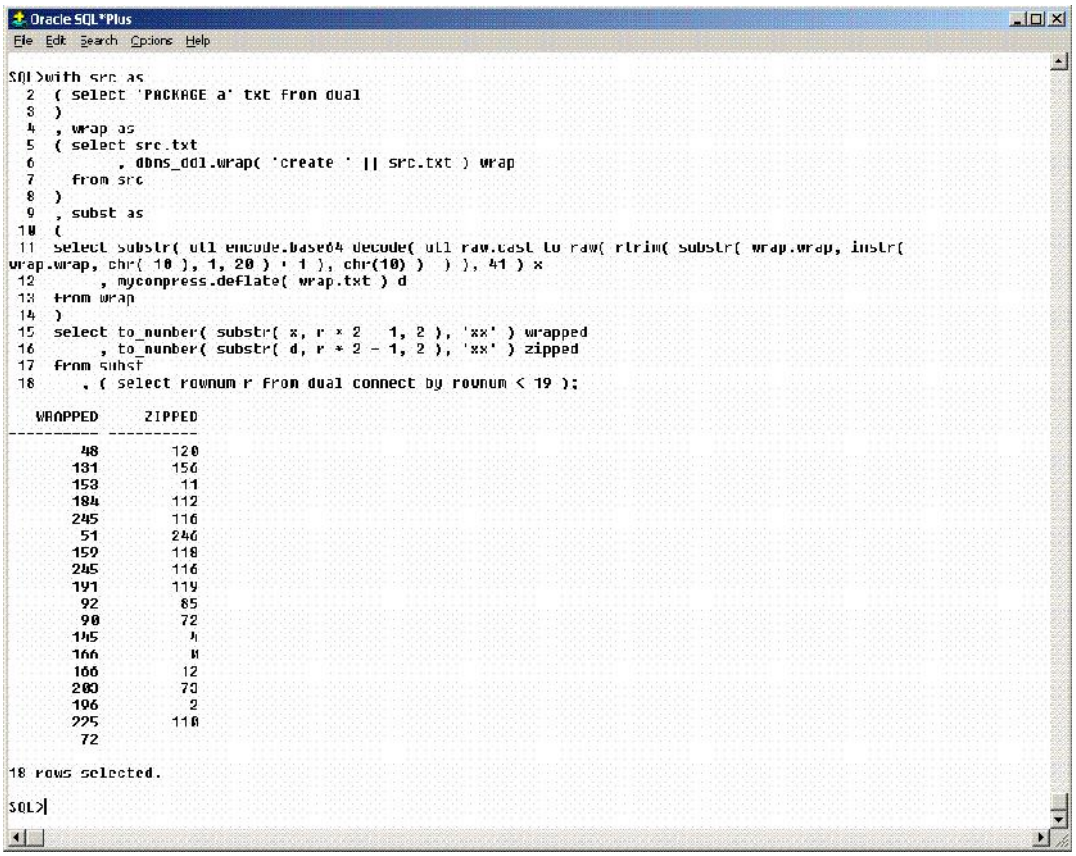

Figure 3 - Trying a line feed.

Unfortunately the output is one byte short, but remember the two hex numbers in the wrapped output. The first hex number was the length of the unwrapped code + 1. So if one adds a newline character to the input of the compression, that will give 1 more byte for the zipped code. This reveals another small problem in that the output of the wrapped column we has 2 lines with the value of 166. These lines should both have the same value in the zipped column which is no so. One alternative is to add a space character.

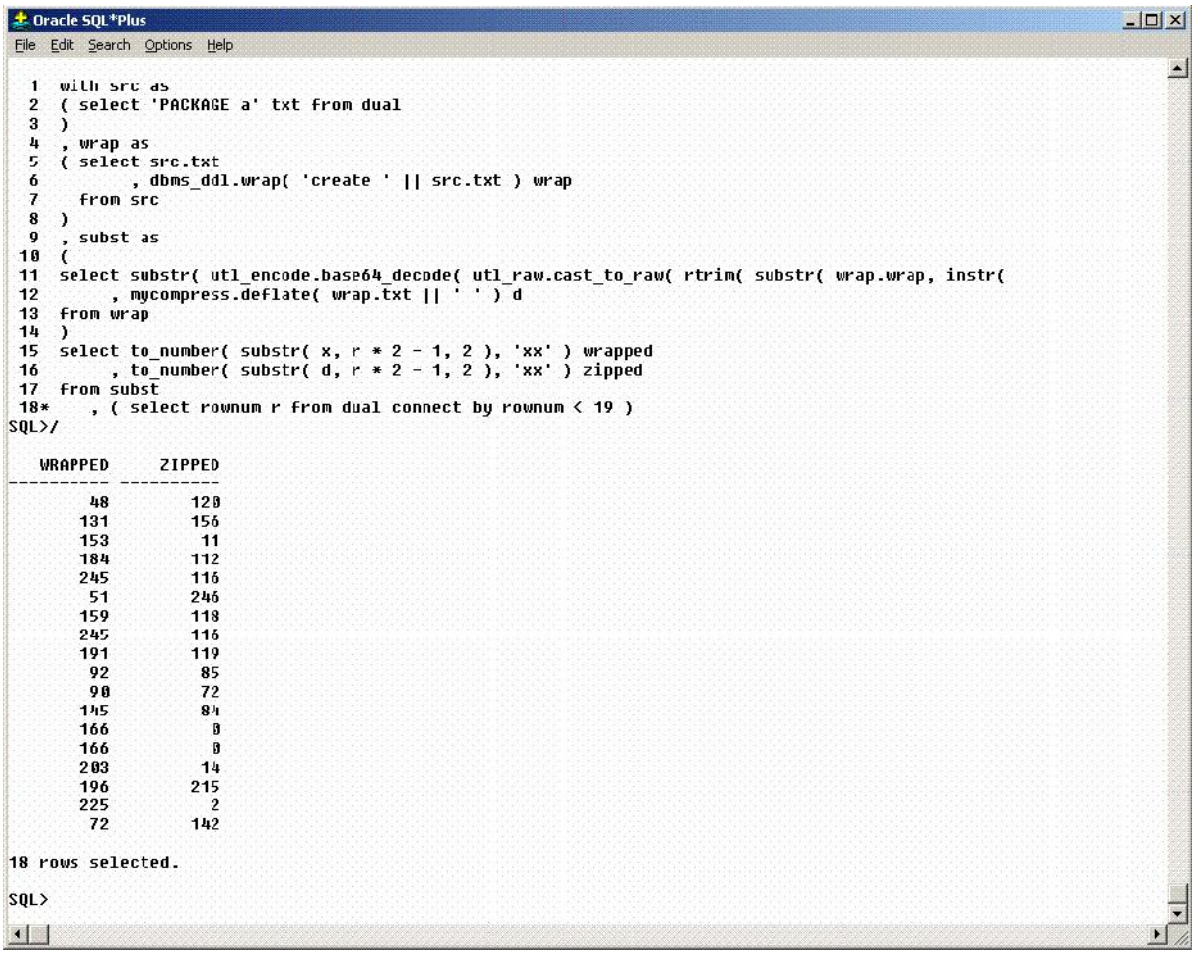

Figure 4 - Trying a space character

The above display thus provides 16 entries of the possible 256, for the substitution table. Changing the test code and retesting reveals other values but unfortunately there are different "zipped values" for the same "wrapped values". This indicates that the use of a space character is incorrect. Further testing reveals that the character to be used has to be a byte 0.

If an attempt is made to create the additional values in a PLSQL loop more errors are discovered, this time related with the second byte. This is where the compression level is stored, and changing it to a value of 9 enables the whole substitution table to be generated.

The above mechanism can be used it in a SQL-statement or in a Java-program to unwrap the plb-files of any lost sources.

2 Code

2.1 Java package code

The following java source package is called by a database PLSQL package supplied below. It uses a java supplied compression algorithm.

```
DROP JAVA SOURCE MY_COMPRESS;

CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED MY_COMPRESS as import java.io.*;
import java.util.zip.*;

public class MY_COMPRESS
{
  public static String Inflate( byte[] src )
  {
    try
    {
      ByteArrayInputStream bis = new ByteArrayInputStream( src );
      InflaterInputStream iis = new InflaterInputStream( bis );
      StringBuffer sb = new StringBuffer();
      for( int c = iis.read(); c != -1; c = iis.read() )
      {
        sb.append( (char) c );
      }
      return sb.toString();
    } catch ( Exception e )
    {
    }
    return null;
  }
  public static byte[] Deflate( String src, int quality )
  {
    try
    {
      byte[] tmp = new byte[ src.length() + 100 ];
      Deflater defl = new Deflater( quality );
      defl.setInput( src.getBytes( "UTF-8" ) );
      defl.finish();
      int cnt = defl.deflate( tmp );
      byte[] res = new byte[ cnt ];
      for( int i = 0; i < cnt; i++ )
        res[i] = tmp[i];
      return res;
    } catch ( Exception e )
    {
    }
    return null;
  }
}
/
```

2.2 PLSQL wrapper for java code

The following is a small wrapper around the Java code supplied above to enable it to be called easily from a PLSQL (or SQL) procedure.

```
CREATE OR REPLACE package mycompress
is
  function deflate( src in varchar2 )
  return raw;
--
  function deflate( src in varchar2, quality in number )
  return raw;
--
  function inflate( src in raw )
  return varchar2;
--
end;
/
```

```
CREATE OR REPLACE package body mycompress
is
  function deflate( src in varchar2 )
  return raw
  is
  begin
    return deflate( src, 6 );
  end;
--
  function deflate( src in varchar2, quality in number )
  return raw
  as language java
  name 'MY_COMPRESS.Deflate( java.lang.String, int ) return byte[]';
--
  function inflate( src in raw )
  return varchar2
  as language java
  name 'MY_COMPRESS.Inflate( byte[] ) return java.lang.String';
--
end;
/
```

2.3 Unwrap package

The following package, which contains the translation table provides a number of mechanisms to call the java code to unwrap the source whether it is supplied as a text string, a database procedure (or package) or as an operating system file. In the case of the latter the usual Oracle requirements for defined directories, etc apply.

Note that the procedure to accept a named package from the dba_source view assumes that the package owner has the correct grants. Using all_source does not always find the package being requested.

```
CREATE OR REPLACE PACKAGE UNWRAP
AS
   PROCEDURE table source (p owner IN VARCHAR2,
                           p name  IN VARCHAR2,
                           p_type  IN VARCHAR2);

   PROCEDURE text_source (p_text IN VARCHAR2);

   PROCEDURE file source (p dir    IN VARCHAR2,
                          p_fname  IN VARCHAR2);

END UNWRAP;
/
CREATE OR REPLACE PACKAGE BODY UNWRAP
AS
/*
   Make into a package with the base functions included below and a few
   procedures.

   One to accept an input string as is.
   One to read a file
   One to table a schema and name to enable a read the text line from the
   all_source table.


   DBMS_OUTPUT output is probably more than suitable since there would be a need
   to add comments and modify the header in some way if only to insert the
   phrase 'CREATE OR REPLACE'.

   The output could be spooled to an output file if desired.


*/

   not_wrapped    EXCEPTION;

   PROCEDURE Print (p_text IN VARCHAR2)
   IS
```

```
BEGIN
   dbms_output.put_line(p_text);
END Print;

FUNCTION TRANS (v_inp VARCHAR2)
RETURN VARCHAR2
IS
/*

*/
BEGIN
   RETURN   UTL_RAW.TRANSLATE ( v_inp,

         '000102030405060708090A0B0C0D0E0F'
      || '101112131415161718191A1B1C1D1E1F'
      || '202122232425262728292A2B2C2D2E2F'
      || '303132333435363738393A3B3C3D3E3F'
      || '404142434445464748494A4B4C4D4E4F'
      || '505152535455565758595A5B5C5D5E5F'
      || '606162636465666768696A6B6C6D6E6F'
      || '707172737475767778797A7B7C7D7E7F'
      || '808182838485868788898A8B8C8D8E8F'
      || '909192939495969798999A9B9C9D9E9F'
      || 'A0A1A2A3A4A5A6A7A8A9AAABACADAEAF'
      || 'B0B1B2B3B4B5B6B7B8B9BABBBCBDBEBF'
      || 'C0C1C2C3C4C5C6C7C8C9CACBCCCDCECF'
      || 'D0D1D2D3D4D5D6D7D8D9DADBDCDDDEDF'
      || 'E0E1E2E3E4E5E6E7E8E9EAEBECEDEEEF'
      || 'F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF',

         '3D6585B318DBE287F152AB634BB5A05F'
      || '7D687B9B24C228678ADEA4261E03EB17'
      || '6F343E7A3FD2A96A0FE935561FB14D10'
      || '78D975F6BC4104816106F9ADD6D5297E'
      || '869E79E505BA84CC6E278EB05DA8F39F'
      || 'D0A271B858DD2C38994C480755E4538C'
      || '46B62DA5AF322240DC50C3A1258B9C16'
      || '605CCFFD0C981CD4376D3C3A30E86C31'
      || '47F533DA43C8E35E1994ECE6A39514E0'
      || '9D64FA5915C52FCABB0BDFF297BF0A76'
      || 'B449445A1DF0009621807F1A82394FC1'
      || 'A7D70DD1D8FF139370EE5BEFBE09B977'
      || '72E7B254B72AC7739066200E51EDF87C'
      || '8F2EF412C62B83CDACCB3BC44EC06936'
      || '6202AE88FCAA4208A64557D39ABDE123'
      || '8D924A1189746B91FBFEC901EA1BF7CE'
   );

END;

FUNCTION subst (v_inp IN VARCHAR2)
RETURN VARCHAR2
IS
/*
*/
BEGIN
   RETURN SUBSTR (
      UTL_ENCODE.base64_decode (
         UTL_RAW.cast_to_raw (RTRIM (SUBSTR (v_inp, INSTR (v_inp,
                                                           CHR (10),
                                                           1,
                                                           20)
                                             + 1), CHR (10)))
      ),
      41
   );
END;

PROCEDURE OpenFile (dir_in  IN VARCHAR2,
                    fname   IN VARCHAR2,
                    fhandle OUT UTL_FILE.FILE_TYPE)
IS
BEGIN
      fhandle := UTL_FILE.FOPEN (dir_in, fname, 'r');
EXCEPTION
   WHEN UTL_FILE.INVALID_PATH
   THEN
```

```
         DBMS_OUTPUT.PUT_LINE ('invalid_path');
      WHEN UTL_FILE.INVALID_MODE
      THEN
         DBMS_OUTPUT.PUT_LINE ('invalid_mode');
      WHEN UTL_FILE.INVALID_FILEHANDLE
      THEN
         DBMS_OUTPUT.PUT_LINE ('invalid_filehandle');
      WHEN UTL_FILE.INVALID_OPERATION
      THEN
         DBMS_OUTPUT.PUT_LINE ('invalid_operation');
      WHEN UTL_FILE.READ_ERROR
      THEN
         DBMS_OUTPUT.PUT_LINE ('read_error');
      WHEN UTL_FILE.WRITE_ERROR
      THEN
         DBMS_OUTPUT.PUT_LINE ('write_error');
      WHEN UTL_FILE.INTERNAL_ERROR
      THEN
         DBMS_OUTPUT.PUT_LINE ('internal_error');
END OpenFile;

PROCEDURE CloseFile (fhandle_in IN UTL_FILE.FILE_TYPE)
IS
BEGIN

   UTL_FILE.FCLOSE_ALL;

EXCEPTION
   WHEN UTL_FILE.INVALID_PATH
   THEN
      DBMS_OUTPUT.PUT_LINE ('invalid_path');
   WHEN UTL_FILE.INVALID_MODE
   THEN
      DBMS_OUTPUT.PUT_LINE ('invalid_mode');
   WHEN UTL_FILE.INVALID_FILEHANDLE
   THEN
      DBMS_OUTPUT.PUT_LINE ('invalid_filehandle');
   WHEN UTL_FILE.INVALID_OPERATION
   THEN
      DBMS_OUTPUT.PUT_LINE ('invalid_operation');
   WHEN UTL_FILE.READ_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('read_error');
   WHEN UTL_FILE.WRITE_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('write_error');
   WHEN UTL_FILE.INTERNAL_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('internal_error');
END CloseFile;

PROCEDURE table_source (p_owner IN VARCHAR2,
                        p name  IN VARCHAR2,
                        p_type  IN VARCHAR2)
IS

   v_s      VARCHAR2 (32000);
   v x      VARCHAR2 (32000);
   v t      VARCHAR2 (32000);
   nlines  INTEGER;
BEGIN

  -- dbms_output.put_line('Procedure: '||p_owner||'.'||p_name||'  '||p_type);

  SELECT count(line)
  INTO nlines
  FROM   dba_source
   WHERE   name = p_name
   AND     owner = p_owner
   AND     type = p type;
   -- dbms_output.put_line('Lines found '||nlines);

   v_s := ' ';

   FOR i IN 1..nlines
   LOOP
      SELECT text
```

```
            INTO   v_t
            FROM   dba_source
            WHERE   name = p name
            AND     owner = p owner
            AND     type = p_type
            AND     line = i;

           IF i = 1 AND INSTR(SUBSTR(v_t, 1, 60), 'wrapped') = 0 THEN
               RAISE not_wrapped;
           END IF;
            v_s := v_s || v_t;
            v_t := '';
         END LOOP;


         v_x := subst (v_s);
         v_t := trans(v_x);

         DBMS_OUTPUT.put_line (mycompress.inflate (v_t));
      EXCEPTION
         WHEN NO DATA FOUND THEN
            dbms_output.put_line('Procedure: '||p_owner||'.'||p_name||'  '||p_type||' not
found.');
         WHEN not_wrapped THEN
            dbms_output.put_line('Procedure: '||p_owner||'.'||p_name||'  '||p_type||' is
not wrapped code.');
      END table_source;

      PROCEDURE file_source (p_dir    IN VARCHAR2,
                             p_fname  IN VARCHAR2)
      IS

         v_s   VARCHAR2 (32000);
         v_x   VARCHAR2 (32000);
         v_t   VARCHAR2 (32000);

         fhandle       UTL FILE.FILE TYPE;
         ufhandle      UTL FILE.FILE TYPE;
         dir_in        VARCHAR2 (100);
         fname         VARCHAR2 (40);
      BEGIN

         Openfile(p_dir, p_fname, fhandle);

         utl_file.get_line(fhandle, v_s, 32000);
           dbms_output.put_line(substr(v_s,1,100));
         CloseFile(fhandle);

         IF upper(substr(v s, 1, 6)) != 'CREATE' THEN
            v_x := subst ( 'CREATE ' || v_s );
         ELSE
            v_x := subst (v_s);
         END IF;

         v_t := trans(v_x);

         DBMS_OUTPUT.put_line (mycompress.inflate (v_t));
      END file_source;

      PROCEDURE text_source (p_text IN VARCHAR2)
      IS

--       v_s   VARCHAR2 (32000);
         v x   VARCHAR2 (32000);
         v t   VARCHAR2 (32000);
      BEGIN
         IF upper(substr(p_text, 1, 6)) != 'CREATE' THEN
            v_x := subst ( 'CREATE ' || p_text );
         ELSE
            v x := subst (p_text);
         END IF;

         v_t := trans(v_x);

         DBMS_OUTPUT.put_line (mycompress.inflate (v_t));
      END text_source;
```

```
END UNWRAP;
/
```

2.4 Test code

The following code demonstrates the usage of the two of the procedures. The example procedure used is intended as an illustration of the technique.

```
declare
   v text VARCHAR2(4000) := ' PROCEDURE osc_alert_compl wrapped
a000000
b2
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
7
4a 79
u3oikr+aJkOMlS5je0wyvsStVqMwg5nnm7+fMr2ywFyFltxKrkehYsX0rv9yYoXMuHSLwMAy
/tKGBtIon0q/UjJtv1IJUGmAXslN4t7XQ/zB4Nemprjhbwo=

';

begin
   dbms_output.put_line ('Test 1');
   unwrap.table_source ('GSC', 'OSC_ALERT_COMPL1','PROCEDURE');

   dbms_output.put_line('Test 2');
   unwrap.text_source(v_text);

end;
/
```

Output displayed:

```
Test 1
PROCEDURE osc_alert_compl1 ( TNSALIAS VARCHAR2 DEFAULT NULL) AS
C_STMT               VARCHAR2(4000)   ;
C_TIME          OWA_UTIL.VC_ARR       ;
C_MESSAGE       OWA_UTIL.VC_ARR       ;
BEGIN
          OSC_HTP('TABLEOPEN','NAME');
          HTP.TABLEROWOPEN;
          OSC_HTP('TABLEDATA_HEAD','MESSAGE');
          HTP.TABLEROWCLOSE;
          C_STMT := 'SELECT TIME,text
                    FROM
                    (SELECT last_value(to_char(TIME,'||''''||'DD.MM.YY
Hh24:MI:SS'||''''||') ignore nulls)
                    over(ORDER BY rownum ASC ROWS unbounded preceding) AS
                    TIME,text
                    FROM
                    (SELECT rownum,
                    CASE
                    WHEN(text LIKE '||''''||'             :  :   20__'||''''||') THEN
                    TO_DATE(text, '||''''||'DY MON DD HH24:MI:SS
YYYY'||''''||','||''''||'nls_date_language=american'||''''||')
                    END AS
                    TIME,
                    CASE
                    WHEN(text NOT LIKE '||''''||'___ ___ __ __:__:__ 20__'||''''||')
THEN
```

```
                    text
                    END AS
                    text
                    FROM sys.osc_alert_text@'||TNSALIAS||')
                    )
                    WHERE text IS NOT NULL';

          EXECUTE IMMEDIATE C_STMT BULK COLLECT INTO C_TIME,C_MESSAGE;
          IF C MESSAGE.COUNT >0 THEN
           FOR X IN 1..C_MESSAGE.COUNT LOOP
            HTP.TABLEROWOPEN;
            OSC_HTP('TABLEDATA_NORMAL',C_TIME(X));
            OSC_HTP('TABLEDATA_NORMAL',C_MESSAGE(X));
            HTP.TABLEROWCLOSE;
           END LOOP;
           HTP.TABLEROWOPEN;
           OSC_HTP('TABLEDATA_NORMAL','END OF LIST');
          HTP.TABLECLOSE;
         ELSE
          OSC HTP('WARNING_GRAY','No Entries in the Alert Log yet');
         END IF;
EXCEPTION
WHEN OTHERS THEN
  OSC_FILL_ERROR('osc_alert_compl1',SYSDATE,NULL,SQLERRM(SQLCODE));
END;
Test 2
PROCEDURE osc_alert_compl AS
BEGIN
       OSC_DEF_DB ('osc_alert_compl1');
END;
/
```

2.5 Script to generate substation table.

The following script can be used to generate the substitution table.

```
declare
  type tp_tab is table of pls_integer index by pls_integer;
  t2 tp_tab;
  cursor c_fill( p_in in varchar2 )
  is
with src as
( select p_in txt
  from dual
)
, wrap as
( select src.txt
       , dbms_ddl.wrap( 'create ' || src.txt ) wrap
  from src
)
, subst as
(
select substr( utl_encode.base64_decode( utl_raw.cast_to_raw(rtrim( substr( wrap.wrap,
instr( wrap.wrap, chr( 10 ), 1, 20 ) + 1 ), chr(10) )  ) ), 41 ) x
     , mycompress.deflate( wrap.txt || chr(0), 9 ) d
from wrap
)
select to_number( substr( x, r * 2 - 1, 2 ), 'xx' ) xr
     , to_number( substr( d, r * 2 - 1, 2 ), 'xx' ) dr
from subst
   , ( select rownum r from dual connect by rownum <= ( select length( x ) / 2 from
subst ) );

  t varchar2(512);
  cnt number;
  procedure fill( p_txt in varchar2, p_from in number, p_to in number, p_extra in
varchar2 := null )
  is
  begin
    for i in p_from .. p_to
    loop
      for r_fill in c_fill( p_txt || chr( i ) || p_extra )
      loop
```

```
        if (  t2( r_fill.xr ) != -1
           and t2( r_fill.xr ) != r_fill.dr
           )
        then
          dbms_output.put_line( 'error: value maps to two different values ' || p_txt
);
          dbms_output.put_line( chr( i ) || ' ' || r_fill.xr || ' ' || t2( r_fill.xr )
|| ' ' || r_fill.dr );
          raise no_data_found;
        end if;
        t2( r_fill.xr ) := r_fill.dr;
      end loop;
    end loop;
  end;
  procedure fill2( p_txt in varchar2 )
  is
  begin
    for i in 0 .. 99
    loop
      fill( p_txt, ascii( 'a' ), ascii( 'z' ), to_char( i, 'fm999' ) );
      fill( p_txt, ascii( 'A' ), ascii( 'Z' ), to_char( i, 'fm999' ) );
    end loop;
  end;
--
begin
  for i in 0 .. 255
  loop
    t2( i ) := -1;
  end loop;
--
  dbms_output.put_line( to_char( sysdate, 'hh24:mi:ss' ) );
  fill2( 'PACKAGE ' );
--  fill2( 'PACKAGE BODY ' );
--  fill2( 'FUNCTION ' );
--  fill2( 'PROCEDURE ' );
--  fill2( 'TYPE BODY ' );
--
  dbms_output.put_line( to_char( sysdate, 'hh24:mi:ss' ) );
  cnt := 0;
  for i in 0 .. 255
  loop
    if t2( i ) != -1
    then
      dbms_output.put_line(cnt||': '||t2(i)|| ' : '||to_char( t2(i), 'xxxx' ));
      cnt := cnt + 1;
    end if;
  end loop;
  dbms_output.put_line( 'cnt ' || cnt );
end;
/
```

A. Notes

A.1 Oracle 9i wrap mechanism

The 10g wrap mechanism seems a lot weaker than the 9i and lower mechanism. The main problem with the 9i mechanism is that the symbol table is visible, with 10g to 11g that is not so BUT is weaker as full reversal is possible. With 9i it is simply the internal state of the PL/SQL compiler, i.e. DIANA written out to disk as IDL. The unwrap process for 9i is a feature of the design of DIANA which was intended for low memory older machines where code would be stored in an intermediate format and it should be possible to reconstruct the source code. Writing an un-wrapper for 9i and lower is a bigger task than with Oracle 10g. With Oracle 10g there is the addition of a hidden symbol table but a much weaker mechanism to hide the code.

A.1.1 Limits on using SQL to unwrap code

In SQL-queries the maximum allowed length of RAW and VARCHAR2 is 4000, which means that sql-queries can not be used for unwrapping “large” pieces of code. For those larger unwrapping tasks one has to use PLSQL, which has a limit of more than 32000, or java. For cutting of the first 40 chars, in the example the RAW value is converted to a VARCHAR2 value, and in that conversion every RAW becomes the 2 byte hexadecimal representation of the value.

A.1.2 Unwrapping Oracle packages

The unwrapping of Oracle supplied packages and procedures is not recommended or encouraged and problems may be encountered. Some are known to unwrap successfully whilst others may encounter problems. The cause of these problems has not been investigated.