UnionSys Technologies

Securing Stored Data Using Transparent Data Encryption And Disaster Recovery Solution

CONTENTS

Introduction

Purpose

Critical business data in database is an obvious target for attackers. Though access control has been deployed since the birth of large databases, still security of the databases was considered a big problem to be solved and there were threats to secrecy integrity of data. Every organization must protect sensitive data or suffer potential legislative, regulatory, legal and brand consequences. Database encryption solutions have been proven the best alternative solution for protection of sensitive data.

This is a specialized and complex solution area and if internal resources don’t have the cryptography expertise, an outside expertise should be used for superior performance.

Scope

To protect your company’s database assets, there are security measures you should take today. These include encrypting data as it moves across your enterprise networks and as it sits at rest, in storage on database systems. Extra steps and precautions should be taken to carefully control access this data.

This document covers the following measures that support database encryption

- Transparent Data Encryption
- DBMS_OBFUSCATION_TOOLKIT(9i).
- DBMS_CRYPTO
- Virtual Private Database Policy(10g).

About Transparent Data Encryption

Oracle Database uses authentication, authorization, and auditing mechanisms to secure data in the database, but not in the operating system data files where data is stored. To protect these data files, Oracle Database provides Transparent Data Encryption (TDE). TDE encrypts sensitive data stored in data files. To prevent unauthorized decryption, TDE stores the encryption keys in a security module external to the database.

Database users and applications do not need to manage key storage or create auxiliary tables, views, and triggers. An application that processes sensitive data can use TDE to provide strong data encryption with little or no change to the application.

Use TDE to protect confidential data, such as credit card and social security numbers, stored in table columns. You can also use TDE to encrypt entire tablespaces.

When you store both the encryption key and the encrypted data in the database, another potential security hole opens up—if the disks containing the entire database are stolen, the data becomes immediately vulnerable. One way around this problem is to encrypt all the data elements and store the keys separately on a different location away from the disk drives where the data resides.

If your database is completely isolated, you may not feel that you need to encrypt its data. However, you may still want to protect the data in case of a disk theft. One solution would be to create a view to show the decrypted value. In this case, if the key is stored elsewhere, a physical disk theft will not make the data vulnerable. This approach works, but it requires an extensive and elaborate setup.

To address these types of situations, Oracle has introduced a new feature known as Transparent Data Encryption (TDE), available starting with Oracle Database 10*g* Release 2. TDE uses a combination of two keys—one master key stored outside the database in a *wallet* and one key for each table.

Benefits of Using Transparent Data Encryption

Transparent Data Encryption (TDE) has the following advantages:

As a security administrator, you can be sure that sensitive data is safe in case the storage media or data file gets stolen.

Implementing TDE helps you address security-related regulatory compliance issues.

You do not need to create triggers or views to decrypt data for the authorized user or application. Data from tables is transparently decrypted for the database user and application.

Database users and applications need not be aware of the fact that the data they are accessing is stored in encrypted form. Data is transparently decrypted for the database users and applications.

Applications need not be modified to handle encrypted data. Data encryption and decryption is managed by the database.

Key management operations are automated. The user or application does not need to manage encryption keys

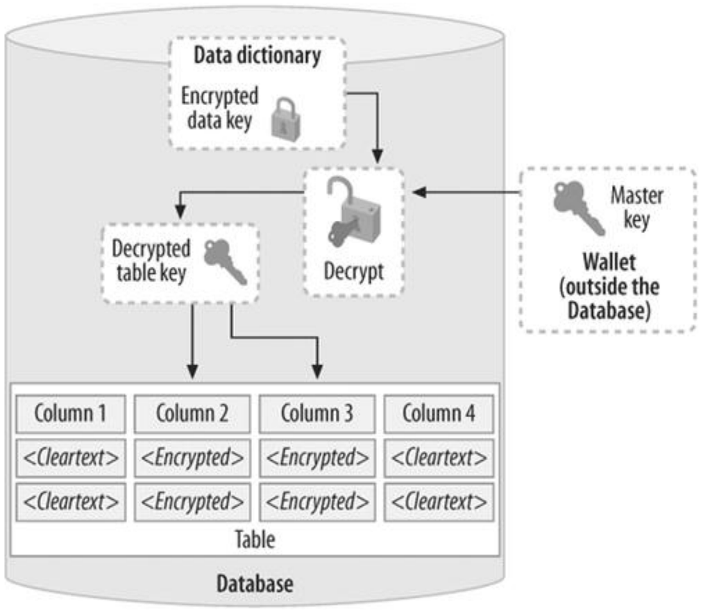
Types of Transparent Data Encryption

Transparent Data Encryption (TDE) column encryption enables you to encrypt sensitive data stored in select table columns. TDE tablespace encryption enables you to encrypt all data stored in a tablespace.

Both TDE column encryption and TDE tablespace encryption use a two-tiered, key-based architecture. Even if the encrypted data is retrieved, it cannot be understood until authorized decryption occurs, which is automatic for users authorized to access the table.

TDE Column Encryption

TDE column encryption is used to protect confidential data, such as credit card and social security numbers, stored in table columns. TDE column encryption uses the two-tiered, key-based architecture to transparently encrypt and decrypt sensitive table columns. The TDE master encryption key is stored in an external security module, which can be an Oracle wallet or Hardware Security Module (HSM). This master encryption key is used to encrypt the table key, which in turn is used to encrypt and decrypt data in the table column.



As shown in figure the master encryption key is stored in an external security module that is outside the database and accessible only to the security administrator. For this external security module, Oracle uses an Oracle wallet or Hardware Security Module (HSM), as described in this chapter. Storing the master encryption key in this way prevents its unauthorized use.

Using an external security module (wallet/HSM) separates ordinary program functions from encryption operations, making it possible to divide duties between database administrators and security administrators. Security is enhanced because the wallet password can be unknown to the database administrator, requiring the security administrator to provide the password.

When a table contains encrypted columns, a single table key is used regardless of the number of encrypted columns. The table keys for all tables are encrypted with the database server master encryption key and stored in a dictionary table in the database. No keys are stored in the clear.

TDE Tablespace Encryption

TDE tablespace encryption enables you to encrypt an entire tablespace. All objects created in the encrypted tablespace are automatically encrypted. TDE tablespace encryption is useful if you want to secure sensitive data in tables. You do not need to perform a granular analysis of each table column to determine the columns that need encryption.

Setting Up TDE

Before you start using TDE, you have to set up the wallet where the master key is stored and secure it. Here is a step-by-step approach to wallet management.

1. Set the wallet location.

When you enable TDE for the first time, you need to create the wallet where the master key is stored. By default, the wallet is created in the directory $ORACLE_BASE/admin/$ORACLE_SID/wallet. You can also choose a different directory by specifying it in the file SQLNET.ORA. For instance, if you want the wallet to be in the /oracle_wallet directory, place the lines in the SQLNET.ORA file as shown here. In this example, I assume that the default location is chosen.

```
ENCRYPTION_WALLET_LOCATION =
  (SOURCE=
      (METHOD=file)
      (METHOD_DATA=
         (DIRECTORY=/oracle_wallet)))
```

Make sure to include the wallet as a part of your regular backup process.

2. Set the wallet password.

Now you have to create the wallet and set the password to access it. This is done in one step by issuing:

```
   ALTER SYSTEM SET ENCRYPTION KEY IDENTIFIED BY
"secretpassword";
```

This command does three things:

a. It creates the wallet in the location specified in Step 1.
b. It sets the password of the wallet as "secretpassword".
c. It opens the wallet for TDE to store and retrieve keys.

The password is case-sensitive, and you must enclose it in double quotes.

3. Open the wallet.

   The previous step opens the wallet for operation. However, after the wallet is created once, you do not need to re-create it. After the database comes up, you just have to open it using the same password via the command:

   ```
       ALTER SYSTEM SET ENCRYPTION WALLET OPEN IDENTIFIED
   BY "secretpassword";
   ```

   You can close the wallet by issuing the command:

   ```
       ALTER SYSTEM SET ENCRYPTION WALLET CLOSE;
   ```

   The wallet needs to be open for TDE to work. If the wallet is not open, all non-encrypted columns are accessible, but the encrypted columns are not.

Examples to encrypt data

Transparent Data Encryption

Transparent Data Encryption {TDE) is an Oracle database feature for encrypting sensitive data within the Oracle datafiles to prevent external access to it via the operating system.

EXAMPLE:

Create a "wallet" directory in $ORACLE_BASE/admin/$ORACLE_SID where Oracle can store its encryption key. If not, you will get error: ORA-28368: cannot auto-create wallet.

```
$ mkdir /u01/app/oracle/admin/orcl/wallet
```

Create the wallet to hold the encryption key:

login as sys

```
SQL> ALTER SYSTEM SET ENCRYPTION KEY AUTHENTICATED BY "secretpassword";
```

The above command

- Creates the wallet in the location specified above.
- Sets the password of the wallet as "secretpassword"
- Opens the wallet for TDE to store and retrieve the master key

The password is case-sensitive and must be enclosed in double quotes.

Create a table with encrypted columns:

login as hr/hr

```
SQL> CREATE TABLE tde_test (
    id   NUMBER,
    data VARCHAR2(30) ENCRYPT);
```

```
SQL> INSERT INTO tde_test (id, data) VALUES (1, 'This data in encrypted!');
```

Select from the table to see the data (wallet is still open):

```
SQL>  SELECT data FROM tde_test;

DATA
------------------------------
This data in encrypted!
```

Closing the wallet to prevent access to encrypted columns:

```
SQL> ALTER SYSTEM SET WALLET CLOSE;
```

Select from the table to see the data (wallet is closed):

```
SQL> SELECT data FROM tde_test;
SELECT data FROM tde_test
                 *
ERROR at line 1:
ORA-28365: wallet is not open
```

The wallet must be open for TDE to work. If the wallet is closed, you can access all nonencrypted columns, but not encrypted columns.

It is possible to encrypt existing column of any existing table also...

```
SQL> create table student(
       rollno varchar2(20),
       name varchar2(20));


SQL> Alter table student
       modify (rollno encrypt);
```

This statement does two things:

- It creates an encryption key for the table. If you change another column in the same table to use the encrypted format, the same table key will be used.
- It converts all values in the column to encrypted format.

By default, the algorithm AES with 192-bit key is used to encrypt. We can also choose a different algorithm by specifying the appropriate additional clause in the command. For instance, to use 128-bit AES encryption, we can use

```
SQL> alter table student modify (name encrypt using 'AES128');
```

We can use AES128, AES192, AES256, or 3DES168 (168-bit Triple DES algorithm) as clauses. The values are self-explanatory; for instance, AES256 is for Advanced Encryption Standard algorithm with 256-bit key.

After encrypting the column, we will see the following when you describe the table:

```
SQL>desc scott.student;

Name                    Null?          Type
----              -----         ----
ROLLNO            VARCHAR2(20)       ENCRYPT
NAME              VARCHAR2(20)       ENCRYPT
```

NOTE: To find the encrypted columns in the database, we can search the data dictionary view DBA_ENCRYPTED_COLUMNS. (TDE can't be enabled on a SYS-owned table.)

```
ORCL>select table_name,column_name,encryption_alg from
dba_encrypted_columns where table_name='STUDENT' and
owner='SCOTT';



TABLE_NAME          COLUMN_NAME
    ENCRYPTION_ALG

----------          -----------           --------------

STUDENT             ROLLNO                        AES 192 bits
key

STUDENT             NAME                          AES 192 bits
key
```

Performance Considerations

Since encryption and decryption consume CPU cycles, we must consider their effect on performance. When we access the nonencrypted columns of a table, the performance isn't any different from tables without TDE. When you access encrypted columns, however, there's a small performance overhead while decrypting during selects and encrypting during inserts, so we might want to encrypt columns selectively. If we no longer need to encrypt a column, we can turn it off with the following:

```
SQL> alter table student modify (name decrypt);
```

Key and Password Management

What if someone learns of the table keys or we suspect someone may have decrypted the encrypted table keys? We can simply create a new key for the table—in other words, rekey the table—and recreate the encrypted column values using the new table key by issuing a simple command. While we are at it, we might also want to choose a different encryption algorithm such as AES256. We can do both by issuing

```
SQL> alter table student rekey using 'aes256';
```

What if someone learns of the wallet password? We can change it via Oracle Wallet Manager. To invoke this graphical tool, type OWM in the command line . From the top menu, choose Wallet -> Open and choose the wallet location we have specified, and then give the password of the wallet. After that, choose Wallet -> Change Password to change the password. Note that changing the wallet password does not change the keys.

Want "Salt" with That?

Encryption is all about hiding data, but sometimes it's easier to guess the value of encrypted data if there's repetition in the original plain text value of the data. For instance, a salary information table may contain repeated values. In that case, the encrypted values will be the same, too, and an intruder could determine all entries with the same salary. To prevent such an occurrence, a "salt" is added to the data that makes the encrypted value different even if the input data is same. TDE, by default, applies a salt.

If you try to create an index on an encrypted column, however, you can't include a salt in it. To remove the salt from the SSN column, for example, execute the following:

```
SQL> alter table student modify
(rollno encrypt no salt);
```

If we try to create an index on a column that's encrypted with a salt, we will get an error, as in the following example:

```
SQL> create index in_std_01
on student (rollno);
```

```
ORA-28338: cannot encrypt indexed column(s) with salt
```

We will get the same error if we try to encrypt an indexed column with salt. Similarly, we can't use salt if there's an implicit index creation, such as when the columns are part of primary key or are defined as unique. Also, we can't use salt if the columns are part of a foreign key.

Using Data Pump with TDE

By default, if we use the Data Pump export utility (EXPDP) to export data from a table with encrypted columns, the data in the resulting dump file will be in clear text, even the encrypted column data.

The following command exports the EMP table—with its encrypted columns—and returns a warning:

```
$ expdp scott/tiger tables=emp
```

```
ORA-39173: Encrypted data has been stored unencrypted in dump file set.
```

This is just a warning, not an error; the rows will still be exported.

To protect encrypted column data in the data pump dump files, we can password-protect your dump file when exporting the table. This password, specified as an ENCRYPTION_PASSWORD parameter in the EXPDP command, applies to this export process only; this is not the password of the wallet. Following example shows the EXPDP command issued with the password "password".

Note how the output of the command in following example does not show the password "password"; it's hidden as a string of asterisks. The resulting dump file will not have visible clear text data for columns encrypted with TDE.

Exporting a password-protected dump file

```
$ expdp scott/tiger ENCRYPTION_PASSWORD=password tables=emp

Export: Release 10.2.0.0.0 - Beta on Friday, 01 July, 2005 16:14:06

Copyright (c) 2003, 2005, Oracle.  All rights reserved.

Connected to: Oracle Database 10g Enterprise Edition Release 10.2.0.0.0 -
Beta
With the Partitioning, OLAP and Data Mining options
Starting "SCOTT"."SYS_EXPORT_TABLE_01":  SCOTT/********
ENCRYPTION_PASSWORD=********* tables=EMP
Estimate in progress using BLOCKS method...
Processing ...
```

When we import this encrypted dump file, we have to provide the same password used with the export, as shown in following example

Importing a password-protected dump file

```
$ impdp scott/tiger ENCRYPTION_PASSWORD=password tables=emp
table_exists_action=replace

Import: Release 10.2.0.0.0 - Beta on Friday, 01 July, 2005 16:04:20

Copyright (c) 2003, 2005, Oracle.  All rights reserved.

Connected to: Oracle Database 10g Enterprise Edition Release 10.2.0.0.0 -
Beta
With the Partitioning, OLAP and Data Mining options
Master table "ARUP"."SYS_IMPORT_TABLE_01" successfully
loaded/unloaded
Starting "SCOTT"."SYS_IMPORT_TABLE_01":  Scott/********
ENCRYPTION_PASSWORD=********* table_exists_action=replace
Processing ...
```

The following shows the result if you omit the ENCRYPTION_PASSWORD parameter during the import:

```
$ impdp scott/tiger tables=emp


ORA-39174: Encryption password must
be supplied.
```

The following shows the result if you provide the wrong password:

```
$ impdp scott/tiger ENCRYPTION_PASSWORD=piglet tables=accounts
ORA-39176: Encryption password is incorrect.
```

NOTE: The original export utility (EXP) cannot export a table with encrypted columns.

Encryption of data using DBMS packages

DATABASE ENCRYPTION

Introduction

Oracle has many security features available within database, encryption of data is one of the security feature that enables a user to encipher data and store it in a different format.

Encryption can be done following ways-

DBMS_OBFUSCATION_TOOLKIT(9i).

DBMS_CRYPTO.

Transparent Data Encryption.

Virtual Private Database Policy(10g).

DBMS_CRYPTO is intended to replace the DBMS_OBFUSCATION-

_TOOLKIT

DBMS_OBFUSCATION_TOOLKIT(9i).

DBMS_OBFUSCATION_TOOLKIT package provides a simple API for data encryption.

Encryption with this toolkit is done in the following steps-

Creating the package

```
CREATE OR REPLACE PACKAGE toolkit AS
  FUNCTION encrypt (p_text  IN  VARCHAR2) RETURN RAW;
  FUNCTION decrypt (p_raw  IN  RAW) RETURN VARCHAR2;
END toolkit;
/
```

Create the package body-

All VARCHAR2 inputs are padded to multiples of 8 charaters, with the encryption key also being a multiple of 8 charaters. The encryption key and padding characters can be altered to suit.

```
CREATE OR REPLACE PACKAGE BODY toolkit AS
  g_key     RAW(32767)  := UTL_RAW.cast_to_raw('12345678');
  g_pad_chr VARCHAR2(1) := '~';

  PROCEDURE padstring (p_text  IN OUT  VARCHAR2);
  FUNCTION encrypt (p_text  IN  VARCHAR2) RETURN RAW IS
     l_text       VARCHAR2(32767) := p_text;
    l_encrypted  RAW(32767);
  BEGIN
    padstring(l_text);
    DBMS_OBFUSCATION_TOOLKIT.desencrypt(input          =>
UTL_RAW.cast_to_raw(l_text),
 key            => g_key,
encrypted_data => l_encrypted);
    RETURN l_encrypted;
  END;



FUNCTION decrypt (p_raw  IN  RAW) RETURN VARCHAR2 IS
    l_decrypted  VARCHAR2(32767);
  BEGIN
    DBMS_OBFUSCATION_TOOLKIT.desdecrypt(input => p_raw,
                                        key   => g_key,
                                       decrypted_data => l_decrypted);

    RETURN RTrim(UTL_RAW.cast_to_varchar2(l_decrypted), g_pad_chr);
  END;


  PROCEDURE padstring (p_text  IN OUT  VARCHAR2) IS
    l_units  NUMBER;
  BEGIN
    IF LENGTH(p_text) MOD 8 > 0 THEN
      l_units := TRUNC(LENGTH(p_text)/8) + 1;
      p_text  := RPAD(p_text, l_units * 8, g_pad_chr);
    END IF;
  END;
END toolkit;
```

```
/
```

Encrypt Table Data

```
CREATE TABLE encrypted_data (
  username  VARCHAR2(20),
  data      RAW(16)
);

CREATE OR REPLACE TRIGGER encrypted_data_biur_trg
BEFORE INSERT OR UPDATE ON encrypted_data
FOR EACH ROW
DECLARE
BEGIN
  :new.data := toolkit.encrypt(UTL_RAW.cast_to_varchar2(:new.data));
END;
/
```

we test the trigger using some simple insert, update and query statements.

```
SQL> INSERT INTO encrypted_data (username, data)
  2  VALUES ('tim_hall', UTL_RAW.cast_to_raw('My Secret Data'));

1 row created.

SQL> SELECT * FROM encrypted_data;

USERNAME             DATA
-------------------- ----------------------------------------
tim_hall             FA57C55510D258C73DE93059E3DC49EC

1 row selected.

SQL> COLUMN data FORMAT A40
SQL> SELECT username, toolkit.decrypt(data) AS data FROM
encrypted_data;
```

```
USERNAME            DATA
-------------------- ----------------------------------------
tim_hall            My Secret Data

1 row selected.

SQL> UPDATE encrypted_data
  2  SET    data     = UTL_RAW.cast_to_raw('My NEW Secret')
  3  WHERE  username = 'tim_hall';

1 row updated.

SQL> COLUMN data FORMAT A40
SQL> SELECT username, toolkit.decrypt(data) AS data FROM
encrypted_data;

USERNAME            DATA
-------------------- ----------------------------------------
tim_hall            My NEW Secret

1 row selected.

SQL>
```

With the exception of the calls to the UTL_RAW package, this method hides most of the work from the developer.

DBMS_CRYPTO

DBMS_CRYPTO provides an interface to encrypt and decrypt stored data, and can be used in conjunction with PL/SQL programs running network communications. It provides support for several industry-standard encryption and hashing algorithms, including the Advanced Encryption Standard (AES) encryption algorithm. AES has been approved by the National Institute of Standards and Technology (NIST) to replace the Data Encryption Standard (DES).

EXAMPLE-

The following listing shows PL/SQL block encrypting and decrypting pre-defined 'input_string' using 256-bit AES algorithm with Cipher Block Chaining and PKCS#5 compliant padding.

```
DECLARE
   input_string       VARCHAR2 (200) :=  'Secret Message';
   output_string      VARCHAR2 (200);
   encrypted_raw      RAW (2000);            -- stores encrypted binary text
   decrypted_raw      RAW (2000);            -- stores decrypted binary text
   num_key_bytes      NUMBER := 256/8;       -- key length 256 bits (32
bytes)
   key_bytes_raw      RAW (32);              -- stores 256-bit encryption key
   encryption_type    PLS_INTEGER :=         -- total encryption type
                       DBMS_CRYPTO.ENCRYPT_AES256
                     + DBMS_CRYPTO.CHAIN_CBC
                     + DBMS_CRYPTO.PAD_PKCS5;
BEGIN
   DBMS_OUTPUT.PUT_LINE ( 'Original string: ' || input_string);
   key_bytes_raw := DBMS_CRYPTO.RANDOMBYTES (num_key_bytes);
   encrypted_raw := DBMS_CRYPTO.ENCRYPT
      (
         src => UTL_I18N.STRING_TO_RAW (input_string,  'AL32UTF8'),
         typ => encryption_type,
         key => key_bytes_raw
      );
    -- The encrypted value "encrypted_raw" can be used here

   decrypted_raw := DBMS_CRYPTO.DECRYPT
      (
         src => encrypted_raw,
         typ => encryption_type,
         key => key_bytes_raw
      );
```

output_string := UTL_I18N.RAW_TO_CHAR (decrypted_raw, 'AL32UTF8');

Transparent Data Encryption (TDE) in Oracle 10g Database Release 2

Transparent Data Encryption (TDE) feature introduced in Oracle 10g Database Release 2 allows sensitive data to be encrypted within the datafiles to prevent access to it from the operating system.

Setup

In order to show the encryption working we need to open a datafile in a HEX editor. Rather than trying to open a huge datafile, it makes sense to create a small file for this test.

```
CONN sys/password AS SYSDBA

CREATE TABLESPACE tde_test
  DATAFILE '/u01/oradata/DB10G/tde_test.dbf' SIZE 128K
  AUTOEXTEND ON NEXT 64K;
```

Next, create a user with with a quota on the new tablespace.

```
CREATE USER test IDENTIFIED BY test DEFAULT TABLESPACE tde_test;
ALTER USER test QUOTA UNLIMITED ON tde_test;
GRANT CONNECT TO test;
GRANT CREATE TABLE TO test;
```

Normal Column

First we will prove that the data from a normal column can be seen from the OS. To do this create a test table and insert some data.

```
CONN test/test

CREATE TABLE tde_test (
  id    NUMBER(10),
  data  VARCHAR2(50)
)
TABLESPACE tde_test;

INSERT INTO tde_test (id, data) VALUES (1, 'This is a secret!');
COMMIT;
```

Then flush the buffer cache to make sure the data is written to the datafile.

```
CONN sys/password AS SYSDBA
ALTER SYSTEM FLUSH BUFFER_CACHE;
```

Open the datafile using a HEX editor (like UltraEdit) and the sentence "This is a secret!" is clearly visible amongst all the non-printable characters.

Encrypted Column

Before attempting to create a table with encrypted columns, a wallet must be created to hold the encryption key. The search order for finding the wallet is as follows:

If present, the location specified by the ENCRYPTION_WALLET_LOCATION parameter in the sqlnet.ora file.

If present, the location specified by the WALLET_LOCATION parameter in the sqlnet.ora file.

The default location for the wallet ($ORACLE_BASE/admin/$ORACLE_SID/wallet).

Although encrypted tablespaces can share the default database wallet, Oracle recommend you use a separate wallet for transparent data encryption functionality by specifying the ENCRYPTION_WALLET_LOCATION parameter in the sqlnet.ora file. To accomplish this we add the following entry into the sqlnet.ora file on the server and make sure the specified directory has been created.

```
ENCRYPTION_WALLET_LOCATION=
  (SOURCE=(METHOD=FILE)(METHOD_DATA=
    (DIRECTORY=/u01/app/oracle/admin/DB10G/encryption_wallet/)))
```

The following command creates and opens the wallet.

```
CONN sys/password AS SYSDBA
ALTER SYSTEM SET ENCRYPTION KEY AUTHENTICATED BY "myPassword";
```

Wallets must be reopened after an instance restart and can be closed to prevent access to encrypted columns.

```
ALTER SYSTEM SET WALLET OPEN IDENTIFIED BY "myPassword";

ALTER SYSTEM SET WALLET CLOSE;
```

Create a test table with an encrypted column and insert some data. Using the ENCRYPT clause on its own is the same as using the ENCRYPT USING 'AES192' clause, as AES192 is the default encryption method.

```
CONN test/test

DROP TABLE tde_test;
PURGE RECYCLEBIN;

CREATE TABLE tde_test (
  id    NUMBER(10),
  data  VARCHAR2(50) ENCRYPT
)
TABLESPACE tde_test;

INSERT INTO tde_test (id, data) VALUES (1, 'This is a secret!');
COMMIT;
```

Flush the buffer cache to make sure the data is written to the datafile.

```
CONN sys/password AS SYSDBA
ALTER SYSTEM FLUSH BUFFER_CACHE;
```

When the file is opened using a HEX editor only non-printable characters are present. The test sentence cannot be seen anywhere, but the data is still clearly visible from a database connection.

```
SELECT * FROM tde_test;

        ID DATA
---------- --------------------------------------------------
         1 This is a secret!

1 row selected.
```

Performance

There is a performance overhead associated with the encryption/decryption process. The following tables are used in a performance comparison.

```
CONN test/test
CREATE TABLE tde_test_1 (
  id    NUMBER(10),
  data  VARCHAR2(50)
)
TABLESPACE tde_test;

CREATE TABLE tde_test_2 (
  id    NUMBER(10),
  data  VARCHAR2(50) ENCRYPT
)
TABLESPACE tde_test;
```

The following script uses these tables to compare the speed of regular and encrypted inserts and regular and decrypted queries. Each test repeats 1000 times, with the timings reported in 100ths of a second.

```
SET SERVEROUTPUT ON SIZE UNLIMITED
DECLARE
  l_loops  NUMBER := 1000;
  l_data   VARCHAR2(50);
  l_start  NUMBER;
BEGIN
  EXECUTE IMMEDIATE 'TRUNCATE TABLE tde_test_1';
  EXECUTE IMMEDIATE 'TRUNCATE TABLE tde_test_2';

  l_start := DBMS_UTILITY.get_time;
  FOR i IN 1 .. l_loops LOOP
    INSERT INTO tde_test_1 (id, data)
    VALUES (i, 'Data for ' || i);
  END LOOP;
  DBMS_OUTPUT.put_line('Normal Insert   : ' ||
(DBMS_UTILITY.get_time - l_start));

  l_start := DBMS_UTILITY.get_time;
  FOR i IN 1 .. l_loops LOOP
    INSERT INTO tde_test_2 (id, data)
    VALUES (i, 'Data for ' || i);
  END LOOP;
  DBMS_OUTPUT.put_line('Encrypted Insert: ' ||
(DBMS_UTILITY.get_time - l_start));

  l_start := DBMS_UTILITY.get_time;
  FOR i IN 1 .. l_loops LOOP
    SELECT data
    INTO   l_data
    FROM   tde_test_1
    WHERE  id = i;
  END LOOP;
  DBMS_OUTPUT.put_line('Normal Query    : ' ||
(DBMS_UTILITY.get_time - l_start));

  l_start := DBMS_UTILITY.get_time;
  FOR i IN 1 .. l_loops LOOP
    SELECT data
    INTO   l_data
    FROM   tde_test_2
    WHERE  id = i;
  END LOOP;
  DBMS_OUTPUT.put_line('Decrypted Query : ' ||
(DBMS_UTILITY.get_time - l_start));
END;
/
Normal Insert   : 31
Encrypted Insert: 45
Normal Query    : 42
Decrypted Query : 58

PL/SQL procedure successfully completed.

SQL>
```

The results clearly demonstrate that encrypted inserts and decrypted queries are slower than their normal counterparts.

Virtual Private Database Policy-

Virtual Private Database (VPD) Enhancements

Column-Level VPD Policy

In conventional Virtual Private Database the VPD Policy is applied to the whole row. By default a Column-Level VPD Policy allows you to restrict the rows displayed only if specified columns are accessed.

```
CONN sys/password@db10g AS SYSDBA

GRANT EXECUTE ON dbms_rls TO scott;

CONN scott/tiger@db10g

-- Create the policy function to restrict access to SAL and COMM columns

-- if the employee is not part of the department 20.

CREATE OR REPLACE FUNCTION pf_job (oowner IN VARCHAR2,
ojname IN VARCHAR2)

RETURN VARCHAR2 AS

  con VARCHAR2 (200);

BEGIN

  con := 'deptno = 20';

  RETURN (con);

END pf_job;

/


-- Apply the policy function to the table.

BEGIN

  DBMS_RLS.ADD_POLICY (object_schema     => 'scott',

                       object_name       => 'emp',

                       policy_name       => 'sp_job',

                       function_schema   => 'scott',

                       policy_function   => 'pf_job',

                       sec_relevant_cols => 'sal,comm');

END;

/


-- We see all records if SAL and COMM are not referenced
```

```
SELECT empno, ename, job FROM emp;

     EMPNO ENAME      JOB
---------- ---------- ---------
      7369 SMITH      CLERK
.
.
      7934 MILLER     CLERK

14 rows selected.

-- Rows are restricted if SAL or COMM are referenced.
SELECT empno, ename, job, sal, comm FROM emp;

     EMPNO ENAME      JOB              SAL       COMM
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK          10000
      7566 JONES      MANAGER         2975
      7788 SCOTT      ANALYST         3000
      7876 ADAMS      CLERK           1100
      7902 FORD       ANALYST         3000

5 rows selected.

-- Remove the policy function from the table.
BEGIN
  DBMS_RLS.DROP_POLICY (object_schema     => 'scott',
                        object_name       => 'emp',
                        policy_name       => 'sp_job');
END;
/
```

Column Masking

Column masking behaviour is implemented by using the "sec_relevant_cols_opt => DBMS_RLS.ALL_ROWS" parameter. This allows you to display all rows but mask the values of the specified columns for the restricted rows.

```
-- Using the same policy function as before.
BEGIN
  DBMS_RLS.ADD_POLICY (object_schema         => 'scott',
                       object_name           => 'emp',
                       policy_name           => 'sp_job',
                       function_schema       => 'scott',
                       policy_function       => 'pf_job',
                       sec_relevant_cols     => 'sal,comm',
                       sec_relevant_cols_opt => DBMS_RLS.ALL_ROWS);
END;
/

-- All rows are returned but the SAL and COMM values are only
-- shown for employees in department 20.
SELECT empno, ename, job, sal, comm FROM emp;

     EMPNO ENAME      JOB              SAL       COMM
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK          10000
      7499 ALLEN      SALESMAN
      7521 WARD       SALESMAN
      7566 JONES      MANAGER         2975
      7654 MARTIN     SALESMAN
      7698 BLAKE      MANAGER
      7782 CLARK      MANAGER
      7788 SCOTT      ANALYST         3000
      7839 KING       PRESIDENT
      7844 TURNER     SALESMAN
      7876 ADAMS      CLERK           1100

     EMPNO ENAME      JOB              SAL       COMM
---------- ---------- --------- ---------- ----------
      7900 JAMES      CLERK
```

```
   7902 FORD       ANALYST        3000
   7934 MILLER     CLERK

14 rows selected.

-- Remove the policy function from the table.
BEGIN
  DBMS_RLS.DROP_POLICY (object_schema     => 'scott',
                        object_name       => 'emp',
                        policy_name       => 'sp_job');
END;
/
```

Policy Types

The correct use of policy types can increase the performance of VPD by caching the output of the policy function and applying it to subsequent queries without executing the policy function again. The POLICY_TYPE parameter of the DBMS_RLS.ADD_POLICY procedure is used to set one of the five policy types:

STATIC - The return value of the policy function is cached and reused repeatedly for an individual object. By definition the return value of the policy function must be static.

SHARED_STATIC - The same as STATIC but the resulting predicate can be applied to several objects.

CONTEXT_SENSITIVE - Used when policy is based around local application context. The result of the policy function is cached and reused. The policy function is only executed again when the value of the application context changes.

SHARED_CONTEXT_SENSITIVE - The same as CONTEXT_SENSITIVE but the resulting predicate can be applied to several objects.

DYNAMIC - The policy function is executed for every SQL statement.

An example of it's usage is shown below.

```
BEGIN
  DBMS_RLS.ADD_POLICY (object_schema        => 'scott',
                      object_name          => 'emp',
                      policy_name          => 'sp_job',
                      function_schema      => 'scott',
                    policy_function       => 'pf_job',
                    policy_type           => DBMS_RLS.STATIC,
                    sec_relevant_cols     => 'sal,comm',
                    sec_relevant_cols_opt => DBMS_RLS.ALL_ROWS);
END;
/
```

Disaster Recovery solution

One the most useful thing in disaster recovery is having standby database for production database. If disaster happens where there is primary database we can easily make standby database as primary and till then we can recover the original primary database by the steps mention at the last of this document. Standby is the major solution for disaster with zero data loss.

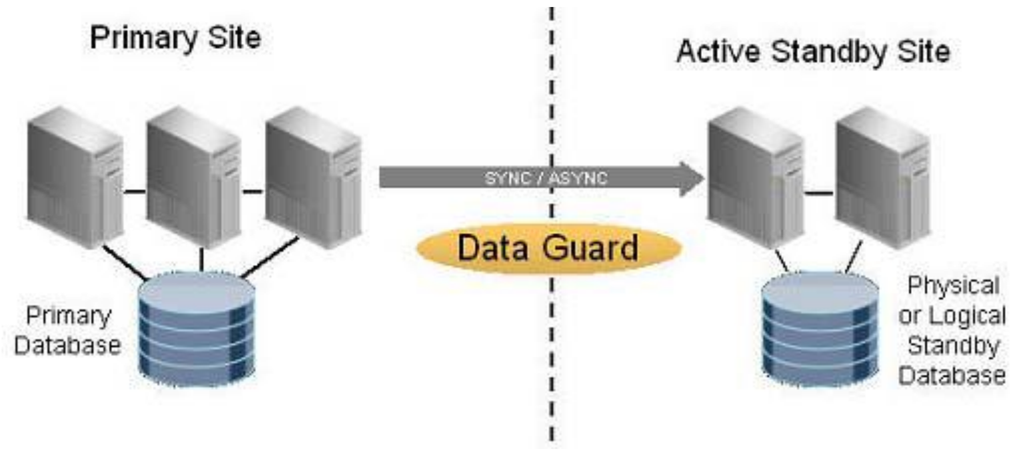
Introduction

Efficient business operations, quality customer service, compliance with government regulations, and safeguarding corporate information assets all require high levels of data protection and data availability. Thus it is no surprise that data protection and data availability are among the top priorities for enterprises of all sizes and industries.
Recovery using off-site backups or storage remote-mirroring, are traditional data protection and disaster recovery (DR) solutions for enterprise data. Unfortunately, these solutions are unable to reliably deliver aggressive recovery point (RPO - data protection) and recovery time (RTO - data availability) objectives. They also fail to provide adequate return on investment due to poor utilization of standby systems that sit idle until a failure occurs.
In contrast, Oracle Data Guard 11g redefines expectations for data protection solutions. Data Guard is the data protection and availability solution for Oracle Database. It provides the management, monitoring, and automation software to create and maintain one or more synchronized standby databases that protect data from failures, disasters, errors, and corruptions. It can address both High Availability and Disaster Recovery requirements and is the ideal complement to Oracle Real Application Clusters.
Data Guard has the requisite knowledge of the Oracle database to provide the highest level of protection for Oracle data. Data Guard is straightforward to implement and manage. Administrators can be certain of the ability of a standby database to assume the production role – eliminating business risk at failover time. Data Guard standby databases also deliver high return on investment when used for queries, reports, backups, testing, or rolling database upgrades and other maintenance, while also providing disaster protection.

Oracle Data Guard11g- Overview

Oracle data guard maintains one or more standby databases to protect data in oracle database from failures, Disasters, errors, corruptions. Administrators can chose either manual or automatic failover of production to a standby system if the primary system fails in order to maintain high availability for mission.

An overview of Data Guard architecture is provided in Figure.



There are two types of standby databases Physical and Logical standby database but A physical standby uses Redo Apply to maintain a block for block, exact replica of the primary database. Physical standby databases provide the best disaster recovery (DR) protection for the Oracle Database.

Physical standby database

Provides a physically identical copy of the primary database, with on disk database structures that are identical to the primary database on a block-for-block basis. The database schemas, including indexes, are the same. A physical standby database is kept synchronized with the primary database, though Redo Apply, which recovers the redo data, received from the primary database and applies the redo to the physical standby database.

A physical standby database can be used for business purposes other than disaster recovery on a limited basis.

Logical standby database

Contains the same logical information as the production database, although the physical organization and structure of the data can be different. The logical standby database is kept synchronized with the primary database though SQL Apply, which transforms the data in the redo received from the primary database into SQL statements and then executing the SQL statements on the standby database.

The main difference between physical and logical standby databases is the manner in which log apply services apply the archived redo data:

For physical standby databases, Data Guard uses Redo Apply technology, which applies redo data on the standby database using standard recovery techniques of an Oracle database.

For logical standby databases, Data Guard uses SQL Apply technology, which first transforms the received redo data into SQL statements and then executes the generated SQL statements on the logical standby database

Role Transitions: Switchover and Failover

Switchover: -

The switchover feature provides you with the ability to switch the role of the primary database to one of the available standby databases. The chosen standby database becomes the primary database, and the original primary database then becomes a standby database.

Failover: -

You invoke a failover operation when a catastrophic failure occurs on the primary database and there is no possibility of recovering the primary database in a timely manner. During a failover operation, the failed primary database is removed from the Data Guard environment, and a standby database assumes the primary database role. You invoke the failover operation on the standby database that you want to fail over to the primary role.

Data Guard Protection Modes

Standby Modes

Maximum protection

This protection mode ensures that no data loss will occur if the primary database fails. To provide this level of protection, the redo data needed to recover each transaction must be written to both the local online redo log and to the standby redo log on at least one standby database before the transaction commits. To ensure data loss cannot occur, the primary database shuts down if a fault prevents it from writing its redo stream to the standby redo log of at least one transactionally consistent standby database.
(This mode offers the highest level of data protection. Data is synchronously transmitted to the standby database from the primary database and transactions are not committed on the primary database unless the redo data is available on at least one standby database configured in this mode. If the last standby database configured in this mode becomes unavailable, processing stops on the primary database. This mode ensures no-data-loss.)

Maximum availability

This protection mode provides the highest level of data protection that is possible without compromising the availability of the primary database. Like maximum protection mode, a transaction will not commit until the redo needed to recover that transaction is written to the local online redo log and to the standby redo log of at least one transactionally consistent standby database. Unlike maximum protection mode, the primary database does not shut down if a fault prevents it from writing its redo stream to a remote standby redo log. Instead, the primary database operates in maximum performance mode until the fault is corrected, and all gaps in redo log files are resolved. When all gaps are resolved, the primary database automatically resumes operating in maximum availability mode.

This mode ensures that no data loss will occur if the primary database fails, but only if a second fault does not prevent a complete set of redo data from being sent from the primary database to at least one standby database.
(This mode is similar to the maximum protection mode, including zero data loss. However, if a standby database becomes unavailable (for example, because of network connectivity problems), processing continues on the primary database. When the fault is corrected, the standby database is automatically resynchronized with the primary database.)

Maximum Performance

This is the default protection mode. This protection mode provides the highest level of data protection that is possible without affecting the performance of a primary database. This is accomplished by allowing transactions to commit as soon as all redo data generated by those transactions has been written to the online log. Redo data is also written to one or more standby databases, but this is done asynchronously with respect to transaction commitment, so primary database performance is unaffected by delays in writing redo data to the standby databases.

This protection mode offers slightly less data protection than maximum availability mode and has minimal impact on primary database performance.
(This mode offers slightly less data protection on the primary database, but higher performance than maximum availability mode. In this mode, as the primary database processes transactions, redo data is asynchronously shipped to the standby database. The commit operation of the primary database does not wait for the standby database to acknowledge receipt of redo data before completing write operations on the primary database. If any standby destination becomes unavailable, processing continues on the primary database and there is little effect on primary database performance.)

Data Guard Benefits

Disaster recovery, data protection, and high availability

Data Guard provides an efficient and comprehensive disaster recovery and high availability solution. Easy-to-manage switchover and failover capabilities allow role reversals between primary and standby databases, minimizing the downtime of the primary database for planned and unplanned outages.

Standby database

A standby database maintains a duplicate, or standby copy of your primary (also known as production database) and provides continued primary database availability in the event of a disaster (when all media is destroyed at your production site). A standby database is constantly in recovery mode. If a disaster occurs, you can take the standby database out of recovery mode and activate it for online use.

Some of the characteristics of Standby Databases are:

You cannot query or open a standby database for any other purpose other than to activate disaster recovery. i.e. standby database is just a dummy sitting at a different location always in recovery mode, in case the production database goes down, then the standby database can be activated to become production.

Once you activate your standby database, i.e. it becomes production, you cannot return it back to standby mode, because the standby database is now your current production, and you will have to re-create another standby database for the current production (your previous standby).

You must place the datafiles, log files and control files of your primary and standby database on separate physical media. More Preferable to place the standby database on separate machine and media.

It is advised to keep the datafile names and location/directories, the same on your production and standby database, if this is not possible use the datafile name conversion parameters. If you do not follow either of the approaches you may end up crashing your standby database.

Your production database should be in archivelog mode, to create standby database. Else creating standby database makes no sense.

To implement the standby database, you should have followed OFA, wherein all the datafiles should be present in single mount point, say /u01

Create a Standby database

In order to create a standby database, the init.ora parameter COMPATIBLE must be 7.3 or higher, and the value of COMPATIBLE in the standby database *must* be identical to its value in the primary database. Create the control file on the Primary database (this will be used to build your standby database), substitute the filename with your choice. Once the command is executed the file with the name you have specified (filename) is created, usually in BACKGROUND_DUMP_DEST directory. It is advisable to have the filename (control filename) the same as the one you have in your production database.

ALTER DATABASE CREATE STANDBY CONTROLFILE AS filename

Archive the current online logs of the production database. So that the most recent transaction activities is trapped in your archive files. This command also ensures consistency among data files, control files, and log files.

ALTER SYSTEM ARCHIVE LOG CURRENT

Backup the datafiles, redolog files (either online or offline) of your production database.
Transfer/Copy the control file, backed up data files and logfiles, archived redo logs to a remote machine using appropriate method (FTP, TAR.etc.)

Copy the initprod.ora file pertaining to your production database to the server where standby database resides (let us call this file initstandby.ora).

Add the following entries in your initstandyby.ora file for your standby database. These entries are added so that all filenames (datafiles listed in your control file) from your production database control file are converted for use by your standby database. For example you have your production datafiles in location /oracle and you want your datafiles pertaining to your standby database in directory /oracle/dbfiles then

DB_FILE_STANDBY_NAME_CONVERT="/oracle","/oracle/standby"

You can also set your LOG_FILE_STANDBY_NAME_CONVERT accordingly.

Start up the Oracle instance of the standby database using NO MOUNT option.

STARTUP NOMOUNT

Mount the database in standby mode

ALTER DATABASE MOUNT STANDBY DATABASE [EXCLUSIVE / PARALLEL]

Place the standby database in recovery mode

RECOVER [FROMlocation] STANDBY DATABASE

As the archive logs are generated on your production box, you should continually transfer and apply them to the standby database. So that your standby and production are in sync.
In case of disaster (production database being lost due to media failure), your should follow the following steps to activate your standby database and make it production:

Activate your standby database

ALTER DATABASE ACTIVATE STANDBY DATABASE

Shutdown your standby database

SHUTDOWN

Startup the standby database, which will be now your production database.

STARTUP