



# Encrypting Data Inside the Database

By Rick Boesen

**I**n this era of identity theft and corporate corruption, additional measures to further protect sensitive or confidential data may be prudent. Encrypting data inside the database with a strong encryption algorithm further protects data from those who may not have authority to view it.

By using an SQL editor, an end-user can query the database and retrieve data. However, even if your application is designed well, there are probably security loop-holes that have been overlooked. Although not widely reported, data theft occurs. Data encryption can further protect data from those who wish to “accidentally” browse tables and objects that they may not have authority to read.

In order to provide some protection from unauthorized access or use of data, roles and privileges are utilized. Even privileges through a proxy user must be granted basic rights to read data tables. Inevitably, someone must be trusted with grants or privileges that could be exploited. Data encryption can assist in reducing the risk of those who wish to find and thus exploit those privileges that have been granted to administer or perform higher end-user tasks.

To help ensure that people do not “accidentally” read protected or confidential data, Oracle has provided a means to encrypt data within tables using the DES or DES3 encryption techniques.

Data Encryption Standard (DES) is an encryption method used to secretly send and receive data. DES applies a 56-bit key to a 64-bit block of data. Although this method of encryption is considered strong by some government agencies and private companies, free versions of the DES algorithm are available from various web sites.

“Triple DES” or DES3, however, uses a longer encryption key and thus is considered a stronger encryption method than DES. In this article the DES3 encryption provides an eight-character key that has 4,304,672,100,000,000 (90^8) permutations, repeated three times.

The DES and DES3 encryption algorithms can be used to encrypt passwords, social security numbers, or proprietary information for confidential storage. By encrypting data, the information becomes safer from intentional as well as accidental threats.

## Purpose

The purpose of this article is to show how sensitive or confidential data can be further protected beyond just using roles and privileges. As it is the stronger of the two encryption algorithms available in Oracle, the DES3 encryption technique is preferred. Examples involving implementation of a two-table application will demonstrate how to encrypt and decrypt data within a table.

## Key Function

Like other encryption algorithms, DES3 requires a key. In DES3, the key is composed of eight bytes repeated three times, and can be created using the dbms\_random function. The create\_des3\_key (Example 1) function provides this application with the appropriate 24-byte DES3 key.

```
create or replace function create_des3_key
return varchar2 is
a int :=0;
b int :=0;
crypto_key varchar2(24) default null;
key_length int := 8;
begin
for b in 1..key_length loop
a := to_number(substr(to_char(abs(dbms_random.random)),1,2));
while a > 90 loop
a := to_number(substr(to_char(abs(dbms_random.random)),1,2));
end loop;
a := a + 33;
crypto_key := crypto_key || chr(a);
end loop;
return(crypto_key || crypto_key || crypto_key);
end;
/
```

Example 1

continued on page 17

(Like most code, Example 1 can be optimized. This is just one possible method of creating the DES3 key.)

Character codes 33 to 122 are used to represent the encryption key. These character codes represent all the lower case and upper case alphabetic characters as well as numbers and some special characters.

Once the `create_des3_key` function is installed, it can be tested using the code in Example 2. Run this code several times to confirm that the key is changing.

```
select create_des3_key from dual
/

CREATE_DES3_KEY
-----
421/x/j0421/x/j0421/x/j0
```

## Example 2

### Application Tables

This example application utilizes two tables (Example 3). The first table, `data_table`, will hold encrypted data, and the second table, `key_table`, will hold the encryption key for each row within the data table. The tables are joined by their primary keys. Special care should be used to make certain a copy of the encryption key table is stored somewhere off the central system and easily accessible for recovery. If the keys are lost, data recovery will be necessary.

```
create table data_table
(pk_col1      number(2)      primary key,
 hex_encrypted_passwd varchar2(2048) not null,
 customer_no  varchar2(10),
 address      varchar2(35),
 city         varchar2(35),
 state        char(2),
 zipcode      varchar2(9))
tablespace users
/

create table key_table
(pk_col1      number(2) primary key,
 des3_key     varchar2(24))
tablespace users
/
```

## Example 3

(To easily view the data within `data_table`, the encrypted value has been changed to hex. This is only for ease of viewing and is not necessary for this application or these examples to run.)

At this point, a DES3 24-byte key can be retrieved and the tables manipulated. However, the data going in and out of the table still needs to be encrypted and decrypted, and the encryption key stored. All these tasks are performed through triggers attached to the `data_table`.

### To Obfuscate

The main database package used is `dbms_obfuscation_toolkit`. This package encrypts and decrypts data when provided with the appropriate values. To encrypt data using DES3, the `dbms_obfuscation_toolkit.des3encrypt` procedure requires four arguments. The arguments required are `input_string`, `key_string`, `encrypted_string`, and `which`. When attempting to encrypt data, use the `input_string` for the original data. The `key_string` will be the DES3 24-byte key. The `encrypted_string` is the value returned by the procedure. The final argument, `which`, is an input variable and defaults to 0 (for a 16-byte key) and 1 (for a 24-byte key).

To decrypt data, use the `dbms_obfuscation_toolkit.des3decrypt` procedure. This procedure only requires three arguments: `input_string`, `key_string`, and `decrypted_string`. The first argument, `input_string`, will be the encrypted data to be decrypted. The `key_string` will be the DES3 24-byte key. The `decrypted_string` is the output of the procedure and is the last argument.

### Triggers

The insert and update triggers use the `dbms_obfuscation_toolkit` package. The triggers receive the data to be encrypted, pass it to the `dbms_obfuscation_toolkit` package and return the hexadecimal representation of the encrypted data. DES3 requires that the length of the data to be encrypted be evenly divisible by eight. When the data to be encrypted is not modulus eight, spaces are padded to the end of the data in the `dt_ins_trig` trigger. Examples 4 and 5 contain the code for the insert and update triggers, respectively.

```
create or replace trigger dt_ins_trig
before insert on data_table
for each row
declare
  des3_24_key          varchar2(24) := create_des3_key;
  var_encrypted_string varchar2(2048);
begin
  if mod(length(:new.hex_encrypted_passwd),8) = 0
  then
    :new.hex_encrypted_passwd :=
      rpad(:new.hex_encrypted_passwd,8, ' ');
  else
    :new.hex_encrypted_passwd :=
      rpad(:new.hex_encrypted_passwd,
        round((length(:new.hex_encrypted_passwd) / 8) + .5) * 8, ' ');
  end if;
  dbms_obfuscation_toolkit.DES3Encrypt
  (
    input_string => :new.hex_encrypted_passwd,
    key_string   => des3_24_key,
    encrypted_string => var_encrypted_string,
    which        => 1
  );
  :new.hex_encrypted_passwd :=
    rawtohex(UTL_RAW.CAST_TO_RAW(var_encrypted_string));
  insert into key_table
  (pk_col1, des3_key) values
  (:new.pk_col1, des3_24_key);
end;
/
```

## Example 4

```
create or replace trigger dt_upd_trig
before update on data_table
for each row
declare
  des3_24_key          varchar2(24);
  var_encrypted_string varchar2(2048);
begin
  select des3_key into des3_24_key
  from key_table
  where pk_col1 = :new.pk_col1;
  dbms_obfuscation_toolkit.DES3Encrypt
  (
    input_string => :new.hex_encrypted_passwd,
    key_string   => des3_24_key,
    encrypted_string => var_encrypted_string,
    which        => 1
  );
  :new.hex_encrypted_passwd :=
    rawtohex(UTL_RAW.CAST_TO_RAW(var_encrypted_string));
end;
/
```

## Example 5

*continued on page 18*

The delete trigger on the `data_table` removes the matching row from the `key_table`. The only item necessary for this trigger is the primary key from the `data_table`. To keep the example simple, no exception handling has been added. The code for the delete trigger is shown in Example 6.

```
create or replace trigger dt_del_trig
before delete on data_table
for each row
begin
  delete key_table
  where pk_col1 = :old.pk_col1;
end;
/
```

### Example 6

#### Display Decrypted Data

The `get_passwd` function (Example 7) was created to retrieve and display the encrypted data. The primary key of the data table is passed to the function and is joined to the applicable key table. The decrypted value is then returned.

```
create or replace function get_passwd
(in_pk_col1 in varchar2)
return varchar2 is
var_hex_encrypted_string varchar2(2048);
var_des3_key_string varchar2(24);
var_encrypted_string varchar2(2048);
var_decrypted_string varchar2(2048);
begin
  select dt.hex_encrypted_passwd, kt.des3_key
  into var_hex_encrypted_string, var_des3_key_string
  from data_table dt, key_table kt
  where dt.pk_col1 = kt.pk_col1
  and dt.pk_col1 = in_pk_col1;
  var_encrypted_string :=
utl_raw.cast_to_varchar2(hextoraw(var_hex_encrypted_string));
  dbms_obfuscation_toolkit.DES3Decrypt
  (
    input_string => var_encrypted_string,
    key_string => var_des3_key_string,
    decrypted_string => var_decrypted_string
  );
  return(var_decrypted_string);
end;
/
```

### Example 7

#### Test The Triggers

To test each trigger, perform an insert, update and delete statement on the data table. An insert statement will activate the before insert trigger which will encrypt the data and place the key to that data in the key table. Example 8 shows an insert statement into the `data_table` and the results. Note the length of the data to be encrypted is rounded up to 16 to be evenly divisible by eight.

```
SQL> insert into data_table
  2 (pk_col1, hex_encrypted_passwd) values
  3 (1, '123456789');

1 row created.

SQL> select pk_col1, hex_encrypted_passwd from data_table;

PK_COL1 HEX_ENCRYPTED_PASSWD
-----
1 48537ED57F638A3165B86ED182C91FA2

SQL> select * from key_table;

PK_COL1 DES3_SEED
-----
1 2, ,40-6m2, ,40-6m2, ,40-6m
```

### Example 8

Use the `get_passwd` function to retrieve the data. Pass the primary key to the function within a SQL statement. In Example 9, a SQL statement is shown returning the row previously inserted. Also, note the length of the data itself is still 9 bytes. The spaces were encrypted as part of the data and removed as part of the `get_passwd` function.

```
SQL> select get_passwd(pk_col1), length(get_passwd(pk_col1))
  2 from data_table;

GET_SSN(PK_COL1) LENGTH(GET_SSN(PK_COL1))
-----
123456789          9
```

### Example 9

Test the update trigger by coding an update statement on the `data_table`. When the update statement is executed, the update trigger will fire and use the existing DES3 key, encrypt the new value and update the table. In Example 10, an update statement is shown and tested. Note the encrypted hex value has changed.

```
SQL> update data_table
  2 set hex_encrypted_passwd = '987654321';

1 row updated.

SQL> select * from data_table;

PK_COL1 HEX_ENCRYPTED_PASSWD
-----
1 DC33B1591E827BC27F3A65BF10B6E3BF

SQL> select get_passwd(pk_col1), length(get_passwd(pk_col1))
  2 from data_table;

GET_SSN(PK_COL1) LENGTH(GET_SSN(PK_COL1))
-----
987654321          9
```

### Example 10

Lastly, test a delete statement. A delete statement will execute the delete trigger which will remove the corresponding row from `key_table`. The obfuscation package is not necessary for this step, as the data is only being removed, not encrypted or decrypted. In Example 11, a delete statement is executed and the rows removed are verified.

```
SQL> delete data_table
  2 where pk_col1 = 1;

1 row deleted.

SQL> select count(*) from data_table;

COUNT(*)
-----
0

SQL> select count(*) from key_table;

COUNT(*)
-----
0
```

### Example 11

## Key Management

Management of the `key_table` and the encryption keys is essential for this application to function properly. When the primary keys of the `data_table` and `key_table` are not correct or are out of sync, all encrypted data can be corrupted. Backing up and storing the `key_table` is fundamental. Much thought must be given to protecting the data within the `key_table`.

The most simplistic approach to protecting the `key_table` from data corruption due to inconsistent primary keys is to export the table. Oracle provides the `exp` utility to extract whole schemas or single tables in this case. Also, `rman` can be used to perform hot or cold backups of the entire database or pieces of it. A sound backup strategy is vital when depending on systemic approaches to data integrity.

## Theory, Design and Practicality

Multiple encrypted keys can seem like overkill in regards to protecting sensitive or confidential data. However, the subject can be approached from two points of view: 1) the customer, or 2) the system provider. If you are the customer or the originator of that data, implied responsibility and assurances of the sanctity of that data is paramount. However, if you are the system provider, you look at overhead, administration resources and maintenance. To assure the credibility of the encryption strategy for each customer, individual encryption keys are assigned to each row of applicable data.

If the encryption key was intercepted over transmission lines and there were only one key, the entire table could be compromised. However, with the extra layer of security provided by multiple keys, only one row, at most, would be in danger if a key were compromised by some means. In a highly security-conscious environment, there would be one encryption `key_table` for every table where sensitive data were to be stored and encrypted. It is up to the application planners to balance customer satisfaction with a sound design.

## Protect the Code

For additional security of encrypted data, the code itself can be concealed. The source code to most database packages, procedures, triggers and functions is easily obtainable via database views. To protect source code, Oracle provides a `wrap` utility to encode stored PL/SQL objects into hexadecimal characters so that it cannot be easily viewed. The `wrap` utility can be used to protect the `get_passwd` function from those who may not have permission to read the code via database views. An example of how to encode the `get_passwd.sql` file using the `wrap` utility is in Example 12.

```
$ wrap iname=get_passwd.sql oname=get_passwd.plb
```

### Example 12

The output of the command above, `get_passwd.plb`, is an encoded, platform independent file that can be installed into the database in place of the original SQL script. The views `DBA_SOURCE`, `ALL_SOURCE` and `USER_SOURCE` display wrapped code, not the original source code.

## Summary

To encrypt and decrypt data within a database, several steps must be taken. First, a column of data must be identified worthy of the overhead necessary to encrypt and decrypt data. An additional `key_table` needs to be created and a backup and recovery strategy created for this table. There must be a

one-to-one relationship between the `data_table` and the `key_table`. Code must be created and tested to deal with all insert, update and delete statements being run against the table `data_table`. Lastly, database functions need to be created to encrypt and decrypt data within the identified `data_table`.

After implementing this application the `hex_encrypted_passwd` column within the `data_table` is encrypted and thus protected from casual data browsers. However, after using the `get_passwd` function, the `key_table` can be joined with the `data_table` and the `hex_encrypted_passwd` column can be appropriately decrypted.

The DES3 encryption algorithm is the stronger of the two algorithms available at this time. In this example, we used a 24-byte key to encrypt, decrypt, store and display data in a fictitious two table application. If data security is an issue for an application, data encryption of a column may be necessary to further assure customers and end-users that their data is protected.

## Conclusion

There are many ways to protect data and database functions from those that should not view or use them. Preventing the accidental user from tripping over protected data may be accomplished by applying this solution to your application. However, confidential or sensitive data needs to be protected from those who purposefully dig for information as well. Within this application the user-defined `get_passwd` function needs to be protected. This function decrypts and displays the data you are trying to obfuscate. Some common security access rules should be applied specifically to `get_passwd`. An example of how to protect the `get_passwd` function would be revoking execute privileges from public and any commonly accessed roles. Another way of protecting this function from prying eyes would be not to include `get_passwd` in any function based indexes or custom views. Lastly, as much as security through obscurity is not a finite solution to protecting data, changing the name or adding unused but required parameters to the function may assist in protecting it from unauthorized use. It is up to the organization to decide to what lengths they will go to protect their data and what costs will they endure to keep it secure.



### About the Author

#### Contact Information

Rick Boesen  
rick.boesen@oracle.com  
Principal Consultant  
Database Administrator  
Oracle Corporation  
North American Consulting

#### Current Assignment

Defense Logistics  
Information Service  
Battle Creek, Michigan