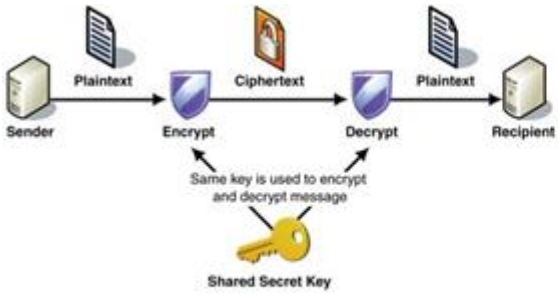
Database Security – master, 2nd year
Laboratory 1

Encryption and decryption of data in a database

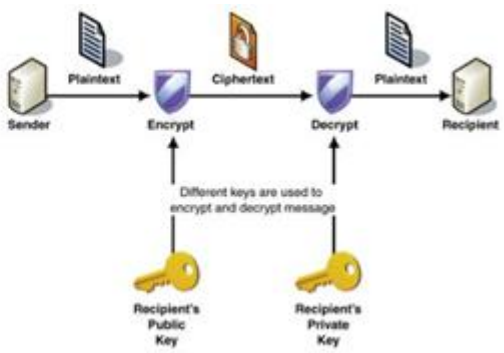
Keywords:	
• Encryption / decryption • Symmetric / asymmetric encryption algorithm • Padding • Chaining	• The package DBMS_CRYPTO • TDE (Transparent Data Encryption) • Confidentiality, integrity • Hashing

1. Introduction

- Data encryption represents a way to protect both current and archived data, conferring them confidentiality.
- Examples of data that need encryption? Passwords, PIN codes, security numbers etc.
- The base elements of an encryption system are:
  - The encryption algorithm – the method used to modify the value;
  - The encryption key – on whose safety depends the strength (lack of vulnerability) of the encrypted data.



Symmetric encryption



Asymmetric encryption

Sursa: http://msdn.microsoft.com/en-us/library/ff650720.aspx

- *Oracle* system supports both symmetric and asymmetric encryption algorithms:
  - ***Symmetric algorithms*** (that use the same key for data encryption and their decryption) – for the encryption of the stored data;
  - ***Asymmetric algorithms*** (in which the receiver generates 2 keys: a private key that it will use for decryption and a public key that it sends to the issuer in order to encrypt the message) – for the database users' authentication and for the communication

between the client and the database. The asymmetric encryption algorithms are a part of the *Oracle Advanced Security* option (which is payable).

We note that Oracle uses symmetric encryption algorithms for the encryption of the stored data.

1.1 Recap: symmetric encryption algorithms

We briefly recall some symmetric encryption algorithms, which are also available in *Oracle*:

- **DES** (*Data Encryption Standard*)
  DES encrypts a plain text block of 64 bits in an encrypted text of 64 bits, too, using 56 bits out of a key of 64 bits.
- **3-DES** (*Triple Data Encryption Standard*)
  It is based on the formula $c = DES_{k3}(DES^{-1}_{k2}(DES_{k1}(m)))$ in the variant 3DES-ede
  or the formula $c = DES_{k3}(DES_{k2}(DES_{k1}(m)))$ in the variant 3DES-eee
  where $k1$, $k2$, $k3$ are keys of 56 bits (thus, using together 168 bits out of a required key of 192 bits), DES_k is the DES encryption using the key k, DES^{-1}_k is the DES decryption using the key k, and m is the original block of 64 bits.
  If $k1 = k2$ or $k2 = k3$ or $k1 = k2 = k3$, then 3DES becomes DES.
- **AES** (*Advanced Encryption Standard*)
  AES encrypts a plain text block of 128 bits as an encrypted text of 128 bits, too, using a key of 128, 192 or 256 bits.

We remark that the encryption algorithms above work with blocks whose size is fixed and established (64 bits=8 bytes for DES and 3-DES, respectively 128 bits=16 bytes for AES). A fragment of plain data will be segmented in blocks of the size required by the algorithm. Then, the algorithm will be applied on each obtained block.

1.2 Padding and chaining

- How can be treated the case in which the plain data size is NOT a multiple of the required block size?
  → The **padding technique** fills the last segment of the plain data fragment up to the block size.
- We recall that we encountered the functions LPAD, RPAD within the string functions:

```
SELECT LPAD('A',3,'#'), RPAD('B',3,'@') FROM DUAL;
```

LPAD('A',3,'#')	RPAD('B',3,'@')
##A	B@@

- In view of the encryption, one can choose the padding with zeros or the padding schema PKCS#5.
- Let *block_size* be the size of the block, as required by the algorithm

data_size the total size (in bytes) of the plain data fragment.

The padding schema PKCS#5 calculates, for the last segment in the fragment, the difference:

$d = block\_size - (data\_size \text{ MOD } block\_size)$

and fills each missing byte with the hexadecimal value 0x0*d*.

Example: *block_size* = 8, *data_size* =100 => *d*=8 – (100 MOD 8) = 8 – 4 = 4

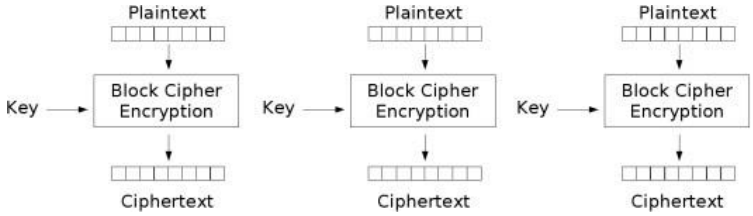
The last segment, the one having 4 bytes, will be padded with 0x04040404 (meaning 00000100 00000100 00000100 00000100)

We note that the padding is applied before encryption and is removed after decryption.

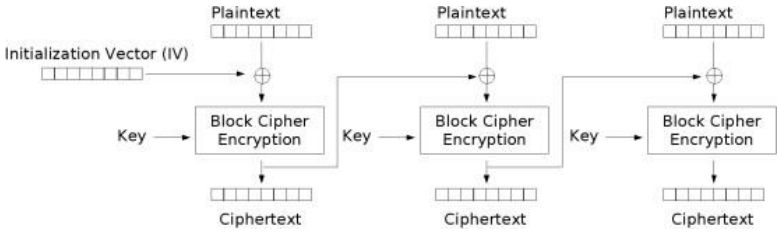
- How can be treated the case in which the plain data fragment consists of several blocks to be encrypted?

  → The **chaining technique** establishes if the encryption of a block depends or not on the encryption of the previous blocks in the plain fragment.

- The following variants of chaining are available in *Oracle*:
  - *Electronic Code Book* (CHAIN_ECB) – each block is encrypted independently of the other blocks in the fragment. The drawback is that one can identify repetitive patterns in the fragment.

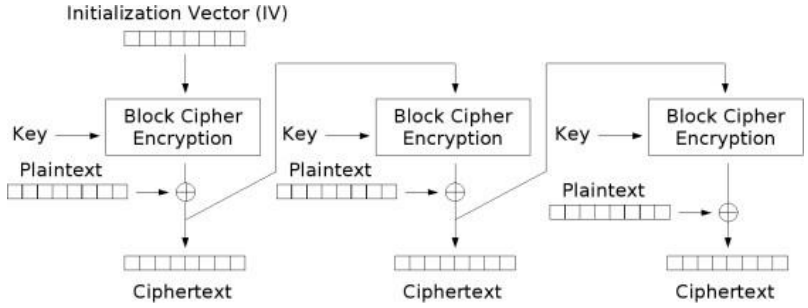


  - *Cipher Block Chaining* (CHAIN_CBC) – each block, before encryption, is applied the XOR operation with a vector. An initialization vector is used for the first block in the fragment. For the other blocks, the result of the previous block's encryption will be used as the vector for the XOR operation.



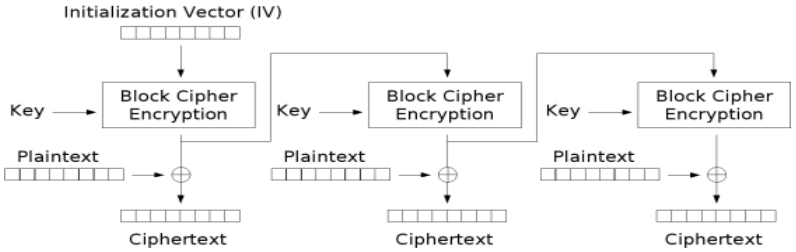
Sursa: http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

➢ *Cipher Feedback* (CHAIN_CFB) :



Sursa: http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

➢ *Output Feedback* (CHAIN_OFB) :



Sursa: http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

2. Data encryption by PL/SQL

- We have the following packages at our disposal:
  - DBMS_CRYPTO (issued in the Oracle 10g version)
  - Prior to Oracle 10g: DBMS_OBFUSCATION_TOOLKIT (*deprecated*)

Remark: Execute the following command while being logged in as *SYS AS SYSDBA* in order to grant the user *username* the privilege *execute* on the package DBMS_CRYPTO:

```
GRANT EXECUTE ON dbms_crypto TO username;
```

If this right is missing, you will get the following error:

2.1 Syntax

- **ENCRYPTION:**

```
dbms_crypto.encrypt(
  plain_fragment IN RAW,
  operation_mode IN PLS_INTEGER,
  key IN RAW,
  initialization_vector IN RAW DEFAULT NULL)
RETURN RAW;
```

- **DECRYPTION**:

```
dbms_crypto.decrypt(
  encrypted_fragment IN RAW,
  operation_mode IN PLS_INTEGER,
  key IN RAW,
```

```
  initialization_vector IN RAW DEFAULT NULL)
RETURN RAW;
```

where:
operation_mode = Algorithm_code + Padding_code + Chaining_code

DBMS_CRYPTO.ENCRYPT_DES	DBMS_CRYPTO.PAD_PCKS5	DBMS_CRYPTO.CHAIN_CBC
DBMS_CRYPTO.ENCRYPT_3DES_2KEY	DBMS_CRYPTO.PAD_ZERO	DBMS_CRYPTO.CHAIN_CFB
DBMS_CRYPTO.ENCRYPT_3DES	DBMS_CRYPTO.PAD_NONE	DBMS_CRYPTO.CHAIN_ECB
DBMS_CRYPTO.ENCRYPT_AES128		DBMS_CRYPTO.CHAIN_OFB
DBMS_CRYPTO.ENCRYPT_AES192		
DBMS_CRYPTO.ENCRYPT_AES256		
DBMS_CRYPTO.ENCRYPT_RC4		

2.2 Other useful functions

- Conversion VARCHAR2 → RAW:

```
utl_i18n.string_to_raw(
  data IN VARCHAR2 CHARACTER SET ANY_CS,
  dst_charset IN VARCHAR2 DEFAULT NULL)
RETURN RAW;
```

where *dst_charset* = 'AL32UTF8'

Alternatively, if the database's character set is *AL32UTF8*, the following function can be used:

```
utl_raw.cast_to_raw(string IN VARCHAR2) RETURN RAW;
```

- Conversion RAW → VARCHAR2 with characters:

```
utl_i18n.raw_to_char(
  data IN RAW,
  src_charset IN VARCHAR2 DEFAULT NULL)
RETURN VARCHAR2;
```

unde *src_charset* = 'AL32UTF8'

- Conversion RAW ↔ VARCHAR2 containing its hexadecimal representation:

```
RAWTOHEX (data IN RAW) RETURN VARCHAR2;
HEXTORAW (data IN VARCHAR2) RETURN RAW;
```

3. Encryption keys' management issues

- It is difficult for the database's users to generate manually efficient encryption keys, having the length required by the algorithm.
- Regarding the manual provision of the encryption key as a string (which will be then converted as a RAW), the string's length is calculated so:

$$L\_string = L\_key\_in\_bytes / 8$$

Example: For ENCRYPT_AES128, the key's length is 128 bits => The string's length will be: *L_string* = 128/8 = 16
Providing the key '1234567890123456' will be accepted as it has 16 characters, while the key'1234' will raise the exception "*key length too short*". Similarly for the rest of the algorithms based on the following table:

Constant	Effective key length
ENCRYPT_DES	56
ENCRYPT_3DES	156
ENCRYPT_AES128	128
ENCRYPT_AES192	192
ENCRYPT_AES256	256

- The alternative is represented by the automatic key generation, having the required length:

```
key RAW (nb_bytes);
key:= DBMS_CRYPTO.randombytes (nb_bytes);
```

- The function *randombytes* implements the algorithm *Pseudo-Random Number Generator*.
- Once they are obtained, the secret keys must be kept safe, as their disclosure may compromise the security of the encrypted data.
- Options:

| ---- a key at the database level |--- stored in the database (in a special table)
| |--- stored in an external file
|
| ---- a key at row level --- stored in the database (in a special table)
|
|-----a combination between the previous ones – there is a master key at the database level and a key at row level (a key for each row). A hybrid key will be used both at encryption and decryption: *hybrid key* = *master key* XOR *row key* (PL/SQL function *UTL_RAW.bit_xor*)

We note that the hybrid key's choice is the most efficient between the above options:

- *If the whole database is stolen, the data cannot be decrypted if the master key is stored in the file system;*
- *If the master key and a row-level key are disclosed, the other rows remain protected.*

4. *Transparent Data Encryption* (TDE)

- TDE is a feature offered by the system starting with the version *Oracle 10g*, that allows to declare encrypted columns at the level of a database table.
- When inserting data in the encrypted columns, Oracle encrypts data automatically and stores their encrypted values in the database. Any SELECT operation will automatically

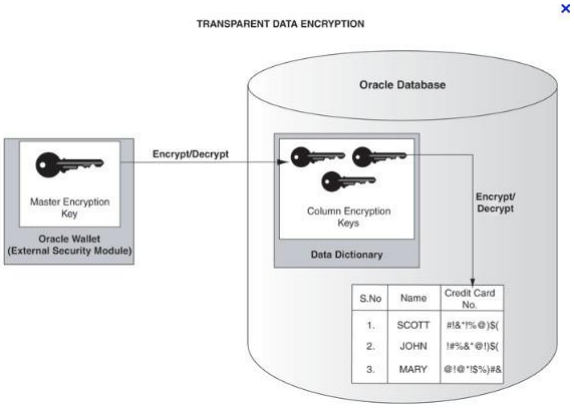
decrypt data in the database. We note that *Transparent Data Encryption does not differentiate between users as long as they are authorized, offering to all of them the decrypted value of data.*

- Not every column can be declared as 'encrypted'; the columns in the foreign key cannot be encrypted TDE.

Example: We define the encrypted columns *card_series* and *balance* in the table *ACCOUNT* (*id_account#, card_series, owner, balance*):

```
ALTER TABLE account MODIFY (card_series ENCRYPT USING 'AES128');
ALTER TABLE account MODIFY (balance ENCRYPT USING 'AES128');
```

- For all the encrypted columns in a table *T* the same private key *Key_T* is used. If we have the tables *T*1, *T*2, ..., *Tn* each of them containing different encrypted columns, it implies that there are *n* private keys *Key_T*1, *Key_T*2, ..., *Key_Tn* .
- Each private key *Key_Tj* , *j* = 1, ..., *n*, is encrypted in turn with a master key *Key_Master* and its encryption's result is stored in the data dictionary.
- The master key is stored externally to the database in a wallet. *Thus, Transparent Data Encryption prevents the data decryption in the case the database is stolen.*



Sursa: http://docs.oracle.com/cd/B28359_01/network.111/b28530/asotrans.htm

- The steps:

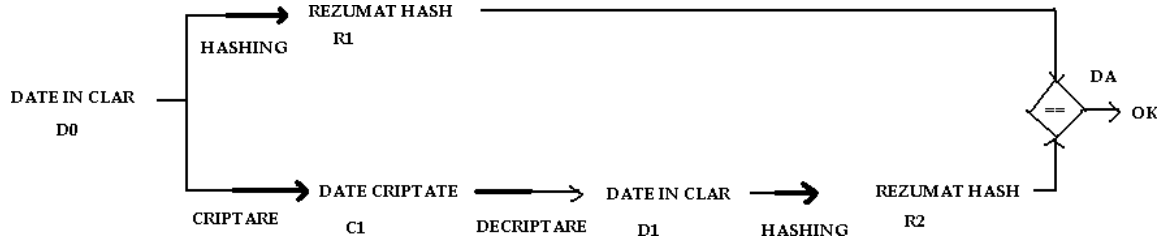
Automatic encryption	***Automatic decryption***
Obtaining the master key *Key_Master* from the external wallet;. Decrypting the private key *Key_Tk* using the master key; Encrypting data to be inserted using the private key *Key_Tk*; Storing encrypted data in the table's columns.	Obtaining the master key *Key_Master* from the external wallet;. Decrypting the private key *Key_Tk* using the master key Decrypting data using the private key *Key_Tk*; Return the result.

More details – in *Oracle* documentation:
https://docs.oracle.com/database/121/ASOAG/introduction-to-transparent-data-encryption.htm#ASOAG10117.

5. Ensuring encrypted data integrity

Data encryption ensures data's confidentiality but does not guarantee its integrity. Thus, encrypted data can be modified.

- The hashing technique is used in order to prevent this danger, in addition to encrypting original data. *Hashing* has two important properties:
  - It does not allow the decryption of the original value;
  - It is deterministic, i.e., applied repetitively on the same data it generates the same result.



- *Oracle* allows *hashing* algorithms: MD5 and SHA-1.
- Syntax:

```
DBMS_CRYPTO.Hash (
  Original_string IN RAW,
  Operation_mode IN PLS_INTEGER)
RETURN RAW;
```

where *operation_mode* ∈ {*DBMS_CRYPTO.HASH_MD5*, *DBMS_CRYPTO.HASH_SH1*}

6. Exercises

1. Create the procedure ENCRYPTION1 which encrypts a string received as a parameter using the algorithm DES, the key '12345678', the padding with zeros and the chaining method ECB. Call the procedure for the string 'Plain Text' from an anonymous PL/SQL block.

2. Create the procedure DECRYPTION1 which decrypts a string received as a parameter using the algorithm DES, the key '12345678', the padding with zeros and the chaining method ECB. Call the procedure from the same anonymous PL/SQL block from the previous exercise.

3. Salary data (*id_employee* and *salary*) of the table *EMPLOYEES* will be encrypted (AES-128, PAD_PKCS5, CHAIN_CBC) and stored in the table *EMPLOYEES_CRYPT* as follows:

- **The odd rows (1, 3,…) will be encrypted with the key *ODD_KEY***
- **The even rows (2, 4,…) will be encrypted with the key *EVEN_KEY*.**

The two keys will be generated automatically and stored in the database.

Create the sequence *SEQ_ID_KEY* and the table *KEYS_TABLE* (*id_key#*, *key*, *table*).

Create the procedure *ENCRYPT_EVEN_ODD* without parameters. Within the procedure, automatically generate 2 private keys with 16 bytes *EVEN_KEY* and *ODD_KEY*. The keys will be stored in the table *KEYS_TABLE*, with the primary keys generated from the sequence *SEQ_ID_KEY*.

4. Attempt to modify (*UPDATE*) the encrypted value of the first employee's salary (the employee in the first row) from the table *EMPLOYEES_CRYPT*. Set his salary to the value 0x1F4 (meaning 500 in decimal). Did the update succeed?

5. Create the procedure *DECRYPT_EVEN_ODD* without parameters, as the pair of the procedure in exercise 4. The keys saved in the table *KEYS_TABLE* will be used for decryption, in the same order (even, odd).

The decrypted data will be stored in the table *EMPLOYEES_DECRYPT*.

Compare the salaries of the first employee in the tables *EMPLOYEES* and *EMPLOYEES_DECRYPT*.

6. Create a function *HASH_MD5* that returns the hash value (MD5) for the row in the table *EMPLOYEES* corresponding to *employee_id* equal to 104. Store the result in a bind variable *hash1*. Update the salary of this employee, giving a raise of 20%.

Create a new hash of this row and store the result in the bind variable *hash2*. What can we notice?