**DBMS\_CRYPTO Package**

**(As it relates to my Oracle package named: DEMO\_SECURITY\_PKG)**

***Summary:***

The `**DBMS\_CRYPTO**` package in Oracle Database 21c provides various features for encryption and decryption, including cryptographic algorithms, padding forms, block cipher chaining modes, cryptographic hash algorithms, keyed hash (MAC) algorithms, and a cryptographic pseudo-random number generator. It supports `RAW`, `CLOB`, and `BLOB` database types. The package contains predefined cryptographic algorithms, modifiers, and cipher suites. The `VARCHAR2` datatype is not directly supported by `DBMS\_CRYPTO` and must be converted to the uniform database character set `AL32UTF8` and then to the `RAW` datatype before it can be encrypted with the `DBMS\_CRYPTO` package. The package includes both `ENCRYPT` and `DECRYPT` procedures and functions. The procedures are used to encrypt or decrypt `LOB` datatypes while the functions are used to encrypt and decrypt `RAW` datatypes. The package also includes two different types of one-way hash functions: the `HASH` function and the `MAC` function.

***Sources Referenced: (See end of document)***

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

[Table 47-1](https://docs.oracle.com/en/database/oracle/oracle-database/21/arpls/DBMS_CRYPTO.html#GUID-1C98C203-29EF-488D-A5FA-42AD4BD7718D__BJFFAJCC) summarizes the DBMS\_CRYPTO package features.

Table 47-1 DBMS\_CRYPTO Features

| **Package Feature** | **DBMS\_CRYPTO** |
| --- | --- |
| Cryptographic algorithms | DES, 3DES, AES, 3DES\_2KEY |
| Padding forms | PKCS5, zeroes |
| Block cipher chaining modes | CBC, CFB, ECB, OFB |
| Cryptographic hash algorithms | MD5, MD4, SHA-1, SHA-2 (SHA-256, SHA-384, SHA-512) |
| Keyed hash (MAC) algorithms | HMAC\_MD5, HMAC\_SH1, HMAC\_SH256, HMAC\_SH384, HMAC\_SH512 |
| Cryptographic pseudo-random number generator | RAW, NUMBER, BINARY\_INTEGER |
| Database types | RAW, CLOB, BLOB |

47.4 DBMS\_CRYPTO Datatypes

Parameters for the DBMS\_CRYPTO subprograms use these datatypes.

Table 47-3 DBMS\_CRYPTO Datatypes

| **Type** | **Description** |
| --- | --- |
| BLOB | A source or destination binary LOB |
| CLOB | A source or destination character LOB (excluding NCLOB) |
| PLS\_INTEGER | Specifies a cryptographic algorithm type (used with BLOB, CLOB, and RAW datatypes) |
| RAW | A source or destination RAW buffer |

47.5 DBMS\_CRYPTO Algorithms

The DBMS\_CRYPTO package contains predefined cryptographic algorithms, modifiers, and cipher suites.

These are shown in the following tables.

Table 47-6 DBMS\_CRYPTO Encryption Algorithms

| **Name** | **Description** |
| --- | --- |
| ENCRYPT\_DES | Data Encryption Standard. Block cipher. Uses key length of 56 bits. |
| ENCRYPT\_3DES\_2KEY | Data Encryption Standard. Block cipher. Operates on a block 3 times with 2 keys. Effective key length of 112 bits. |
| ENCRYPT\_3DES | Data Encryption Standard. Block cipher. Operates on a block 3 times. |
| **ENCRYPT\_AES128** | **Advanced Encryption Standard. Block cipher. Uses 128-bit key size.** |
| ENCRYPT\_AES192 | Advanced Encryption Standard. Block cipher. Uses 192-bit key size. |
| ENCRYPT\_AES256 | Advanced Encryption Standard. Block cipher. Uses 256-bit key size. |

**Encryption Algorithms:**

**As shown in the Table above (**Table 47-5), several encryption algorithms exist.

I use this algorithm in my encryption: **DBMS\_CRYPTO.ENCRYPT\_AES128**

***Note:***

In order to generate a suitable key for this algorithm, you should provide a key of length:

AES128 => 128 bits => 128/8 bits => 16 bytes

Key Length = 16-byte-long key

i.e. V\_KEY VARCHAR2 := ‘ABCDEFG\_12345678’

\**Note: The `DBMS\_CRYPTO.ENCRYPT\_AES128` algorithm requires a key of length 128 bits. Since most computers use 8-bit word sizes, where each byte is 8 bits long, you can calculate the required key length in bytes by dividing the number of bits by 8. In this case, 128 bits / 8 bits/byte = 16 bytes. So, a suitable key for the `DBMS\_CRYPTO.ENCRYPT\_AES128` algorithm would be a text string that is 16 characters long. For example: `ABCDEFG\_12345678`.*

**-----------------**

**For: ENCRYPT\_AES256: 256/8 = 32**

**Key Length = 32-**byte-long key **etc.**

Table 47-8 DBMS\_CRYPTO Block Cipher Chaining Modifiers

| **Name** | **Description** |
| --- | --- |
| CHAIN\_ECB | Electronic Codebook. Encrypts each plaintext block independently. |
| **CHAIN\_CBC** | **Cipher Block Chaining. Plaintext is XORed with the previous ciphertext block before it is encrypted.** |
| CHAIN\_CFB | Cipher-Feedback. Enables encrypting units of data smaller than the block size. |
| CHAIN\_OFB | Output-Feedback. Enables running a block cipher as a synchronous stream cipher. Similar to CFB, except that *n* bits of the previous output block are moved into the right-most positions of the data queue waiting to be encrypted. |
|  |  |

**Chaining:**

After dividing the data into chunks, there needs to be a way to connect them.

This process is called chaining.

The security of an encryption system depends on how the chunks are connected and encrypted.

The most common chaining format is Cipher Block Chaining (CBC), which can be selected in Oracle using a constant defined in the CHAIN\_CPC built-in package.

This is the constant I use in my code:

i.e. **DBMS\_CRYPTO.CHAIN\_CBC**

Other chaining options include Electronic Code Book format (CHAIN\_ECB), Cypher Feedback (CHAIN\_CFB), and Output Feedback (CHAIN\_OFB).

Table 47-9 DBMS\_CRYPTO Block Cipher Padding Modifiers

| **Name** | **Description** |
| --- | --- |
| **PAD\_PKCS5** | **Provides padding which complies with the PKCS #5: Password-Based Cryptography Standard** |
| PAD\_NONE | Provides option to specify no padding. Caller must ensure that blocksize is correct, else the package returns an error. |
| PAD\_ZERO | Provides padding consisting of zeroes |

**Padding:**

When data is encrypted, it is typically divided into eight-byte chunks and each chunk is encrypted separately.

However, the length of the data may not always be a multiple of eight. In such cases, additional characters are added to the last chunk to make it eight bytes long.

This process is called padding.

It is important to pad the values securely to prevent attackers from guessing the key.

One secure padding method implemented in Oracle is Public Key Cryptography System #5 (PKCS#5). This is the padding method I use in my code.

i.e. **DBMS\_CRYPTO.PAD\_PKCS5**

Other padding options include padding with zeros or no padding at all.

47.6 DBMS\_CRYPTO Restrictions

The VARCHAR2 datatype is not directly supported by DBMS\_CRYPTO. **Before you can perform cryptographic operations on data of the type VARCHAR2, you must convert it to the uniform database character set AL32UTF8, and then convert it to the RAW datatype**. After performing these conversions, you can then encrypt it with the DBMS\_CRYPTO package.

In my code I use:

i.e. **UTL\_I18N.STRING\_TO\_RAW (V\_TEXT\_2\_ENCRYPT\_IN, 'AL32UTF8')**

**Note:**

The UTL\_I18N package can be used to convert string to raw (STRING\_TO\_RAW) as well as (for decrypting) converting Raw to string (UTL\_I18N.RAW\_TO\_CHAR)

The UTL\_I18N package also has another function: CAST\_TO\_RAW.

We use the **UTL\_i18N.STRING\_2\_RAW** function instead of **UTL\_RAW.CAST\_2\_RAW** to convert VARCHAR2 data to RAW.

This is because the **ENCRYPT** function **requires the input to be in RAW format** **and in the AL32UTF8 character set**, which may not be the same as the database’s character set.

To use a **VARCHAR2** string for encryption, two conversions are necessary: first, from the database’s character set to **AL32UTF8**, and second, from **VARCHAR2** to **RAW**.

The **UTL\_i18N.STRING\_2\_RAW** function performs both of these conversions, while the **UTL\_RAW.CAST\_2\_RAW** function does not perform character set conversion.

[DBMS\_CRYPTO Operational Notes](https://docs.oracle.com/en/database/oracle/oracle-database/21/arpls/DBMS_CRYPTO.html#GUID-5FF9978D-59D1-4F4A-9A48-C3A03DC569AE) for information about the conversion rules for converting datatypes.

47.7 DBMS\_CRYPTO Exceptions

The following table lists exceptions that have been defined for DBMS\_CRYPTO.

Table 47-10 DBMS\_CRYPTO Exceptions

| **Exception** | **Code** | **Description** |
| --- | --- | --- |
| CipherSuiteInvalid | 28827 | The specified cipher suite is not defined. |
| CipherSuiteNull | 28829 | No value has been specified for the cipher suite to be used. |
| KeyNull | 28239 | The encryption key has not been specified or contains a NULL value. |
| KeyBadSize | 28234 | DES keys: Specified key size is too short. DES keys must be at least 8 bytes (64 bits).  AES keys: Specified key size is not supported. AES keys must be 128, 192, or 256 bits in length. |
| DoubleEncryption | 28233 | Source data was previously encrypted. |

When to Use Encrypt and Decrypt Procedures or Functions

This package includes both **ENCRYPT** and **DECRYPT** procedures and functions.

The procedures are used to encrypt or decrypt LOB datatypes (overloaded for CLOB and BLOB datatypes).

In contrast, the **ENCRYPT** and **DECRYPT** **functions** are used to encrypt and decrypt **RAW** datatypes. **Data of type VARCHAR2 must be converted to RAW before you can use DBMS\_CRYPTO functions** to encrypt it.

**HASHING:**

Hashing can be useful when you want to mask and/or protect data from being altered.

It is not encryption/decryption.

For example, you may use it to ensure that no financial data is being altered, or for password creation and verification.

**Cryptographic Hashing is a one-way encryption. It is not reversable**.

As an example, it can be used to compare a value stored in the database with a user supplied value. If they match, no altercations have been made.

**We will use it for Password Validation in our code.**

Table 47-4 DBMS\_CRYPTO Cryptographic Hash Functions

| **Name** | **Description** |
| --- | --- |
| HASH\_MD4 | Produces a 128-bit hash, or message digest of the input message. |
| HASH\_MD5 | Also produces a 128-bit hash, but is more complex than MD4. |
| HASH\_SH1 | Secure Hash Algorithm (SHA-1). Produces a 160-bit hash. |
| HASH\_SH256 | SHA-2, produces a 256-bit hash. |
| HASH\_SH384 | SHA-2, produces a 384-bit hash. |
| HASH\_SH512 | SHA-2, produces a 512-bit hash. |

Table 47-5 DBMS\_CRYPTO MAC (Message Authentication Code) Functions

| **Name** | **Description** |
| --- | --- |
| **HMAC\_MD5** | **Same as MD5 hash function, except it requires a secret key to verify the hash value.** |
| HMAC\_SH1 | Same as SHA hash function, except it requires a secret key to verify the hash value.  Complies with IETF RFC 2104 standard. |
| HMAC\_SH256 | Same as SHA-2 256-bit hash function, except it requires a secret key to verify the hash value. |
| HMAC\_SH384 | Same as SHA-2 384-bit hash function, except it requires a secret key to verify the hash value. |
| HMAC\_SH512 | Same as SHA-2 512-bit hash function, except it requires a secret key to verify the hash value. |

When to Use Hash or Message Authentication Code (MAC) Functions

The **DBMS\_CRYPTO** package includes two different types of one-way hash functions: the **HASH** function and the **MAC** function.

Hash functions operate on an arbitrary-length input message, and return a fixed-length hash value.

One-way hash functions work in one direction only.

It is easy to compute a hash value from an input message, but it is extremely difficult to generate an input message that hashes to a particular value.

Note that hash values should be **at least 128** bits in length to be considered secure.

You can use hash values to verify whether data has been altered. For example, before storing data, the user runs **DBMS\_CRYPTO.HASH** against the stored data to create a hash value.

On returning the stored data, the user can again run the hash function against it, using the same algorithm.

If the second hash value is identical to the first one, then the data has not been altered. Hash values are similar to "file fingerprints" and are used to ensure data integrity.

The **HASH** function included with **DBMS\_CRYPTO**, is a one-way hash function that you can use to generate a hash value from either **RAW**or **LOB** data.

The MAC function is also a one-way hash function, but with the addition of a secret key. It works the same way as the **DBMS\_CRYPTO.HASH** function, except only someone with the key can verify the hash value.

**To prevent a user from simply generating a new hash code, you can password protect the hash value. This is called “Message Authentication Code” or MAC.**

For example:

* MACs can be used to authenticate files between users.
* They can also be used by a single user to determine if her files have been altered, perhaps by a virus.

A user could compute the MAC of his files and store that value in a table.

If the user did not use a MAC function, then the virus could compute the new hash value after infection and replace the table entry.

A virus cannot do that with a MAC because the virus does not know the key.

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

In our code we will apply this technique in the **DEMO\_SECURITY\_PKG.**

In my **DEMO\_SECURITY\_PKG.CREATE\_UPDATE\_USER** function in the I have the following code:

-- Convert Security Master Key into RAW

SECURITY\_MASTER\_KEY\_RAW := UTL\_RAW.CAST\_TO\_RAW(CONVERT(SECURITY\_MASTER\_KEY,'AL32UTF8','US7ASCII'));

-- Convert User Supplied Password To Raw

RAW\_PASSWORD\_IN := UTL\_RAW.CAST\_TO\_RAW(CONVERT(V\_PASSWORD\_IN,'AL32UTF8','US7ASCII'));

-- Encrypt User Supplied Password - Add MAC

ENCRYPTED\_PSW := **DBMS\_CRYPTO.MAC**(SRC => RAW\_PASSWORD\_IN, TYP => DBMS\_CRYPTO.HMAC\_MD5, KEY => SECURITY\_MASTER\_KEY\_RAW);

***Here is a summary of what that code does:***

This **code** performs several operations to **encrypt a user-supplied password and add a Message Authentication Code (MAC) to it.**

1. The first line converts the `SECURITY\_MASTER\_KEY` from the `US7ASCII` character set to the `AL32UTF8` character set using the `CONVERT` function. Then, it converts the result to `RAW` format using the `UTL\_RAW.CAST\_TO\_RAW` function and assigns it to the `SECURITY\_MASTER\_KEY\_RAW` variable.

2. The second line converts the user-supplied password (`V\_PASSWORD\_IN`) from the `US7ASCII` character set to the `AL32UTF8` character set using the `CONVERT` function. Then, it converts the result to `RAW` format using the `UTL\_RAW.CAST\_TO\_RAW` function and assigns it to the `RAW\_PASSWORD\_IN` variable.

3. The third line uses the `DBMS\_CRYPTO.MAC` function to generate a MAC for the `RAW\_PASSWORD\_IN` variable using the `HMAC\_MD5` algorithm and the `SECURITY\_MASTER\_KEY\_RAW` as the key. The result is assigned to the `ENCRYPTED\_PSW` variable.

**UTL\_RAW package:**

The `UTL\_RAW` package in Oracle Database 21c provides SQL functions for manipulating `RAW` datatypes. This package is necessary because normal SQL functions do not operate on `RAW` data and PL/SQL does not allow overloading between a `RAW` and a `CHAR` datatype. `UTL\_RAW` also includes subprograms that convert various COBOL number formats to and from `RAW`. The package is not specific to the database environment and can be used in other environments as well.

**UTL\_I18N package:**

The `UTL\_I18N` package in Oracle Database 21c provides additional globalization functionality for applications written in PL/SQL. The package consists of several categories of services, including string conversion functions for various datatypes and functions that convert a text string to character references and vice versa. It also includes functions that map between Oracle, Java, and ISO languages and territories.

***Sources Referenced:***

**Bing Chat.**

***Bing Chat. (2023, May 29). Explanation of DBMS\_CRYPTO Package [Chat transcript]. Microsoft Corporation.***

**Oracle Documentation:**

***Oracle Corporation. (2021). DBMS\_CRYPTO - Oracle Help Center [Documentation]. Retrieved from https://docs.oracle.com/en/database/oracle/oracle-database/21/arpls/DBMS\_CRYPTO.html***