18
On-Demand Encryption of Data

You can use the DBMS_CRYPTO PL/SQL package to perform on-demand encryption of data.

- About On-Demand Encryption of Data
  To perform on-demand encryption of data, you use the DBMS_CRYPTO PL/SQL package.
- Security Problems That Encryption Does Not Solve
  While there are many good reasons to encrypt data, there are many reasons not to encrypt data.
- Data Encryption Challenges
  In cases where encryption can provide additional security, there are some associated technical challenges.
- Data Encryption Storage with the DBMS_CRYPTO Package
  The DBMS_CRYPTO package enables you to perform on-demand encryption and decryption of stored data.
- Asymmetric Key Operations with the DBMS_CRYPTO Package
  The DBMS_CRYPTO package provides four functions that enable you to perform asymmetric key operations for encryption, decryption, signing, and verification.
- Examples of Using the Data Encryption API
  Examples of using the data encryption API include using the DBMS_CRYPTO.SQL procedure, encrypting AES 256-bit data, and encrypting BLOB data.

18.1 About On-Demand Encryption of Data

To perform on-demand encryption of data, you use the DBMS_CRYPTO PL/SQL package.

This package enables you to encrypt and decrypt stored data. You can use the DBMS_CRYPTO functions and procedures with PL/SQL programs that run network communications. This package supports 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).

In most cases, you should use TDE to encrypt data. If you want to encrypt data at rest, then you should use TDE.

There are several use cases for the manual encryption of data, using the DBMS_CRYPTO PL/SQL package:

- Manual encryption enables you to encrypt data at the point of data collection, and then keep this data encrypted in all other layers in the database.
- Manual encryption is useful in cases where your database may retrieve information that had already been encrypted in another source outside the database. The DBMS_CRYPTO can use the encryption key to decrypt the data and then present it in an unencrypted format.
- Manual encryption is also useful for scenarios in which you must hash passwords, protect extremely sensitive data, and use data signatures.

Disadvantages to performing on-demand encryption of data include the following:

- Indexes will be irrelevant or can have performance issues.
- Decrypting each row can result in a performance overhead.

Related Topics

- *Oracle Database PL/SQL Packages and Types Reference*

18.2 Security Problems That Encryption Does Not Solve

While there are many good reasons to encrypt data, there are many reasons not to encrypt data.

- Principle 1: Encryption Does Not Solve Access Control Problems
  When you encrypt data, you should be aware that encryption must not interfere with how you configure access control.
- Principle 2: Encryption Does Not Protect Against a Malicious Administrator
  You can protect your databases against malicious database administrators by using other Oracle features, such as Oracle Database Vault.
- Principle 3: Encrypting Everything Does Not Make Data Secure
  A common error is to think that if encrypting some data strengthens security, then encrypting everything makes all data secure.

18.2.1 Principle 1: Encryption Does Not Solve Access Control Problems

When you encrypt data, you should be aware that encryption must not interfere with how you configure access control.

Most organizations must limit data access to users who need to see this data. For example, a human resources system may limit employees to viewing only their own employment records, while allowing managers of employees to see the employment records of subordinates. Human resource specialists may also need to see employee records for multiple employees.

Typically, you can use access control mechanisms to address security policies that limit data access to those with a need to see it. Oracle Database has provided strong, independently evaluated access control mechanisms for many years. It enables access control enforcement to a fine level of granularity through Virtual Private Database.

Because human resource records are considered sensitive information, it is tempting to think that all information should be encrypted for better security. However, encryption cannot enforce granular access control, and it may hinder data access. For example, an employee, the employee's manager, and a human resources clerk may all need to access an employee record. If all employee data is encrypted, then all three must be able to access the data in unencrypted form. Therefore, the employee, the manager and the human resources clerk would have to share the same encryption key to decrypt the data. Encryption would, therefore, not provide any additional security in the sense of better access control, and the encryption might hinder the proper or efficient functioning of the application. An additional issue is that it is difficult to securely transmit and share encryption keys among multiple users of a system.

A basic principle behind encrypting stored data is that it must not interfere with access control. For example, a user who has the `SELECT` privilege on `emp` should not be limited by the encryption mechanism from seeing all the data they are otherwise allowed to see. Similarly, there is little benefit to encrypting part of a table with one key and part of a table with another key if users need to see all encrypted data in the table. In this case, encryption adds to the overhead of decrypting the data before users can read it. If access controls are implemented well, then encryption adds little additional security within the database itself. A user who has

privileges to access data within the database has no more nor any less privileges as a result of encryption. Therefore, you should never use encryption to solve access control problems.

18.2.2 Principle 2: Encryption Does Not Protect Against a Malicious Administrator

You can protect your databases against malicious database administrators by using other Oracle features, such as Oracle Database Vault.

Some organizations, concerned that a malicious user might gain elevated (database administrator) privileges by guessing a password, like the idea of encrypting stored data to protect against this threat.

However, the correct solution to this problem is to protect the database administrator account, and to change default passwords for other privileged accounts. The easiest way to break into a database is by using a default password for a privileged account that an administrator allowed to remain unchanged. One example is `SYS`/`CHANGE_ON_INSTALL`.

While there are many destructive things a malicious user can do to a database after gaining the `DBA` privilege, encryption will not protect against many of them. Examples include corrupting or deleting data, exporting user data to the file system to email the data back to himself to run a password cracker on it, and so on.

Some organizations are concerned that database administrators, typically having all privileges, are able to see all data in the database. These organizations feel that the database administrators should administer the database, but should not be able to see the data that the database contains. Some organizations are also concerned about concentrating so much privilege in one person, and would prefer to partition the DBA function, or enforce two-person access rules.

It is tempting to think that encrypting all data (or significant amounts of data) will solve these problems, but there are better ways to protect against these threats. For example, Oracle Database supports limited partitioning of `DBA` privileges. Oracle Database provides native support for `SYSDBA` and `SYSOPER` users. `SYSDBA` has all privileges, but `SYSOPER` has a limited privilege set (such as startup and shutdown of the database).

Furthermore, you can create smaller roles encompassing several system privileges. A `jr_dba` role might not include all system privileges, but only those appropriate to a junior database administrator (such as `CREATE TABLE`, `CREATE USER`, and so on).

Oracle Database also enables auditing the actions taken by `SYS` (or `SYS`-privileged users) and storing that audit trail in a secure operating system location. Using this model, a separate auditor who has root privileges on the operating system can audit all actions by `SYS`, enabling the auditor to hold all database administrators accountable for their actions.

You can also fine-tune the access and control that database administrators have by using Oracle Database Vault.

The database administrator function is a trusted position. Even organizations with the most sensitive data, such as intelligence agencies, do not typically partition the database administrator function. Instead, they manage their database administrators strongly, because it is a position of trust. Periodic auditing can help to uncover inappropriate activities.

Encryption of stored data must not interfere with the administration of the database, because otherwise, larger security issues can result. For example, if by encrypting data you corrupt the data, then you create a security problem, the data itself cannot be interpreted, and it may not be recoverable.

You can use encryption to limit the ability of a database administrator or other privileged user to see data in the database. However, it is not a substitute for managing the database administrator privileges properly, or for controlling the use of powerful system privileges. If untrustworthy users have significant privileges, then they can pose multiple threats to an organization, some of them far more significant than viewing unencrypted credit card numbers.

Related Topics

- *Oracle Database Vault Administrator's Guide*

18.2.3 Principle 3: Encrypting Everything Does Not Make Data Secure

A common error is to think that if encrypting some data strengthens security, then encrypting everything makes all data secure.

As the discussion of the previous two principles illustrates, encryption does not address access control issues well, and it is important that encryption not interfere with normal access controls. Furthermore, encrypting an entire production database means that all data must be decrypted to be read, updated, or deleted. Encryption is inherently a performance-intensive operation; encrypting all data will significantly affect performance.

Availability is a key aspect of security. If encrypting data makes data unavailable, or adversely affects availability by reducing performance, then encrypting everything will create a new security problem. Availability is also adversely affected by the database being inaccessible when encryption keys are changed, as good security practices require on a regular basis. When the keys are to be changed, the database is inaccessible while data is decrypted and reencrypted with a new key or keys.

18.3 Data Encryption Challenges

In cases where encryption can provide additional security, there are some associated technical challenges.

- Encrypted Indexed Data
  Special difficulties arise when encrypted data is indexed.
- Generated Encryption Keys
  Encrypted data is only as secure as the key used for encrypting it.
- Transmitted Encryption Keys
  If the encryption key is to be passed by the application to the database, then you must encrypt it.
- Storing Encryption Keys
  You can store encryption keys in the database or on an operating system.
- Importance of Changing Encryption Keys
  Prudent security practice dictates that you periodically change encryption keys.
- Encryption of Binary Large Objects
  Certain data types require more work to encrypt.

18.3.1 Encrypted Indexed Data

Special difficulties arise when encrypted data is indexed.

For example, suppose a company uses a national identity number, such as the U.S. Social Security number (SSN), as the employee number for its employees. The company considers employee numbers to be sensitive data, and, therefore, wants to encrypt data in the

employee_number column of the employees table. Because employee_number contains unique values, the database designers want to have an index on it for better performance.

However, if DBMS_CRYPTO (or another mechanism) is used to encrypt data in a column, then an index on that column will also contain encrypted values. Although an index can be used for equality checking (for example, SELECT * FROM emp WHERE employee_number = '987654321'), if the index on that column contains encrypted values, then the index is essentially unusable for any other purpose. You should not perform on-demand encryption of indexed data.

Oracle recommends that you do not use national identity numbers as unique IDs. Instead, use the CREATE SEQUENCE statement to generate unique identity numbers. Reasons to avoid using national identity numbers are as follows:

- There are privacy issues associated with overuse of national identity numbers (for example, identity theft).
- Sometimes national identity numbers can have duplicates, as with U.S. Social Security numbers.

18.3.2 Generated Encryption Keys

Encrypted data is only as secure as the key used for encrypting it.

An encryption key must be securely generated using secure cryptographic key generation. Oracle Database provides support for secure random number generation, with the RANDOMBYTES function of DBMS_CRYPTO. DBMS_CRYPTO calls the secure random number generator (RNG) previously certified by RSA Security.

> **Note:**
>
> Do not use the DBMS_RANDOM package. The DBMS_RANDOM package generates pseudo-random numbers, which, as Randomness Recommendations for Security (RFC-1750) states that using pseudo-random processes to generate secret quantities can result in pseudo-security.

Be sure to provide the correct number of bytes when you encrypt a key value. For example, you must provide a 16-byte key for the ENCRYPT_AES128 encryption algorithm.

18.3.3 Transmitted Encryption Keys

If the encryption key is to be passed by the application to the database, then you must encrypt it.

Otherwise, an intruder could get access to the key as it is being transmitted. Network data encryption protects all data in transit from modification or interception, including cryptographic keys.

Related Topics

- Configuring Oracle Database Native Network Encryption and Data Integrity
  You can configure native Oracle Net Services data encryption and data integrity for both servers and clients.

18.3.4 Storing Encryption Keys

You can store encryption keys in the database or on an operating system.

- About Storing Encryption Keys
  Storing encryption keys is one of the most important, yet difficult, aspects of encryption.
- Storage of Encryption Keys in the Database
  Storing encryption keys in the database does not always prevent a database administrator from accessing encrypted data.
- Storage of Encryption Keys in the Operating System
  When you store encryption keys in an operating system flat file, you can make callouts from PL/SQL to retrieve these encryption keys.
- Users Managing Their Own Encryption Keys
  Having the user supply the key assumes the user will be responsible with the key.
- Manual Encryption with Transparent Database Encryption and Tablespace Encryption
  Transparent database encryption and tablespace encryption provide secure encryption with automatic key management for the encrypted tables and tablespaces.

18.3.4.1 About Storing Encryption Keys

Storing encryption keys is one of the most important, yet difficult, aspects of encryption.

To recover data encrypted with a symmetric key, the key must be accessible to an authorized application or user seeking to decrypt the data. At the same time, the key must be inaccessible to someone who is maliciously trying to access encrypted data that the malicious person is not supposed to see.

18.3.4.2 Storage of Encryption Keys in the Database

Storing encryption keys in the database does not always prevent a database administrator from accessing encrypted data.

An all-privileged database administrator could still access tables containing encryption keys. However, it can often provide good security against the casual curious user or against someone compromising the database file on the operating system.

As a trivial example, suppose you create a table (`EMP`) that contains employee data. You want to encrypt the employee Social Security number (SSN) stored in one of the columns. You could encrypt employee SSN using a key that is stored in a separate column. However, anyone with `SELECT` access on the entire table could retrieve the encryption key and decrypt the matching SSN.

While this encryption scheme seems easily defeated, with a little more effort you can create a solution that is much harder to break. For example, you could encrypt the SSN using a technique that performs some additional data transformation on the `employee_number` before using it to encrypt the SSN. This technique might be as simple as using an `XOR` operation on the `employee_number` and the birth date of the employee to determine the validity of the values.

As additional protection, PL/SQL source code performing encryption can be wrapped, (using the `WRAP` utility) which obfuscates (scrambles) the code. The `WRAP` utility processes an input SQL file and obfuscates the PL/SQL units in it. For example, the following command uses the `keymanage.sql` file as the input:

```
wrap iname=/mydir/keymanage.sql
```

A developer can subsequently have a function in the package call the `DBMS_CRYPTO` package calls with the key contained in the wrapped package.

Oracle Database enables you to obfuscate dynamically generated PL/SQL code. The `DBMS_DDL` package contains two subprograms that allow you to obfuscate dynamically generated PL/SQL program units. For example, the following block uses the `DBMS_DDL.CREATE_WRAPPED` procedure to wrap dynamically generated PL/SQL code.

```
BEGIN
......
SYS.DBMS_DDL.CREATE_WRAPPED(function_returning_PLSQL_code());
......
END;
```

While wrapping is not unbreakable, it makes it harder for an intruder to get access to the encryption key. Even in cases where a different key is supplied for each encrypted data value, you should not embed the key value within a package. Instead, wrap the package that performs the key management (that is, data transformation or padding).

An alternative to wrapping the data is to have a separate table in which to store the encryption key and to envelope the call to the keys table with a procedure. The key table can be joined to the data table using a primary key to foreign key relationship. For example, `employee_number` is the primary key in the `employees` table that stores employee information and the encrypted SSN. The `employee_number` column is a foreign key to the `ssn_keys` table that stores the encryption keys for the employee SSN. The key stored in the `ssn_keys` table can also be transformed before use (by using an `XOR` operation), so the key itself is not stored unencrypted. If you wrap the procedure, then that can hide the way in which the keys are transformed before use.

The strengths of this approach are:

- Users who have direct table access cannot see the sensitive data unencrypted, nor can they retrieve the keys to decrypt the data.
- Access to decrypted data can be controlled through a procedure that selects the encrypted data, retrieves the decryption key from the key table, and transforms it before it can be used to decrypt the data.
- The data transformation algorithm is hidden from casual snooping by wrapping the procedure, which obfuscates the procedure code.
- `SELECT` access to both the data table and the keys table does not guarantee that the user with this access can decrypt the data, because the key is transformed before use.

The weakness to this approach is that a user who has `SELECT` access to both the key table and the data table, and who can derive the key transformation algorithm, can break the encryption scheme.

The preceding approach is not infallible, but it is adequate to protect against easy retrieval of sensitive information stored in clear text.

Related Topics

- *Oracle Database PL/SQL Language Reference*

18.3.4.3 Storage of Encryption Keys in the Operating System

When you store encryption keys in an operating system flat file, you can make callouts from PL/SQL to retrieve these encryption keys.

However, if you store keys in the operating system and make callouts to it, then your data is only as secure as the protection on the operating system.

If your primary security concern is that the database can be broken into from the operating system, then storing the keys in the operating system makes it easier for an intruder to retrieve encrypted data than storing the keys in the database itself.

18.3.4.4 Users Managing Their Own Encryption Keys

Having the user supply the key assumes the user will be responsible with the key.

Considering that 40 percent of help desk calls are from users who have forgotten their passwords, you can see the risks of having users manage encryption keys. In all likelihood, users will either forget an encryption key, or write the key down, which then creates a security weakness. If a user forgets an encryption key or leaves the company, then your data is not recoverable.

If you do decide to have user-supplied or user-managed keys, then you need to ensure you are using native network encryption so that the key is not passed from the client to the server in the clear. You also must develop key archive mechanisms, which is also a difficult security problem. Key archives and backdoors create the security weaknesses that encryption is attempting to solve.

18.3.4.5 Manual Encryption with Transparent Database Encryption and Tablespace Encryption

Transparent database encryption and tablespace encryption provide secure encryption with automatic key management for the encrypted tables and tablespaces.

If the application requires protection of sensitive column data stored on the media, then these two types of encryption are a simple and fast way of achieving this.

Related Topics

- *Oracle Database Advanced Security Guide*

18.3.5 Importance of Changing Encryption Keys

Prudent security practice dictates that you periodically change encryption keys.

For stored data, this requires periodically unencrypting the data, and then reencrypting it with another well-chosen key.

You would most likely change the encryption key while the data is not being accessed, which creates another challenge. This is especially true for a Web-based application encrypting credit card numbers, because you do not want to shut down the entire application while you switch encryption keys.

18.3.6 Encryption of Binary Large Objects

Certain data types require more work to encrypt.

For example, Oracle Database supports storage of binary large objects (BLOBs), which stores very large objects (for example, multiple gigabytes) in the database. A BLOB can be either stored internally as a column, or stored in an external file.

Related Topics

- Example: Encryption and Decryption Procedures for BLOB Data
  You can encrypt BLOB data.

18.4 Data Encryption Storage with the DBMS_CRYPTO Package

The `DBMS_CRYPTO` package enables you to perform on-demand encryption and decryption of stored data.

While encryption is not the ideal solution for addressing several security threats, it is clear that selectively encrypting sensitive data before storage in the database does improve security. Examples of such data could include credit card numbers and national identity numbers.

The `DBMS_CRYPTO` package enables encryption and decryption for common Oracle Database data types, including `RAW` and large objects (LOBs), such as images and sound. Specifically, it supports BLOBs and CLOBs. In addition, it provides Globalization Support for encrypting data across different database character sets.

The following cryptographic algorithms are supported:

- AES, DES (deprecated), 3DES (deprecated), PBE_MD5DES (deprecated), 3DES_2KEY (deprecated), RC4 (deprecated), SM4
- Cryptographic hash algorithms MD5(deprecated), SHA1(deprecated), SHA2, SHA3, SM3, SHAKE
- Keyed hash (MAC) algorithms MD5 (deprecated), SHA1 (deprecated), SHA2, SHA3
- Public Key Encryption Algorithm RSA_PKCS1_OAEP, RSA_PKCS1_OAEP_SHA2, SM2
- Sign and verify algorithms SHA1-RSA, SHA2-RSA, SHA3-RSA, SHA2-ECDSA, SHA3-ECDSA, SM3-SM2

Block cipher modifiers are also provided with `DBMS_CRYPTO`. You can choose from several padding options, including Public Key Cryptographic Standard (PKCS) #5, and from four block cipher chaining modes, including Galois/Counter Mode (GCM). Padding must be done in multiples of eight bytes.

Note:

- DES is no longer recommended by the National Institute of Standards and Technology (NIST).
- Usage of SHA-1 is more secure than MD5. (MD5 has been deprecated starting in Oracle Database 21c.)

  Starting with Oracle Database 21c, older encryption and hashing algorithms are deprecated. Deprecated algorithms include MD4, MD5, DES, 3DES, and RC4-related algorithms. Removing older, less secure cryptography algorithms prevents accidental use of these APIs. To meet your security requirements, Oracle recommends that you use more modern cryptography algorithms such as AES.

  Starting with Oracle Database 21c, older encryption and hashing algorithms are deprecated.
  As a consequence of this deprecation, Oracle recommends that you review your network encryption configuration to see if you have specified use of any of the deprecated algorithms. If any are found, then switch to using a more modern cipher, such as AES. See Configuring Oracle Database Native Network Encryption and Data Integrity for more information.
- Usage of SHA-2 is more secure than SHA-1.
- Keyed MD5 is not vulnerable.

The following table summarizes the `DBMS_CRYPTO` package features.

Table 18-1 DBMS_CRYPTO Package Feature Summary

Feature	DBMS_CRYPTO Supported Functionality
`HASH`	`DBMS_CRYPTO` supported algorithms
`HMAC`	MD5 (deprecated), SHA1 (deprecated), SHA2, SHA3, SM3, SHAKE
`KMACXOF`	KMAC
`ENCRYPT`	AES, DES (deprecated), 3DES (deprecated), PBE_MD5DES (deprecated), 3DES_2KEY (deprecated), RC4 (deprecated), SM4
`ENCRYPT` algorithm chaining modifiers	CBC, CFB, ECB, OFB, GCM, CCM, XTS
`ENCRYPT` algorithm padding modifiers	PAD_PKCS5, PAD_NONE, PAD_ZERO, PAD_ORCL
Public key encryption	SHA-1, SHA-2, SM2
Public key types	RSA, ECDSA, SM2
Signature algorithms	SHA1-RSA, SHA2-RSA, SHA3-RSA, SHA2-ECDSA, SHA3-ECDSA, SM3-SM2

The following table shows supported hash functions.

Table 18-2 Hash Algorithms

Hash Algorithm	Description
HASH_MD5 (deprecated)	MD5 hash
HASH_SH1 (deprecated)	SHA-1 hash
HASH_SH256	256-bit SHA-2 hash
HASH_SH384	384-bit SHA-2 hash
HASH_SH512	512-bit SHA-2 hash
HASH_SHA3_224	224-bit SHA-3 hash
HASH_SM3	SM3 hash
HASH_SHA3_256	256-bit SHA-3 hash
HASH_SHA3_384	384-bit SHA-3 hash
HASH_SHA3_512	512-bit SHA-3 hash
HASH_SHAKE128	128-bit SHAKE hash
HASH_SHAKE256	256-bit SHAKE hash

The following table shows supported HMAC algorithms.

Table 18-3 HMAC Algorithms

Algorithm	Description
HMAC_MD5 (deprecated)	MD5 HMAC
HMAC_SH1 (deprecated)	SHA-1 HMAC
HMAC_SH256	256-bit SHA-2 HMAC
HMAC_SH384	384-bit SHA-2 HMAC
HMAC_SH512	512-bit SHA-2 HMAC
HMAC_SHA3_224	224-bit SHA-3 HMAC
HMAC_SHA3_256	256-bit SHA-3 HMAC
HMAC_SHA3_384	384-bit SHA-3 HMAC
HMAC_SHA3_512	512-bit SHA-3 HMAC

The following table shows KMACXOF algorithms.

Table 18-4 KMACXOF Algorithms

Algorithm	Description
KMACXOF_128	128-bit KMAC
KMACXOF_256	256-bit KMAC

The following table shows ENCRYPT algorithms.

Table 18-5 ENCRYPT Algorithms

Algorithm	Description
ENCRYPT_RC4 (deprecated)	RC4 encrypt
ENCRYPT_DES (deprecated)	DES encrypt
ENCRYPT_3DES_2KEY (deprecated)	3DES_2KEY encrypt
ENCRYPT_3DES (deprecated)	3DES encrypt
ENCRYPT_PBE_MD5DES (deprecated)	PBE_MD5DES encrypt
ENCRYPT_AES	AES encrypt
ENCRYPT_AES128	128-bit AES encrypt
ENCRYPT_AES192	192-bit AES encrypt
ENCRYPT_AES256	256-bit AES encrypt
ENCRYPT_SM4	SM4 Encrypt

The following table shows ENCRYPT alogorithm chaining modifiers.

Table 18-6 ENCRYPT Algorithm Chaining Modifiers

Algorithm	Description
CHAIN_CBC	CBC Chain mode
CHAIN_CFB	CFB Chain mode
CHAIN_ECB	ECB Chain mode
CHAIN_OFB	OFB Chain mode
CHAIN_GCM	GCM Chain mode
CHAIN_CCM	CCM Chain mode
CHAIN_XTS	XTS Chain mode

The following table shows ENCRYPT algorithm padding modifiers.

Table 18-7 ENCRYPT Algorithm Padding Modifiers

ENCRYPT Alogorithm Padding Modifier	Description
PAD_PKCS5	PKCS#5 padding
PAD_NONE	No padding
PAD_ZERO	Zero padding
PAD_ORCL	ORCL padding

The following table shows convenience constants for block ciphers.

Table 18-8 Convenience Constants for Block Ciphers

Convenience Constant for Block Ciphers	Description
DES_CBC_PKCS5 (deprecated)	DES Encrypt with CBC Chain mode and PKCS#5 padding
DES3_CBC_PKCS5 (deprecated)	3DES Encrypt with CBC Chain mode and PKCS#5 padding
AES_CBC_PKCS5	AES Encrypt with CBC Chain mode and PKCS#5 padding
AES_GCM_NONE	AES Encrypt with GCM Chain mode and no padding
AES_CCM_NONE	AES Encrypt with CCM Chain mode and no padding
AES_XTS_NONE	AES Encrypt with XTS Chain mode and no padding
SM4_CFB_NONE	SM4 Encrypt with CFB Chain mode and no padding
SM4_OFB_NONE	SM4 Encrypt with OFB Chain mode and no padding

The following table shows public key encryption algorithms.

Table 18-9 Public Key Encryption Algorithms

Public Key Encryption Algorithm	Description
PKENCRYPT_RSA_PKCS1_OAEP (deprecated)	RSA with OAEP
PKENCRYPT_RSA_PKCS1_OAEP_SHA2	RSA with SHA-2 and OAEP
PKENCRYPT_SM2	SM2 encrypt

The following table shows public key types.

Table 18-10 Public Key Types

Public Key Type	Description
KEY_TYPE_RSA	RSA key type
KEY_TYPE_ECDSA	ECDSA key type
KEY_TYPE_SM2	SM2 key typeSM2 key type

The following table shows SIGN algorithms.

Table 18-11 Signature Algorithms

Algorithm	Description
SIGN_SHA224_RSA	224-bit SHA-2 hash function with RSA
SIGN_SHA256_RSA	256-bit SHA-2 hash function with RSA
SIGN_SHA256_RSA_X931	256-bit SHA-2 hash function with RSA and X931 padding

Table 18-11 (Cont.) Signature Algorithms

Algorithm	Description
SIGN_SHA384_RSA	384-bit SHA-2 hash function with RSA
SIGN_SHA384_RSA_X931	384-bit SHA-2 hash function with RSA and X931 padding
SIGN_SHA512_RSA	512-bit SHA-2 hash function with RSA
SIGN_SHA512_RSA_X931	512-bit SHA-2 hash function with RSA and X931 padding
SIGN_SHA1 (deprecated)	SHA-1 hash function with RSA
SIGN_SHA1_RSA_X931 (deprecated)	SHA-1 hash function with RSA and X931 padding
SIGN_SHA224_ECDSA	224-bit SHA-2 hash function with ECDSA
SIGN_SHA256_ECDSA	256-bit SHA-2 hash function with ECDSA
SIGN_SHA384_ECDSA	384-bit SHA-2 hash function with ECDSA
SIGN_SHA512_ECDSA	512-bit SHA-2 hash function with ECDSA
SIGN_ECDSA	Elliptic Curve Digital Signature Algorithm
SIGN_SM3_SM2	SM3 hash function with SM2 Signature Algorithm
SIGN_SHA3_224_RSA	224-bit SHA-3 hash function with RSA
SIGN_SHA3_256_RSA	256-bit SHA-3 hash function with RSA
SIGN_SHA3_384_RSA	384-bit SHA-3 hash function with RSA
SIGN_SHA3_512_RSA	512-bit SHA-3 hash function with RSA
SIGN_SHA3_224_ECDSA	224-bit SHA-3 hash function with ECDSA
SIGN_SHA3_256_ECDSA	256-bit SHA-3 hash function with ECDSA
SIGN_SHA3_384_ECDSA	384-bit SHA-3 hash function with ECDSA
SIGN_SHA3_512_ECDSA	512-bit SHA-3 hash function with ECDSA

DBMS_CRYPTO supports a range of algorithms that accommodate both new and existing systems. Although 3DES_2KEY and MD4 are provided for backward compatibility, you achieve better security using 3DES, AES, or SHA-1. Therefore, 3DES_2KEY is not recommended.

The DBMS_CRYPTO package includes cryptographic checksum capabilities (MD5), which are useful for comparisons, and the ability to generate a secure random number (the RANDOMBYTES function). Secure random number generation is an important part of cryptography; predictable keys are easily guessed keys; and easily guessed keys may lead to easy decryption of data. Most cryptanalysis is done by finding weak keys or poorly stored keys, rather than through brute force analysis (cycling through all possible keys).

> **Note:**
>
> Do not use `DBMS_RANDOM`, because it is unsuitable for cryptographic key generation.

Key management is programmatic. That is, the application (or caller of the function) must supply the encryption key. This means that the application developer must find a way of storing and retrieving keys securely. The relative strengths and weaknesses of various key management techniques are discussed in the sections that follow. The DES algorithm itself has an effective key length of 56-bits.

18.5 Asymmetric Key Operations with the DBMS_CRYPTO Package

The `DBMS_CRYPTO` package provides four functions that enable you to perform asymmetric key operations for encryption, decryption, signing, and verification.

Asymmetric key operations (also called public key cryptography) use a public key and private key to encrypt and decrypt a message in order to protect it from unauthorized access.

The asymmetric key operation functions are as follows:

- `PKDECRYPT` decrypts `RAW` data using a private key assisted with key algorithm and encryption algorithm.
- `PKENCRYPT` encrypts `RAW` data using a public key assisted with key algorithm and encryption algorithm.
- `SIGN` signs `RAW` data using a private key assisted with key algorithm and sign algorithm
- `VERIFY` verifies `RAW` data using signature, public key assisted with key algorithm and sign algorithm.

Related Topics

- *Oracle Database PL/SQL Packages and Types Reference*

18.6 Examples of Using the Data Encryption API

Examples of using the data encryption API include using the `DBMS_CRYPTO.SQL` procedure, encrypting AES 256-bit data, and encrypting BLOB data.

- Example: Data Encryption Procedure
  The `DBMS_CRYPTO.SQL` PL/SQL program can be used to encrypt data.
- Example: AES 256-Bit Data Encryption and Decryption Procedures
  You can use a PL/SQL block to encrypt and decrypt a predefined variable.
- Example: Encryption and Decryption Procedures for BLOB Data
  You can encrypt BLOB data.
- Example: Encrypting or Decrypting a Number String
  You can use the `DBMS_CRYPTO` PL/SQL package to create functions that will perform the on-demand encryption or decryption of a number string.

18.6.1 Example: Data Encryption Procedure

The `DBMS_CRYPTO.SQL` PL/SQL program can be used to encrypt data.

This example code performs the following actions:

- Encrypts a string (`VARCHAR2` type) using DES after first converting it into the `RAW` data type.

  This step is necessary because encrypt and decrypt functions and procedures in `DBMS_CRYPTO` package work on the `RAW` data type only.
- Shows how to create a 160-bit hash using SHA-1 algorithm.
- Demonstrates how MAC, a key-dependent one-way hash, can be computed using the MD5 algorithm. (Starting in Oracle Database release 21c, the MD5 algorithm has been deprecated.)

The `DBMS_CRYPTO.SQL` procedure follows:

```
DECLARE
    input_string     VARCHAR2(16) := 'tigertigertigert';
    raw_input        RAW(128) :=
UTL_RAW.CAST_TO_RAW(CONVERT(input_string,'AL32UTF8','US7ASCII'));
    key_string       VARCHAR2(16)  := 'scottscoscottsco';
    raw_key          RAW(128) :=
UTL_RAW.CAST_TO_RAW(CONVERT(key_string,'AL32UTF8','US7ASCII'));
    encrypted_raw    RAW(2048);
    encrypted_string VARCHAR2(2048);
    decrypted_raw    RAW(2048);
    decrypted_string VARCHAR2(2048);
-- Begin testing Encryption:
BEGIN
    dbms_output.put_line('> Input String                    : ' ||
    CONVERT(UTL_RAW.CAST_TO_VARCHAR2(raw_input),'US7ASCII','AL32UTF8'));
    dbms_output.put_line('> ========= BEGIN TEST Encrypt =========');
    encrypted_raw := dbms_crypto.Encrypt(
        src => raw_input,
        typ => DBMS_CRYPTO.AES_CBC_PKCS5,
        key => raw_key);
        dbms_output.put_line('> Encrypted hex value             : ' ||
        rawtohex(UTL_RAW.CAST_TO_RAW(encrypted_raw)));
decrypted_raw := dbms_crypto.Decrypt(
        src => encrypted_raw,
        typ => DBMS_CRYPTO.AES_CBC_PKCS5,
        key => raw_key);
    decrypted_string :=
    CONVERT(UTL_RAW.CAST_TO_VARCHAR2(decrypted_raw),'US7ASCII','AL32UTF8');
dbms_output.put_line('> Decrypted string output         : ' ||
        decrypted_string);
if input_string = decrypted_string THEN
    dbms_output.put_line('> String DES Encyption and Decryption successful');
END if;
dbms_output.put_line('');
dbms_output.put_line('> ========= BEGIN TEST Hash =========');
    encrypted_raw := dbms_crypto.Hash(
        src => raw_input,
        typ => DBMS_CRYPTO.HASH_SH1);
dbms_output.put_line('> Hash value of input string      : ' ||
        rawtohex(UTL_RAW.CAST_TO_RAW(encrypted_raw)));
dbms_output.put_line('> ========= BEGIN TEST Mac =========');
    encrypted_raw := dbms_crypto.Mac(
        src => raw_input,
```

```
        typ => DBMS_CRYPTO.HMAC_MD5,
        key => raw_key);
dbms_output.put_line('> Message Authentication Code      : ' ||
        rawtohex(UTL_RAW.CAST_TO_RAW(encrypted_raw)));
dbms_output.put_line('');
dbms_output.put_line('> End of DBMS_CRYPTO tests  ');
END;
/
```

18.6.2 Example: AES 256-Bit Data Encryption and Decryption Procedures

You can use a PL/SQL block to encrypt and decrypt a predefined variable.

For the following example, the predefined variable is named `input_string` and it uses the AES 256-bit algorithm with Cipher Block Chaining and PKCS #5 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 in the encrypted_raw variable 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');
   DBMS_OUTPUT.PUT_LINE ('Decrypted string: ' || output_string);
end;
```

18.6.3 Example: Encryption and Decryption Procedures for BLOB Data

You can encrypt BLOB data.

The following sample PL/SQL program (`blob_test.sql`) shows how to encrypt and decrypt BLOB data. This example code does the following, and prints out its progress (or problems) at each step:

- Creates a table for the BLOB column
- Inserts the raw values into that table
- Encrypts the raw data
- Decrypts the encrypted data

The `blob_test.sql` procedure follows:

```
-- 1. Create a table for BLOB column:
create table table_lob (id number, loc blob);

-- 2. Insert 3 empty lobs for src/enc/dec:
insert into table_lob values (1, EMPTY_BLOB());
insert into table_lob values (2, EMPTY_BLOB());
insert into table_lob values (3, EMPTY_BLOB());

set echo on
set serveroutput on

declare
    srcdata    RAW(1000);
    srcblob    BLOB;
    encrypblob BLOB;
    encrypraw  RAW(1000);
    encrawlen  BINARY_INTEGER;
    decrypblob BLOB;
    decrypraw  RAW(1000);
    decrawlen  BINARY_INTEGER;

    leng       INTEGER;

begin

    -- RAW input data 16 bytes
    srcdata := hextoraw('6D6D6D6D6D6D6D6D6D6D6D6D6D6D6D6D');

    dbms_output.put_line('---');
    dbms_output.put_line('input is ' || srcdata);
    dbms_output.put_line('---');

    -- select empty lob locators for src/enc/dec
    select loc into srcblob from table_lob where id = 1;
    select loc into encrypblob from table_lob where id = 2;
    select loc into decrypblob from table_lob where id = 3;

    dbms_output.put_line('Created Empty LOBS');
    dbms_output.put_line('---');

    leng := DBMS_LOB.GETLENGTH(srcblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Source BLOB Len NULL ');
    ELSE
        dbms_output.put_line('Source BLOB Len ' || leng);
    END IF;

    leng := DBMS_LOB.GETLENGTH(encrypblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Encrypt BLOB Len NULL ');
    ELSE
        dbms_output.put_line('Encrypt BLOB Len ' || leng);
    END IF;

    leng := DBMS_LOB.GETLENGTH(decrypblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Decrypt  BLOB Len NULL ');
    ELSE
        dbms_output.put_line('Decrypt BLOB Len ' || leng);
    END IF;
```

```
    -- 3. Write source raw data into blob:
    DBMS_LOB.OPEN (srcblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.WRITEAPPEND (srcblob, 16, srcdata);
    DBMS_LOB.CLOSE (srcblob);

    dbms_output.put_line('Source raw data written to source blob');
    dbms_output.put_line('---');

    leng := DBMS_LOB.GETLENGTH(srcblob);
    IF leng IS NULL THEN
        dbms_output.put_line('source BLOB Len NULL ');
    ELSE
        dbms_output.put_line('Source BLOB Len ' || leng);
    END IF;

    /*
    * Procedure Encrypt
    * Arguments: srcblob -> Source BLOB
    *            encrypblob -> Output BLOB for encrypted data
    *            DBMS_CRYPTO.AES_CBC_PKCS5 -> Algo : AES
    *                                         Chaining : CBC
    *                                         Padding : PKCS5
    *            256 bit key for AES passed as RAW
    *                ->
    hextoraw('000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F')
    *            IV (Initialization Vector) for AES algo passed as RAW
    *                -> hextoraw('00000000000000000000000000000000')
    */

    DBMS_CRYPTO.Encrypt(encrypblob,
                srcblob,
                DBMS_CRYPTO.AES_CBC_PKCS5,
                hextoraw
('000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F'),
                hextoraw('00000000000000000000000000000000'));


    dbms_output.put_line('Encryption Done');
    dbms_output.put_line('---');

    leng := DBMS_LOB.GETLENGTH(encrypblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Encrypt BLOB Len NULL');
    ELSE
        dbms_output.put_line('Encrypt BLOB Len ' || leng);
    END IF;

    -- 4. Read encrypblob to a raw:
    encrawlen := 999;

    DBMS_LOB.OPEN (encrypblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.READ (encrypblob, encrawlen, 1, encrypraw);
    DBMS_LOB.CLOSE (encrypblob);

    dbms_output.put_line('Read encrypt blob to a raw');
    dbms_output.put_line('---');

    dbms_output.put_line('Encrypted data is (256 bit key) ' || encrypraw);
    dbms_output.put_line('---');

    /*
```

```
    * Procedure Decrypt
    * Arguments: encrypblob -> Encrypted BLOB to decrypt
    *            decrypblob -> Output BLOB for decrypted data in RAW
    *            DBMS_CRYPTO.AES_CBC_PKCS5 -> Algo : AES
    *                                         Chaining : CBC
    *                                         Padding : PKCS5
    *            256 bit key for AES passed as RAW (same as used during Encrypt)
    *                -> 
    hextoraw('000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F')
    *            IV (Initialization Vector) for AES algo passed as RAW (same as
    *            used during Encrypt)
    *                -> hextoraw('00000000000000000000000000000000')
    */

    DBMS_CRYPTO.Decrypt(decrypblob,
                encrypblob,
                DBMS_CRYPTO.AES_CBC_PKCS5,
                hextoraw
           ('000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F'),
                hextoraw('00000000000000000000000000000000'));

    leng := DBMS_LOB.GETLENGTH(decrypblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Decrypt BLOB Len NULL');
    ELSE
        dbms_output.put_line('Decrypt BLOB Len ' || leng);
    END IF;

    -- Read decrypblob to a raw
    decrawlen := 999;

    DBMS_LOB.OPEN (decrypblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.READ (decrypblob, decrawlen, 1, decrypraw);
    DBMS_LOB.CLOSE (decrypblob);

    dbms_output.put_line('Decrypted data is (256 bit key) ' || decrypraw);
    dbms_output.put_line('---');

    DBMS_LOB.OPEN (srcblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.TRIM (srcblob, 0);
    DBMS_LOB.CLOSE (srcblob);

    DBMS_LOB.OPEN (encrypblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.TRIM (encrypblob, 0);
    DBMS_LOB.CLOSE (encrypblob);

    DBMS_LOB.OPEN (decrypblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.TRIM (decrypblob, 0);
    DBMS_LOB.CLOSE (decrypblob);

end;
/

truncate table table_lob;
drop table table_lob;
```

18.6.4 Example: Encrypting or Decrypting a Number String

You can use the `DBMS_CRYPTO` PL/SQL package to create functions that will perform the on-demand encryption or decryption of a number string.

The following procedure provides an example of how you can create and use functions to encrypt and decrypt number strings. It also provides an example of testing how the functions work by inserting a converted number string into a table.

1. Create a function that will encrypt a number string.

   The following example function, `f_encrypt_number`, uses the input value `number_in`, the return value as the raw type, and `DES_CBC_PKCS5` as the encryption algorithm.

```
CREATE OR REPLACE FUNCTION f_encrypt_number(number_in IN NUMBER)
 RETURN RAW IS
  number_in_raw RAW(128):=UTL_I18N.STRING_TO_RAW(number_in,'AL32UTF8');
  key_number number(32):=32432432343243279898;
  key_raw RAW(128):=UTL_RAW.cast_from_number(key_number);
  encrypted_raw RAW(128);
BEGIN

encrypted_raw:=DBMS_CRYPTO.ENCRYPT(src=>number_in_raw,typ=>DBMS_CRYPTO.DES_
CBC_PKCS5,key=>key_raw);
  RETURN encrypted_raw;
END;
/
```

2. Run the function `f_encrypt_number` to encrypt the number string `2`.

```
SELECT f_encrypt_number('2') FROM DUAL;
```

   The result in this example is `84A8B8D7D8925582`:

```
F_ENCRYPT_NUMBER('2')
--------------------------------------------------------------------------
-----
84A8B8D7D8925582
```

3. Create a function to decrypt a number string.

   The following example function, `f_decrypt_number`, can decrypt an encrypted raw value `encrypted_raw`. The input is `encrypted_raw`. It uses `DES_CBC_PKCS5` as the decryption algorithm

```
CREATE OR REPLACE FUNCTION f_decrypt_number (encrypted_raw IN RAW)
 RETURN NUMBER IS
  decrypted_raw raw(48);
  key_number number(32):=32432432343243279898;
  key_raw RAW(128):=UTL_RAW.cast_from_number(key_number);
BEGIN
  decrypted_raw := DBMS_CRYPTO.DECRYPT
      (
         src => encrypted_raw,
         typ => DBMS_CRYPTO.DES_CBC_PKCS5,
```

```
         key => key_raw
      );
RETURN   UTL_I18N.RAW_TO_CHAR (decrypted_raw, 'AL32UTF8');
END;
/
```

4. Run the function `f_decrypt_number` to decrypt `84A8B8D7D8925582`.

   :

   ```
   SELECT f_decrypt_number('84A8B8D7D8925582') FROM DUAL;
   ```

   The result is 2:

   ```
   F_DECRYPT_NUMBER('84A8B8D7D8925582')
   -----------------------------------
                                      2
   ```

5. Test the encrypted number string.

   In this test, you run `f_encrypt_number` to encrypt number `2`. (The result should be `84A8B8D7D8925582`). Then you insert (`f_encrypt_number('2')`, `username`) into table `test_dbms_crypto`. You will be able to see `84A8B8D7D8925582 username` inserted to the table. When you run `f_encrypt_number` to decrypt the ID `84A8B8D7D8925582`, the result is `2`.

   a. Insert the encrypted number string into the `test_dbms_crypto` table.

      ```
      INSERT INTO test_dbms_crypto VALUES (f_encrypt_number('2'),'username');
      1 row created.
      COMMIT;
      Commit complete.
      ```

   b. Select from the `test_dbms_crypto` table.

      ```
      SELECT * FROM test_dbms_crypto;
      ```

      The following output should appear:

      ```
      ID                          NAME
      --------------------------  --------------------
      84A8B8D7D8925582            username
      ```

   c. Select from the `test_dbms_crypto` table.

      ```
      SELECT f_decrypt_number(id), NAME FROM test_dbms_crypto ;
      ```

      The following output should appear:

      ```
      F_DECRYPT_NUMBER(ID) NAME
      -------------------- --------------------
                         2 username
      ```