SYBASE®

An SAP® Company

Advanced Security in Sybase IQ

Sybase IQ 15.3

DOCUMENT ID: DC01015-01-1530-01
LAST REVISED: May 2011

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

Advanced Security in Sybase IQ

The Sybase® IQ Advanced Security Option supports column encryption, Federal Information Processing Standards (FIPS) approved network encryption technology, and Kerberos authentication for database connections, operating system logins, and network logins. The Advanced Security Option is a separately licensed Sybase IQ option.

FIPS Support in Sybase IQ

Sybase IQ includes enhancements to Federal Information Processing Standards (FIPS) approved encryption technology. FIPS is supported on all platforms supported by Sybase IQ.

The main impact of FIPS support for Sybase IQ is that encryption can be nondeterministic, which is now the default behavior. A nondeterministic algorithm is one in which the same input yields different output values each time. This means that when you use a key to encrypt a string, the encrypted string is different each time. The algorithm, however, is still able to decrypt the nondeterministic result using the key. This feature makes analyzing the encryption algorithm more difficult, and encryption more secure.

Support of FIPS is part of the separately licensed Sybase IQ Advanced Security Option.

Both RSA and FIPS security are included with Sybase IQ. RSA encryption requires no separate libraries, but FIPS requires two optional libraries: `dbfips11.dll` and `sbgse2.dll`. The library `sbgse2.dll` is provided by Certicom. Both security models require certificates. The **rsaserver** certificate has been renamed from `rsaserver.crt` to `rsaserver.id`.

FIPS also requires this registry setting, which is set automatically by the Sybase IQ installation utility:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Certicom\libsb]
"expectedtag"=hex:5b,0f,4f,a6,e2,4a,ef,3b,
44,07,05,2e,b0,49,02,71,1f,d9,91,b6
```

See *SQL Anywhere 11.0.1 > SQL Anywhere Server – Database Administration > Security > Transport-layer security* and *SQL Anywhere 11.0.1 > SQL Anywhere Server – Database Administration > Security > Keeping your data secure*.

Kerberos Authentication Support in Sybase IQ

Sybase IQ supports Kerberos authentication, a login feature that allows you to maintain a single user ID and password for both database connections and operating system and network logins.

You can use your Kerberos credentials to connect to the database without specifying a user ID or password.

Kerberos authentication is part of the separately licensed Sybase IQ Advanced Security Option.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server – Database Administration > Starting and Connecting to Your Database > SQL Anywhere database connections > Kerberos authentication*.

Column Encryption in Sybase IQ

Sybase IQ supports user encrypted columns.

Strong encryption of the Sybase IQ database file uses a 128-bit algorithm and a security key. The data is unreadable and virtually undecipherable without the key. The algorithm supported is described in FIPS-197, the Federal Information Processing Standard for the Advanced Encryption Standard.

Sybase IQ supports user encrypted columns with the **AES_ENCRYPT** and **AES_DECRYPT** functions and the **LOAD TABLE ENCRYPTED** clause. These functions permit explicit encryption and decryption of column data via calls from the application. Encryption and decryption key management is the responsibility of the application.

Note: Users must be specifically licensed to use the encrypted column functionality of the Sybase IQ Advanced Security Option described in this product documentation.

Certain database options affect column encryption. Before using this feature, see *Database Options for Column Encryption*.

See also

- *Database Options for Column Encryption* on page 9

Definitions of Encryption Terms

Definitions of terms used when describing encryption of stored data.

- plaintext – data in its original, intelligible form. Plaintext is not limited to string data, but is used to describe any data in its original representation.

- ciphertext – data in an unintelligible form that preserves the information content of the plaintext form.
- encryption – a reversible transformation of data from plaintext to ciphertext. Also known as enciphering.
- decryption – the reverse transformation of ciphertext back to plaintext. Also known as deciphering.
- key – a number used to encrypt or decrypt data. Symmetric-key encryption systems use the same key for both encryption and decryption. Asymmetric-key systems use one key for encryption and a different (but mathematically related) key for decryption. The Sybase IQ interfaces accept character strings as keys.
- Rijndael – pronounced “reign dahl.” A specific encryption algorithm that supports a variety of key and block sizes. The algorithm was designed to use simple whole-byte operations and thus is relatively easy to implement in software.
- AES – the Advanced Encryption Standard, a FIPS-approved cryptographic algorithm for the protection of sensitive (but unclassified) electronic data. AES adopted the Rijndael algorithm with restrictions on the block sizes and key lengths. AES is the algorithm supported by Sybase IQ.

Data Types for Encrypted Columns

The data types supported for encrypted columns and working with these data types.

Supported Data Types

The first parameter of the **AES_ENCRYPT** function must be one of the supported data types.

CHAR	NUMERIC
VARCHAR	FLOAT
TINYINT	REAL
SMALLINT	DOUBLE
INTEGER	DECIMAL
BIGINT	DATE
BIT	TIME
BINARY	DATETIME
VARBINARY	TIMESTAMP
UNSIGNED INT	SMALLDATETIME
UNSIGNED BIGINT	

The LOB data type is not currently supported for Sybase IQ column encryption.

Preservation of Data Types

Sybase IQ ensures that the original data type of the plaintext is preserved when decrypting data, if the **AES_DECRYPT** function is given the data type as a parameter, or is within a **CAST** function.

Sybase IQ compares the target data type of the **CAST** with the data type of the originally encrypted data. If the two data types do not match, a -1001064 error is returned with details about the original and target data types.

For example, given an encrypted `VARCHAR(1)` value and this valid decryption statement:

```
SELECT AES_DECRYPT ( thecolumn, ‘theKey’,
VARCHAR(1) ) FROM thetable
```

If you attempt to decrypt the data using:

```
SELECT AES_DECRYPT ( thecolumn, ‘theKey’,
SMALLINT ) FROM thetable
```

the error returned is:

```
Decryption error: Incorrect CAST type smallint(5,0)
for decrypt data of type varchar(1,0).
```

This data type check is made only when supplied. Without the **CAST** or the data type parameter, the query returns the ciphertext as binary data.

Note: When using the **AES_ENCRYPT** function on literal constants, as in this statement:

```
INSERT INTO t (cipherCol) VALUES (AES_ENCRYPT (1, ‘key’))
```

be aware that the data type of 1 is ambiguous. The data type of 1 can be a `TINYINT`, `SMALLINT`, `INTEGER`, `UNSIGNED INT`, `BIGINT`, `UNSIGNED BIGINT` or possibly other data types.

Sybase recommends explicit use of the **CAST** function to resolve any potential ambiguity, as in:

```
INSERT INTO t (cipherCol)
VALUES ( AES_ENCRYPT (CAST (1 AS UNSIGNED INTEGER), ‘key’))
```

Explicitly converting the data type using the **CAST** function when encrypting data prevents problems using the **CAST** function when the data is decrypted.

There is no ambiguity if the data being encrypted is from a column or if the encrypted data was inserted by **LOAD TABLE**.

Effect of Different Data Types on Ciphertext

Cast the input of **AES_ENCRYPT** to the same data type to produce identical ciphertext.

The ciphertext produced by **AES_ENCRYPT** differs for two different data types given the same input value and same key. A join of two ciphertext columns that hold encrypted values of two different data types may therefore not return identical results.

For example, assume:

```
CREATE TABLE tablea(c1 int, c2 smallint);
INSERT INTO tablea VALUES (100,100);
```

The value `AES_ENCRYPT(c1, 'key')` differs from `AES_ENCRYPT(c2,'key')` and the value `AES_ENCRYPT(c1,'key')` differs from `AES_ENCRYPT(100,'key')`.

To resolve this issue, cast the input of **AES_ENCRYPT** to the same data type. For example, the results of these code fragments are the same:

```
AES_ENCRYPT(c1, 'key');
```

```
AES_ENCRYPT(CAST(c2 AS INT), 'key');
```

```
AES_ENCRYPT(CAST(100 AS INT), 'key');
```

See also

- *AES_ENCRYPT Function [String]* on page 5

AES_ENCRYPT Function [String]

Encrypts the specified values using the supplied encryption key, and returns a `VARBINARY` or `LONG VARBINARY`.

Syntax

```
AES_ENCRYPT( string-expression, key )
```

Parameters

string-expression – the data to be encrypted. Binary values can also be passed to **AES_ENCRYPT**. This parameter is case-sensitive, even in case-insensitive databases.

key – the encryption key used to encrypt the *string-expression*. To obtain the original value, you must also use the same key to decrypt the value. This parameter is case sensitive, even in case-insensitive databases.

As with most passwords, it is best to choose a key value that is difficult to guess. Sybase recommends that you choose a value for your key that is at least 16 characters long, contains a mix of uppercase and lowercase letters, and includes numbers and special characters. You need this key each time you want to decrypt the data.

Warning! Protect your key; store a copy of your key in a safe location. If you lose your key, encrypted data becomes completely inaccessible and unrecoverable.

Usage

AES_ENCRYPT returns a `VARBINARY` value, which is at most 31 bytes longer than the input *string-expression*. The value returned by this function is the ciphertext, which is not human-readable. You can use the **AES_DECRYPT** function to decrypt a *string-expression* that was encrypted with the **AES_ENCRYPT** function. To successfully decrypt a *string-expression*, use the same encryption key and algorithm used to encrypt the data. If you specify an incorrect encryption key, an error is generated.

If you are storing encrypted values in a table, the column should be of data type `VARBINARY` or `VARCHAR`, and greater than or equal to 32 bytes, so that character set conversion is not performed on the data. (Character set conversion would prevent decryption of the data.) If the length of the `VARBINARY` or `VARCHAR` column is less than 32 bytes, then the **AES_DECRYPT** function returns an error.

The result data type of an **AES_ENCRYPT** function may be a `LONG BINARY`. If you use **AES_ENCRYPT** in a **SELECT INTO** statement, you must have an Unstructured Data Analytics Option license, or use **CAST** and set **AES_ENCRYPT** to the correct data type and size.

For additional details and usage information, see *Reference: Building Blocks, Tables, and Procedures > SQL Functions > Alphabetical List of Functions > REPLACE function [String]*.

Standards and Compatibility

- SQL – vendor extension to ISO/ANSI SQL grammar
- Sybase – not supported by Adaptive Server Enterprise

See also

- *AES_DECRYPT Function [String]* on page 6
- *Encryption and Decryption Example* on page 11
- *LOAD TABLE ENCRYPTED Clause* on page 7
- *Effect of Different Data Types on Ciphertext* on page 5
- *Data Types for Encrypted Columns* on page 3

AES_DECRYPT Function [String]

Decrypts the string using the supplied key, and returns, by default, a `VARBINARY` or `LONG BINARY`, or the original plaintext type.

Syntax

```
AES_DECRYPT( string-expression, key [, data-type ] )
```

Parameters

string-expression – the string to be decrypted. Binary values can also be passed to this function. This parameter is case sensitive, even in case-insensitive databases.

key – the encryption key required to decrypt the *string-expression*. To obtain the original value that was encrypted, the key must be the same encryption key that was used to encrypt the *string-expression*. This parameter is case-sensitive, even in case-insensitive databases.

Warning! Protect your key; store a copy of your key in a safe location. If you lose your key, the encrypted data becomes completely inaccessible and unrecoverable.

data-type – this optional parameter specifies the data type of the decrypted *string-expression* and must be the same data type as the original plaintext.

If you do not use a **CAST** statement while inserting data using the **AES_ENCRYPT** function, you can view the same data using the **AES_DECRYPT** function by passing `VARCHAR` as the *data-type*. If you do not pass *data-type* to **AES_DECRYPT**, `VARBINARY` data type is returned.

Usage

You can use the **AES_DECRYPT** function to decrypt a *string-expression* that was encrypted with the **AES_ENCRYPT** function. This function returns a `VARBINARY` or `LONG VARBINARY` value with the same number of bytes as the input string, if no data type is specified. Otherwise, the specified data type is returned.

To successfully decrypt a *string-expression*, you must use the same encryption key that was used to encrypt the data. An incorrect encryption key returns an error.

Example

Decrypt the password of a user from the `user_info` table.

```
SELECT AES_DECRYPT(user_pwd, '8U3dkA', CHAR(100))
FROM user_info;
```

Standards and Compatibility

- SQL – vendor extension to ISO/ANSI SQL grammar
- Sybase – not supported by Adaptive Server Enterprise

See also

- *AES_ENCRYPT Function [String]* on page 5
- *Encryption and Decryption Example* on page 11
- *LOAD TABLE ENCRYPTED Clause* on page 7
- *Data Types for Encrypted Columns* on page 3

LOAD TABLE ENCRYPTED Clause

The **LOAD TABLE** statement supports the column-spec keyword **ENCRYPTED**.

The *column-specs* must follow the column name in a **LOAD TABLE** statement in this order:

- *format-specs*

- *null-specs*
- *encrypted-specs*

For full syntax, see *Reference: Statements and Options > SQL Statements > LOAD TABLE statement*.

Syntax

```
| ENCRYPTED(data-type ‘key-string’ [, ‘algorithm-string’ ] )
```

Parameters

- *data-type* – the data type that the input file field should be converted to as input to the **AES_ENCRYPT** function. *data-type* should be the same data type as the data type of the output of the **AES_DECRYPT** function.
- *key-string* – the encryption key used to encrypt the data. This key must be a string literal. To obtain the original value, you must use the same key to decrypt the value. This parameter is case sensitive, even in case-insensitive databases.

  As with most passwords, it is best to choose a key value that cannot be easily guessed. Sybase recommends that you choose a value for your key that is at least 16 characters long, contains a mix of uppercase and lowercase letters, and includes numbers and special characters. You will need this key each time you want to decrypt the data.

  Warning! Protect your key; store a copy of your key in a safe location. A lost key results in the encrypted data becoming completely inaccessible, from which there is no recovery.
- *algorithm-string* – the algorithm used to encrypt the data. This parameter is optional, but data must be encrypted and decrypted using the same algorithm. Currently, AES is the default, as it is the only supported algorithm. AES is a block encryption algorithm chosen as the new Advanced Encryption Standard (AES) for block ciphers by the National Institute of Standards and Technology (NIST).

Usage

The **ENCRYPTED** column specification allows you to specify the encryption key and, optionally, the algorithm to use to encrypt the data that is loaded into the column. The target column for this load should be `VARBINARY`. Specifying other data types returns an error.

LOAD TABLE ENCRYPTED Example

```
LOAD TABLE table_name
(
plaintext_column_name,
a_ciphertext_column_name
NULL('nil')
ENCRYPTED(varchar(6),'tHefiRstkEy') ,
another_encrypted_column
ENCRYPTED(bigint,'thEseconDkeY','AES')
)
FROM '/path/to/the/input/file'
FORMAT ascii
DELIMITED BY ';'
```

```
ROW DELIMITED BY '\0xa'
QUOTES OFF
ESCAPES OFF
```

where the format of the input file for the **LOAD TABLE** statement is:

```
a;b;c;
d;e;f;
g;h;i;
```

See also

- *AES_ENCRYPT Function [String]* on page 5
- *AES_DECRYPT Function [String]* on page 6
- *Encryption and Decryption Example* on page 11
- *Data Types for Encrypted Columns* on page 3

String Comparisons on Encrypted Text

If data is case insensitive, or uses a collation other than ISO_BINENG, you must decrypt ciphertext columns to perform string comparisons.

When performing comparisons on strings, the distinction between equal and identical strings is important for many collations and depends on the **CASE** option of **CREATE DATABASE**. In a database that is set to **CASE RESPECT** and uses the ISO_BINENG collation, the defaults for Sybase IQ, equality, and identity questions are resolved the same way.

Identical strings are always equal, but equal strings may not be identical. Strings are identical only if they are represented using the same byte values. When data is case insensitive or uses a collation where multiple characters must be treated as equal, the distinction between equality and identity is significant. ISO1LATIN1 is such a collation.

For example, the strings “ABC” and “abc” in a case insensitive database are not identical but are equal. In a case sensitive database, they are neither identical nor equal.

The ciphertext created by the Sybase encryption functions preserves identity but not equality. In other words, the ciphertext for “ABC” and “abc” will never be equal.

To perform equality comparisons on ciphertext when your collation or **CASE** setting does not allow this type of comparison, your application must modify the values within that column into some canonical form, where there are no equal values that are not also identical values. For example, if your database is created with **CASE IGNORE** and the ISO_BINENG collation and your application applies UCASE to all input values before placing them into the column, then all equal values are also identical.

Database Options for Column Encryption

Certain Sybase IQ database option settings affect column encryption and decryption; the default settings are not optimal for most column encryption operations.

Protect Ciphertext from Accidental Truncation

Set STRING_RTRUNCATION to protect ciphertext from accidental truncation.

To prevent accidental truncation of the ciphertext output of the encrypt function (or accidental truncation of any other character or binary string), set this database option:

```
SET OPTION STRING_RTRUNCATION = 'ON'
```

When STRING_RTRUNCATION is ON (the default), the engine raises an error whenever a string would be truncated during a load, insert, update, or **SELECT INTO** operation. This is ISO/ANSI SQL behavior and is a recommended practice.

When explicit truncation is required, use a string expression such as **LEFT**, **SUBSTRING**, or **CAST**.

Setting STRING_RTRUNCATION OFF forces silent truncation of strings.

The **AES_DECRYPT** function also checks input ciphertext for valid data length, and checks text output to verify both the resulting data length and the correctness of the supplied key. (If the data type argument is supplied, the data type is checked as well.)

Preserve Ciphertext Integrity

Set ASE_BINARY_DISPLAY to preserve ciphertext integrity.

```
SET OPTION ASE_BINARY_DISPLAY = 'OFF'
```

When ASE_BINARY_DISPLAY is OFF (the default), the system leaves binary data unmodified, and in its raw binary form.

When ASE_BINARY_DISPLAY is ON, the system converts binary data into its hexadecimal string display representation. Temporarily set the option to ON only if you need data to display to an end user or if you need to export the data to another external system, where raw binary could become altered in transit.

Prevent Misuse of Ciphertext

Set CONVERSION_MODE to prevent implicit data type conversions of encrypted data that would result in semantically meaningless operations.

The CONVERSION_MODE database option restricts implicit conversion between binary data types (BINARY, VARBINARY, and LONG BINARY) and other nonbinary data types (BIT, TINYINT, SMALLINT, INT, UNSIGNED INT, BIGINT, UNSIGNED BIGINT, CHAR, VARCHAR, and LONG VARCHAR) on various operations. Use CONVERSION_MODE to prevent implicit data type conversions of encrypted data that would result in semantically meaningless operations:

```
SET TEMPORARY OPTION CONVERSION_MODE = 1
```

Setting CONVERSION_MODE to 1 restricts implicit conversion of binary data types to any other nonbinary data type on **INSERT** and **UPDATE** commands, and in queries. The restrict binary conversion mode also applies to **LOAD TABLE** default values and **CHECK** constraint.

The CONVERSION_MODE option default value of 0 maintains the implicit conversion behavior of binary data types in versions of Sybase IQ earlier than 12.7.

See *Reference: Statements and Options > Database Options > Alphabetical List of Options > CONVERSION_MODE Option*.

Encryption and Decryption Example

An example using the **AES_ENCRYPT** and **AES_DECRYPT** functions, written in commented SQL.

```
--   This example of aes_encrypt and aes_decrypt function use is
presented in three parts:
--
-- Part I: Preliminary description of target tables and users as DDL
-- Part II: Example schema changes motivated by introduction of
encryption
-- Part III: Use of views and stored procedures to protect encryption
keys
--


--   Part I: Define target tables and users

--   Assume two classes of user, represented here by the instances
--   PrivUser and NonPrivUser, assigned to groups reflecting
differing
--   privileges.

--   The initial state reflects the schema prior to the introduction
--   of encryption.

--   Set up the starting context: There are two tables with a common
key.
--   Some columns contain sensitive data, the remaining columns do
not.
--   The usual join column for these tables is sensitiveA.
--   There is a key and a unique index.

    grant connect to PrivUser identified by 'verytrusted' ;
    grant connect to NonPrivUser identified by 'lesstrusted' ;

    grant connect to high_privileges_group ;
    grant group to high_privileges_group ;
    grant membership in group high_privileges_group to PrivUser ;

    grant connect to low_privileges_group ;
    grant group to low_privileges_group ;
    grant membership in group low_privileges_group to NonPrivUser ;
    create table DBA.first_table
                    (sensitiveA char(16) primary key
                    ,sensitiveB numeric(10,0)
                    ,publicC    varchar(255)
                    ,publicD    date
                    )  ;


--   There is an implicit unique HG (HighGroup) index enforcing the
primary key.

    create table second_table
                    (sensitiveA char(16)
                    ,publicP integer
                    ,publicQ tinyint
                    ,publicR varchar(64)
                    )  ;

    create hg index second_A_HG on second_table ( sensitiveA ) ;

--   TRUSTED users can see the sensitive columns.

    grant select ( sensitiveA, sensitiveB, publicC, publicD )
        on DBA.first_table to PrivUser ;
    grant select ( sensitiveA, publicP, publicQ, publicR )
        on DBA.second_table to PrivUser ;

--   Non-TRUSTED users in existing schema need to see sensitiveA to
be
--   able to do joins, even though they should not see sensitiveB.

    grant select ( sensitiveA, publicC, publicD )
        on DBA.first_table to NonPrivUser ;
    grant select ( sensitiveA, publicP, publicQ, publicR )
        on DBA.second_table to NonPrivUser ;

--   Non-TRUSTED users can execute queries such as

    select I.publicC, 3*II.publicQ+1
    from DBA.first_table I, DBA.second_table II
    where I.sensitiveA = II.sensitiveA and I.publicD IN
( '2006-01-11' ) ;

--   and

    select count(*)
    from DBA.first_table I, DBA.second_table II
    where I.sensitiveA = II.sensitiveA and SUBSTR(I.sensitiveA,4,3)
    BETWEEN '345' AND '456' ;

--   But only TRUSTED users can execute the query

    select I.sensitiveB, 3*II.publicQ+1
    from DBA.first_table I, DBA.second_table II
    where I.sensitiveA = II.sensitiveA and I.publicD IN
( '2006-01-11' ) ;
```

```
--   Part II: Change the schema in preparation for encryption
--
--    The DBA introduces encryption as follows:
--
--    For applicable tables, the DBA changes the schema, adjusts
access
--    permissions, and updates existing data. The encryption
--    keys used are hidden in a subsequent step.


--   DataLength comparison for length of varbinary encryption result
--   (units are Bytes):
--
--    PlainText CipherText    Corresponding Numeric Precisions
--
--            0       16
--      1 -  16       32      numeric(1,0)   - numeric(20,0)
--     17 -  32       48      numeric(21,0)  - numeric(52,0)
--     33 -  48       64      numeric(53,0)  - numeric(84,0)
--     49 -  64       80      numeric(85,0)  - numeric(116,0)
--     65 -  80       96      numeric(117,0) - numeric(128,0)
--     81 -  96      112
--     97 - 112      128
--    113 - 128      144
--    129 - 144      160
--    145 - 160      176
--    161 - 176      192
--    177 - 192      208
--    193 - 208      224
--    209 - 224      240

--   The integer data types tinyint, small int, integer, and bigint
--   are varbinary(32) ciphertext.

--   The exact relationship is
--   DATALENGTH(ciphertext) =
--   (((DATALENGTH(plaintext)+ 15) / 16) + 1) * 16

--   For the first table, the DBA chooses to preserve both the
plaintext and
--   ciphertext forms. This is not typical and should only be done if
the
--   database files are also encrypted.

--   Take away NonPrivUser's access to column sensitiveA and transfer
--   access to the ciphertext version.

--   Put a unique index on the ciphertext column. The ciphertext
--   itself is indexed.

--   NonPrivUser can select the ciphertext and use it.

--   PrivUser can still select either form (without paying decrypt
costs).
```

```
    revoke select ( sensitiveA ) on DBA.first_table from
NonPrivUser ;
    alter table DBA.first_table add encryptedA varbinary(32) ;
    grant select ( encryptedA ) on DBA.first_table to PrivUser ;
    grant select ( encryptedA ) on DBA.first_table to NonPrivUser ;
    create unique hg index first_A_unique on first_table
( encryptedA ) ;
    update DBA.first_table
        set encryptedA = aes_encrypt(sensitiveA, 'seCr3t')
        where encryptedA is null ;
    commit

--   Now change column sensitiveB.

    alter table DBA.first_table add encryptedB varbinary(32) ;
    grant select ( encryptedB ) on DBA.first_table to PrivUser ;
    create unique hg index first_B_unique on first_table
( encryptedB ) ;
    update DBA.first_table
        set encryptedB = aes_encrypt(sensitiveB,
        'givethiskeytonoone') where encryptedB is null ;
    commit

--   For the second table, the DBA chooses to keep only the
ciphertext.
--   This is more typical and encrypting the database files is not
required.

    revoke select ( sensitiveA ) on DBA.second_table from
NonPrivUser ;
    revoke select ( sensitiveA ) on DBA.second_table from PrivUser ;
    alter table DBA.second_table add encryptedA varbinary(32) ;
    grant select ( encryptedA ) on DBA.second_table to PrivUser ;
    grant select ( encryptedA ) on DBA.second_table to NonPrivUser ;
    create unique hg index second_A_unique on second_table
( encryptedA ) ;
    update DBA.second_table
        set encryptedA = aes_encrypt(sensitiveA, 'seCr3t')
        where encryptedA is null ;
    commit
    alter table DBA.second_table drop sensitiveA ;

--   The following types of queries are permitted at this point,
before
--   changes are made for key protection:

--   Non-TRUSTED users can equi-join on ciphertext; they can also
select
--   the binary, but have no way to interpret it.

    select I.publicC, 3*II.publicQ+1
    from DBA.first_table I, DBA.second_table II
    where I.encryptedA = II.encryptedA and I.publicD IN
( '2006-01-11' ) ;
```

```
--   Ciphertext-only access rules out general predicates and
expressions.
--   The following query does not return meaningful results.
--
--   NOTE: These four predicates can be used on the varbinary
containing
--   ciphertext:
--      = (equality)
--      <> (inequality)
--      IS NULL
--      IS NOT NULL

    select count(*)
    from DBA.first_table I, DBA.second_table II
    where I.encryptedA = II.encryptedA and SUBSTR(I.encryptedA,4,3)
        BETWEEN '345' AND '456' ;

--   The TRUSTED user still has access to the plaintext columns that
--   were retained. Therefore, this user does not need to call
--   aes_decrypt and does not need the key.

    select count(*)
    from DBA.first_table I, DBA.second_table II
    where I.encryptedA = II.encryptedA and SUBSTR(I.sensitiveA,4,3)
        BETWEEN '345' AND '456' ;


--   Part III: Protect the encryption keys

--   This section illustrates how to grant access to the plaintext,
but
--   still protect the keys.

--   For the first table, the DBA elected to retain the plaintext
columns.
--   Therefore, the following view has the same capabilities as the
trusted
--   user above.
--   Assume group_member is being used for additional access control.

--   NOTE: In this example, NonPrivUser still has access to the
ciphertext
--   encrypted in the base table.

    create view DBA.a_first_view (sensitiveA, publicC, publicD)
        as
            select
            IF group_member('high_privileges_group',user_name()) = 1
                 THEN sensitiveA
                 ELSE NULL
              ENDIF,
              publicC,
              publicD
            from first_table ;

    grant select on DBA.a_first_view to PrivUser ;
    grant select on DBA.a_first_view to NonPrivUser ;

--   For the second table, the DBA did not keep the plaintext.
--   Therefore, aes_decrypt calls must be used in the view.
--   IMPORTANT: Hide the view definition with ALTER VIEW, so that no
one
--   can discover the key.

    create view DBA.a_second_view
(sensitiveA,publicP,publicQ,publicR)
        as
            select
            IF group_member('high_privileges_group',user_name()) = 1
                 THEN aes_decrypt(encryptedA,'seCr3t', char(16))
                 ELSE NULL
              ENDIF,
              publicP,
              publicQ,
              publicR
            from second_table ;

    alter view DBA.a_second_view set hidden ;
    grant select on DBA.a_second_view to PrivUser ;
    grant select on DBA.a_second_view to NonPrivUser ;

--   Likewise, the key used for loading can be protected in a stored
     procedure.
--   By hiding the procedure (just as the view is hidden), no-one can
see
--   the keys.

    create procedure load_first_proc(@inputFileName varchar(255),
                        @colDelim varchar(4) default '$',
                        @rowDelim varchar(4) default '\n')
        begin
            execute immediate with quotes
                'load table DBA.second_table
                (encryptedA encrypted(char(16),' ||
             '''' || 'seCr3t' || '''' || '),publicP,publicQ,publicR)
' ||
                ' from ' || '''' || @inputFileName || '''' ||
                ' delimited by ' || '''' || @colDelim || '''' ||
                ' row delimited by ' || '''' || @rowDelim || '''' ||
                ' quotes off escapes off' ;
        end
    ;

    alter procedure DBA.load_first_proc set hidden ;

--   Call the load procedure using the following syntax:

    call load_first_proc('/dev/null', '$', '\n') ;

--   Below is a comparison of several techniques for protecting the
--   encryption keys by using user-defined functions (UDFs), other
views,
--   or both. The first and the last alternatives offer maximum
performance.

--   The second_table is secured as defined earlier.


--   Alternative 1:
--   This baseline approach relies on restricting access to the
entire view.

     create view

DBA.second_baseline_view(sensitiveA,publicP,publicQ,publicR)
        as
            select
            IF group_member('high_privileges_group',user_name()) = 1
                 THEN aes_decrypt(encryptedA,'seCr3t', char(16))
                 ELSE NULL
              ENDIF,
              publicP,
              publicQ,
              publicR
            from DBA.second_table ;

     alter view DBA.second_baseline_view set hidden ;
     grant select on DBA.second_baseline_view to NonPrivUser ;
     grant select on DBA.second_baseline_view to PrivUser ;


--   Alternative 2:
--    Place the encryption function invocation within a user-defined
--   function (UDF).
--    Hide the definition of the UDF. Restrict the UDF permissions.
--   Use the UDF in a view that handles the remainder of the security
--     and business logic.
--    Note: The view itself does not need to be hidden.

     create function DBA.second_decrypt_function(IN datum
varbinary(32))
            RETURNS char(16) DETERMINISTIC
            BEGIN
                RETURN aes_decrypt(datum,'seCr3t', char(16));
            END  ;

     grant execute on DBA.second_decrypt_function to PrivUser ;
     alter function DBA.second_decrypt_function set hidden ;

     create view

DBA.second_decrypt_view(sensitiveA,publicP,publicQ,publicR)
        as
                select
               IF group_member('high_privileges_group',user_name())
= 1
                     THEN second_decrypt_function(encryptedA)
                     ELSE NULL
```

```
                  ENDIF,
                  publicP,
                 publicQ,
                 publicR
                from DBA.second_table ;

     grant select on DBA.second_decrypt_view to NonPrivUser ;
     grant select on DBA.second_decrypt_view to PrivUser ;

--  Alternative 3:
--    Sequester only the key selection in a user-defined function.
--    This function could be extended to support selection of any
--  number of keys.
--    This UDF is also hidden and has restricted execute privileges.
--   Note: Any view that uses this UDF therefore does not compromise
-- the key values.

     create function DBA.second_key_function()
              RETURNS varchar(32) DETERMINISTIC
              BEGIN
                return 'seCr3t' ;
              END

     grant execute on DBA.second_key_function to PrivUser ;
     alter function DBA.second_key_function set hidden ;

     create view
DBA.second_key_view(sensitiveA,publicP,publicQ,publicR)
                as
                    select
                      IF
group_member('high_privileges_group',user_name()) = 1
                         THEN
aes_decrypt(encryptedA,second_key_function(),
                         char(16))
                         ELSE NULL
                      ENDIF,
                      publicP,
                      publicQ,
                      publicR
                    from DBA.second_table ;

     grant select on DBA.second_key_view to NonPrivUser ;
     grant select on DBA.second_key_view to PrivUser ;


--  Alternative 4:
--    The recommended alternative is to separate the security logic
--   from the business logic by dividing the concerns into two views.
--    Only the security logic view needs to be hidden.
--   Note: The performance of this approach is similar to that of the
first
-- alternative.

     create view
```

```
DBA.second_SecurityLogic_view(sensitiveA,publicP,publicQ,publicR)
            as
                select
               IF group_member('high_privileges_group',user_name())
= 1
                     THEN aes_decrypt(encryptedA,'seCr3t', char(16))
                     ELSE NULL
                  ENDIF,
                  publicP,
                  publicQ,
                  publicR
                from DBA.second_table ;

     alter view DBA.second_SecurityLogic_view set hidden ;

     create view

DBA.second_BusinessLogic_view(sensitiveA,publicP,publicQ,publicR)
            as
                select
                  sensitiveA,
                  publicP,
                  publicQ,
                  publicR
                from DBA.second_SecurityLogic_view ;

     grant select on DBA.second_BusinessLogic_view to NonPrivUser ;
     grant select on DBA.second_BusinessLogic_view to PrivUser ;

-- End of encryption example
```

See also

- *AES_ENCRYPT Function [String]* on page 5
- *AES_DECRYPT Function [String]* on page 6
- *LOAD TABLE ENCRYPTED Clause* on page 7

Index