# **Encrypted Columns in Sybase IQ**

Document ID: DC00467-01-1270-01

Last revised: June 2006

This document describes the Sybase IQ Encrypted Column option.

Topic	Page
Introduction to column encryption	2
Data types for encrypted columns	4
AES_ENCRYPT function [String]	6
AES_DECRYPT function [String]	8
LOAD TABLE ENCRYPTED clause	10
Encryption and decryption examples	12
Setting database options for column encryption	22
Protecting ciphertext data from accidental truncation	22
Preserving ciphertext integrity	22
Preventing misuse of ciphertext	23

Copyright 1991-2006 by Sybase, Inc. All rights reserved. Sybase, SYBASE (logo), ADA Workbench, Adaptive Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Enterprise, Adaptive Server Enterprise, Maptive Server Enterprise Replication, Adaptive Server Enterprise, Replication, Adaptive Server Enterprise, Adaptive Server Certified PowerBuilder Developer, Certified SYBASE Professional, Certified SYBASE Professional Logo, ClearConnect, Client-Library, Client Services, CodeBank, Column Design, ComponentPack, Connection Manager, Con voy/DM, Copernicus, CSP, Data Pipeline, Data Workbench, DataArchitect, Database Analyzer, DataExpress, DataServer, DataWindow, DataWindow, NET, DB-Library, dbQueue, Dejima, Dejima Direct, Developers Workbench, DirectConnect Anywhere, DirectConnect, Distribution Director, Dynamic Mobility Model, e-ADK, E-Anywhere, e-Biz Integrator, E-Whatever, EC Gateway, ECMAP, ECRTP, eFulfillment Accelerator, EII Plus, Electronic Case Management, Embedded SQL, EMS, Enterprise Application Studio, Enterprise Client/Server, Enterprise Connect, Enterprise Data Studio, Enterprise Manager, Enterprise Portal (logo), Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, eProcurement Accelerator, eremote, Everything Works Better When Everything Works Together, EWA, Extended Assist, Extended Systems, Extended View, Financial Fusion, Financial Fusion (and design), Financial Fusion Server, Formula One, Fusion Powered e-Finance, Fusion Powered Financial Destinations, Fusion Powered STP, Gateway Manager, GeoPoint, GlobalFIX, i Anywhere, iAnywhere Solutions, ImpactNow, Industry Warehouse Studio, InfoMaker, Information Anywhere, Information Everywhere, Information Connect, InstaHelp, Intelligent Self-Care, InternetBuilder, iremote, irLite, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, Legion, Logical Memory Manager, M2M Anywhere, Mach Desktop, Mail Anywhere Studio, Mainframe Connect, Maintenance Express, Manage Anywhere Studio, MAP, M-Business Anywhere, M-Business Channel, M-Business Network, M-Business Suite, MDI Access Server, MDI Database Gateway, media.splash, Message Anywhere Server, MetaWorks, MethodSet, mFolio, Mirror Activator, ML Query, MobiCATS, MobileQ, MySupport, Net-Gateway, Net-Library, New Era of Networks, Next Generation Learning, Next Generation Learning Studio, O DEVICE, OASIS, OASIS logo, ObjectConnect, ObjectCycle, OmniConnect, OmniQ, OmniSQL Access Module, OmniSQL Toolkit, OneBridge, Open Biz, Open Business Interchange, Open Client, Open Client, Open Client/Server, Open Client/Server Interfaces, Open Gateway, Open Server, Open Server, Open Solutions, Optima++, Partnerships that Work, PB-Gen, PC APT Execute, PC DB-Net, PC Net Library, Pharma Anywhere, PhysicalArchitect, Pocket PowerBuilder, PocketBuilder, Power+, Power-Through Knowledge, power.stop, PowerAMC, PowerBuilder, PowerBuilder Foundation Class Library, PowerDesigner, PowerDimensions, PowerDynamo, Powering the New Economy, PowerScript, PowerScript, PowerScript, PowerScript, PowerScript, PowerScript, PowerScript, PowerScript, PowerDimensions, PowerTips, PowerDimensions, PowerDynamo, PowerIng the New Economy, PowerScript, PowerScript, PowerScript, PowerScript, PowerScript, PowerScript, PowerDimensions, PowerTips, PowerDimensions, PowerDime ersoft Portfolio, Powersoft Professional, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, Pylon, Pylon Anywhere, Pylon Application Server, Pylon Conduit, Pylon PIM Server, Pylon Prof. QAnywhere, Rapport, Relational Beans, Remote Ware, RepConnector, Report Workbench, Report-Execute, Replication Driver, Replication Driver, Replication Server Manager, Replication Toolkit, Resource Manager, RFID Anywhere, RW-Display-Lib, RW-Library, SAFE, SAFE/PRO, Sales Anywhere, Search Anywhere, SDF, Search Anywhere, Secure SQL Server, Secure SQL Toolset, Security Guardian, ShareSpool, ShareLink, SKILS, smart.partners, smart.parts, Smart.script, SOA Anywhere Trademark, SQL Advantage, SQL Anywhere, SQL Anywhere Studio, SQL Code Checker, SQL Debug, SQL Edit, SQL SQL SMART, SQL Toolset, SQL ServerCFT, SQL ServerDBM, SQL Server SNMP SubAgent, SQL Station, SQLJ, Stage III Engineering, Startup.Com, STEP, SupportNow, S.W.I.F.T. Message Format Libraries, Sybase Central, Sybase Client/Server Interfaces, Sybase Development Framework, Sybase Financial Server, Sybase Gateways, Sybase Ig. Sybase Learning Connection, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase Virtual Server Architecture, Sybase User Workbench, SybaseWare, Syber Financial, SyberAssist, SybFlex, SybMD, SyBooks, System 10, System II, System XI (logo), SystemTools, Tabular Data Stream, The Enterprise Client/Server Company, The Extensible Software Platform, The Future Is Wide Open, The Learning Connection, The Model For Client/Server Solutions, The Online Information Center, The Power of One, TotalFix, TradeForce, Transact-SQL, Translation Toolkit, Turning Imagination Into Reality, UltraLite, UltraLite, NET, UNIBOM, Untill, Unisep, Unistring, URK Runtime Kit for UniCode, Viafone, Viewer, VisualWriter, VQL. Warehouse Architect, Warehouse Control Center, Warehouse Studio, Warehouse WORKS, Watcom, Watcom SQL, Server, Web Deployment Kit, Web.PB, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, XA-Server, XcelleNet, XP Server, XTNDAccess and XTNDConnect are trademarks of Sybase, Inc. or its subsidiaries. 05/06

## Introduction to column encryption

Sybase IQ 12.6 GA introduced support of strong encryption of the database file for greater security. Strong encryption is achieved through the use of 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 12.7 supports user encrypted columns with the addition of the AES\_ENCRYPT and AES\_DECRYPT functions and the LOAD TABLE ENCRYPTED clause. These functions permit explicit (via calls from the application) encryption and decryption of column data. Encryption and decryption key management is the responsibility of the application.

Users must be specifically licensed to use the encrypted column functionality of the Sybase IQ Encrypted Column Option described in this product documentation.

Certain database options affect column encryption. Before using this feature, see "Setting database options for column encryption" on page 22.

The following terms are 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 IQ interfaces accept character strings as keys.

**Rijndael** Pronounced "reign dahl." A specific encryption algorithm supporting 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.

Definitions

**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 IQ.

#### Data types for encrypted columns

This section lists the supported and unsupported data types for encrypted columns and discusses the preservation of the original data type of an encrypted column.

Supported data types

The first parameter of the AES\_ENCRYPT function must be one of the following supported data types:

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

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

Preserving 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. 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 the following valid decryption statement:

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

If you attempt to decrypt the data using the following statement:

```
SELECT AES_DECRYPT ( thecolumn, 'theKey', SMALLINT ) FROM thetable
```

the decryption 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 the following 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 the following statement:

```
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.

## **AES\_ENCRYPT** function [String]

**Function** 

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. For a list of supported data types, see "Data types for encrypted columns" on page 4. Binary values can also be passed to this function. This parameter is case sensitive, even in case-insensitive databases.

**key** The encryption key used to encrypt the *string-expression*. This same key must be used to decrypt the value in order to obtain the original 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, and includes numbers, letters, and special characters. You will require this key each time you want to decrypt the data.

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

Usage

This function 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. In order to decrypt a *string-expression* successfully, you must use the same encryption key and algorithm that were 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 VARBINARY, so that character set conversion is not performed on the data. (Character set conversion would prevent decryption of the data.)

Standards and compatibility

- **SQL92** Vendor extension.
- SQL99 SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

"AES\_DECRYPT function [String]" on page 8

"LOAD TABLE ENCRYPTED clause" on page 10

Example

See "Encryption and decryption examples" on page 12 for an example of the use of the AES\_ENCRYPT function.

## **AES\_DECRYPT function [String]**

**Function** 

Decrypts the string using the supplied key and returns a VARBINARY or LONG VARBINARY by default, 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*. This must be the same encryption key that was used to encrypt the *string-expression* in order to obtain the original value that was encrypted. This parameter is case sensitive, even in case-insensitive databases.

**Warning!** Protect your key. Be sure to 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 data type of the original plaintext.

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.

In order 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.

Standards and compatibility

- SQL92 Vendor extension.
- SQL99 Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

Usage

- "AES\_ENCRYPT function [String]" on page 6
- "Encryption and decryption examples" on page 12
- "LOAD TABLE ENCRYPTED clause" on page 10

Example

The following example decrypts the password of a user from the user\_info table.

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

#### LOAD TABLE ENCRYPTED clause

The LOAD TABLE statement now supports the new 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

See "Example" on page 11.

Syntax

**Parameters** 

| ENCRYPTED(data-type 'key-string' [, 'algorithm-string' ])

**data-type** The data type that the input file field should be converted to as input to the AES\_ENCRYPT function. For supported data types, see "Data types for encrypted columns" on page 4. This should be the same data type as the data type of the output of the AES\_DECRYPT function. See "AES\_DECRYPT function [String]" on page 8.

**key-string** The encryption key used to encrypt the data. This key must be a string literal. This same key must be used to decrypt the value in order to obtain the original 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, and includes numbers, letters, and special characters. You will need this key each time you want to decrypt the data.

**Warning!** Protect your key. Be sure to store a copy of your key in a safe location. A lost key will result 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 only supported algorithm and it is used by default. 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.

See also

- "AES ENCRYPT function [String]" on page 6
- "AES\_DECRYPT function [String]" on page 8
- "Encryption and decryption examples" on page 12

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 '\n'
QUOTES OFF
ESCAPES OFF
```

## **Encryption and decryption examples**

Example 1

The following example of the AES\_ENCRYPT and AES\_DECRYPT functions is written in commented SOL.

```
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 either sensitiveA
   or 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 uses 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;
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) ' ||
             ' 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
            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
               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
                       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)
             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 ;
      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
```

#### Example 2

The ciphertext produced by the AES\_ENCRYPT function 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 the following:

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

The value AES\_ENCRYPT(c1, 'key') will differ from AES\_ENCRYPT(c2,'key') and the value AES\_ENCRYPT(c1,'key') will differ from AES\_ENCRYPT(100,'key').

To resolve this issue, cast the input of the AES\_ENCRYPT function to the same data type. For example, the following will be the same:

```
AES_ENCRYPT(c1, 'key');
AES_ENCRYPT(CAST(c2 AS INT), 'key');
AES_ENCRYPT(CAST(100 AS INT), 'key')
```

## Setting database options for column encryption

Certain Sybase IQ database option settings affect column encryption and decryption. Check the options mentioned in this section before using the AES\_ENCRYPT or AES\_DECRYPT function because the default settings are not optimal for most column encryption operations.

#### Protecting ciphertext data 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 the following database option:

```
SET OPTION STRING RTRUNCATION = 'ON'
```

When STRING\_RTRUNCATION is set ON, the engine raises an error whenever a string would be truncated during a load, insert, update, or SELECT INTO operation. This is ANSI/ISO SQL92 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 (the default) forces silent truncation of strings.

The AES\_DECRYPT function itself also checks the 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.)

#### Preserving ciphertext integrity

To preserve ciphertext integrity, set the following database option:

```
SET OPTION ASE BINARY DISPLAY = 'OFF'
```

When ASE\_BINARY\_DISPLAY is set OFF, the system leaves binary data unmodified in its raw binary form.

When ASE\_BINARY\_DISPLAY is set ON (the default), the system converts binary data into its hexadecimal string display representation. Set the option temporarily to ON only in those circumstances when you need data to display to an end user or when you need to export the data to another external system, where raw binary could become altered in transit.

#### **Preventing misuse of ciphertext**

The CONVERSION\_MODE database option restricts implicit conversion between binary data types (BINARY, VARBINARY, and LONG BINARY) and other non-binary data types (BIT, TINYINT, SMALLINT, INT, UNSIGNED INT, BIGINT, UNSIGNED BIGINT, CHAR, VARCHAR, and LONG VARCHAR) on various operations. The use of this option prevents 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 non-binary data type on INSERT, UPDATE, 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 prior to version 12.7.

For more information on the CONVERSION\_MODE option and its usage, see "CONVERSION\_MODE option" in Chapter 2, "Database Options" of the *Sybase IQ Reference Manual*.

