

Database Encryption Design Considerations and Best Practices for ASE 15

BY JEFFREY GARBUS, SOARING EAGLE CONSULTING

EXECUTIVE OVERVIEW

This article will explore best practices and design considerations in column-level encryption as practiced with Sybase® Adaptive Server® Enterprise 15 (ASE 15). It's intended to be a great read prior to designing an encryption scheme for your secure-server-to-be. There are a variety of decisions you're going to need to make, from encryption options to storage of encryption keys. We'll discuss pros & cons and make recommendations based upon some of your security business decisions.

INTRODUCTION

At this point, you've already determined that you want to encrypt the data in your database. You have a handle on basic data encryption techniques. So, you don't need to be told that encryption performs two purposes: protection of data against internal prying eyes, as well as protection of data against external threats (hacking, theft of backup tapes, etc.). You also don't need a discussion on the merits of performing encryption in the database tier, where you have database-caliber performance and protection of encryption keys (among other things) rather than incur the overhead and additional cost of using a 3rd-party encryption tool in an application tier. And, you don't need to be told that you need to install the ASE_ENCRYPTION license option, enable the encryption configuration option, and create keys.

But, you may want to know what design considerations will affect storage and performance. In this article, we're going to explore best practices and design considerations in column-level encryption as practiced with Sybase Adaptive Server Enterprise 15 (ASE 15). If you need a more basic overview, or a basic discussion of why and how we encrypt, read the precursor article, "*Column-level encryption in ASE 15.*" There is also great information in the *ASE Systems Administration Guide* and the *Encryption Guide*, available with online books.

This article is intended to be a great read prior to designing an encryption scheme for your secure-server-to-be. There are a variety of decisions you're going to need to make, from encryption options to storage of encryption keys. We'll discuss pros & cons and make recommendations based upon some of your security business decisions.

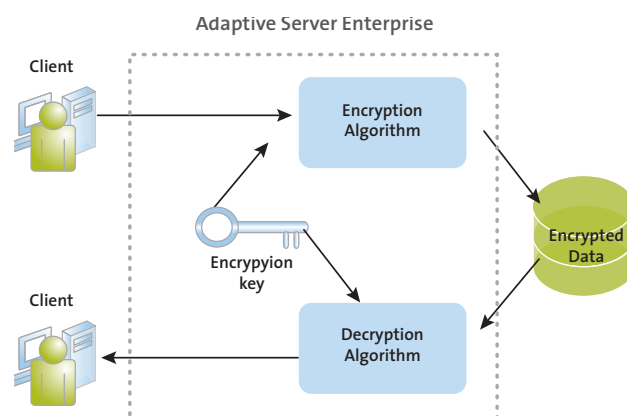
BRIEF REVIEW OF DATA ENCRYPTION CONCEPTS


First, let's make sure we're on the same page from a terminology and process standpoint.

There are two basic components to data encryption: Encryption, wherein we store and protect the data, and decryption, with which we retrieve and unscramble the data.

On the encryption side, you need to be able to create and manage an encryption key, as well as the ability to set permissions for logins who should have access. On the decryption side, you need to be able to transparently select from the appropriate columns.

Figure 1: Encryption and Decryption in Adaptive Server





The Key Custodian (this is a role initially granted to the System Security Officer – SSO) uses the *create encryption key command* to create an encryption key. The server may contain one encryption key for the server, one per database, one per column, or any combination thereof. (This will be one of your early decisions; more on that later.) The server uses a symmetric encryption algorithm, which means that the same encryption key is used for both encryption and decryption.

When the client process accesses an encrypted column, the server knows which encryption key was associated with the column (it is stored in the *sysencryptkeys* table in each database). At insert or update time, the column is transparently encrypted immediately before writing the row. At select time, the server decrypts the data immediately after reading the data. This is all done transparently to the user, if the user has both select and decrypt permissions on the column.

From a utilization standpoint, to enable encryption, you follow these steps:

- 1) Install the license option (encryption does not come standard with ASE)
- 2) Enable encryption using *sp_configure 'enable encrypted columns',1*
- 3) Set the system encryption password for the database in which you're going to store the keys (more on this later, you can do this for each database, or store them all together)
- 4) Create the column encryption key(s) (CEK) using *create encryption key ...* you may end up creating one or many. The CEK is used in conjunction with a system encryption password (SEP), a database-specific password which is set (or reset) by the SSO. It is unique per database which contains the SEK. In other words, if the keys are all stored in one database (more on that later), the SSO only needs to set an SEP in one database.
- 5) Specify column(s) to be encrypted (and with which keys)
- 6) Grant decrypt permission to users authorized to use the information. If the users have select permission only, they will get gibberish back

DESIGN CONSIDERATIONS

We're going to start off with design consideration in support of the basic scheme we've described.

First, here's a list of the things you need to decide / know before embarking upon an encryption solution:

- 1) Where are you going to keep the CEKs? Choices include in each database, or in a separate database, or some combination (potential dump/load issues here).
- 2) How many CEKs are you going to keep? One per column? One per database? One per server? One per type of column you're going to encrypt (we'll go into detail about the limitations of individual encryption types)?
- 3) What size CEK are we going to use? Choices are 128-192-256 bits; the bigger the key, the higher CPU cost of encryption.
- 4) Will we use an init vector or pad the CEKs?
- 5) Are we going to search/join/create relational integrity on (and, of course, correspondingly index) any encrypted columns?
- 6) How often (and/or) will we change encryption keys? This requires the reencryption of all data that uses the key.

Don't answer yet; first, we'll go into ramifications of each decision.

CREATION OF ENCRYPTION KEYS

Column encryption in ASE uses Advanced Encryption Standard (AES) with 128, 192, or 256-bit encryption key sizes. The longer the bit string, the more difficult it is for a hacker to decrypt. On the other hand, the more complicated the encryption, the more CPU resources will be taken up by the encryption/decryption algorithm.

Which level is right for you? That's really a question for your site security officer. You should be aware that most public, commercial products/projects are using 128-bit encryption, and the government uses that up to the SECRET level; TOP SECRET requires 192 or 256-bit encryption. How difficult is this to crack? It's been calculated that cracking a 128-bit algorithm requires 2^{120} (2 to the 120th power) operations, which is currently not considered feasible (Source: Wikipedia).

That said, 128-bit encryption seems secure enough for most applications. But, check with your security officer about your shop standard. If you go with a higher level, be sure to benchmark the effect of the higher level of encryption against CPU utilization.

To securely protect key values, ASE uses the system encryption password (set up by the Key Custodian) in conjunction with random values to generate a 128-bit key-encryption key (KEK). The KEK is in turn used to encrypt (prior to storage) all of the CEKs you create.

Syntax:

```
create encryption key keyname [as default] for algorithm
[with [keylength num_bits]
[init_vector [null | random]]
[pad [null | random]]]
```

Where:

keyname	must be unique in the user's table, view, and procedure name space in the current database.
as default	allows the Key Custodian to create a database default key for encryption. This CEK will be used when a table creator fails to specify a CEK name with encryption.
algorithm	Advanced Encryption Standard (AES) is the only algorithm supported.
keylength num_bits	Valid key lengths are 128, 192, and 256 bits, with a default of 128 bits.
init_vector	Random/null Instructs the server to randomly pattern an initialization string, so that the ciphertext of two identical pieces of plaintext are different. Use this to prevents detection of data patterns . the catch: You can create indexes and optimize joins and searches only on columns where the encryption key does not specify an initialization vector. The default is to use an initialization vector, that is, init_vector random.
pad	Random/null The default, null, omits random padding of data, and supports the use of an index.

When random, data is automatically padded with random bytes before encryption. You can use padding instead of or in addition to an initialization vector to randomize the ciphertext (the underlying data, stored as varbinary).

Initialization vectors (init_vector) and column padding (pad) might be a reason that you have multiple CEKs. You might have one which uses the default, random vector, and another which you use for searchable strings.

You may also have multiple columns in a single table which might require a higher level of security, or which might need to be searchable.

USING CEKS

The encryption key is referenced at table creation time (or via the alter table, which will dynamically encrypt all of the target data). You may reference a CEK in the local database, or a remote database. After the CEK is created, you will have to grant select permission on the key to the group (role) that will be creating/altering the tables.

Local vs. remote database keys

Encrypting in a database other than the secure database provides an additional layer of security. If the database dump is ever stolen, the encryption key is nowhere to be found. In addition, the administrator or operator can password protect the database dump, making things that much harder for a hacker.

The flip side is that if you are storing CEKs in a different database, you must make sure you synchronize your database dumps. In fact, you might consider treating the CEK database as another master database, dumping it each time a change occurs. Keep in mind this goes in the opposite direction too... you want to dump the "data" database if the changed "CEK" database gets dumped, for consistency's sake.

More on dumps & loads

To prevent accidental loss of keys, the *drop database* command fails if the database contains keys currently used to encrypt columns in other databases. Before dropping the database containing the encryption keys, first remove the encryption on the columns by using alter table, then drop the table or database containing the encrypted columns.

Store database dumps and the dumps of the key databases in separate physical locations. This prevents loss in case an archive is stolen.

Column support

ASE 15 supports the encryption of int, smallint, tinyint, unsigned int, smallint, tinyint, float4, float8, decimal, numeric, char, varchar, binary and varbinary datatypes. Note that null values are not encrypted.

Cross platform encryption

Cross platform support is available if both platforms use the same character set (and if you have the encryption key); will need the encryption key. In other words, you can decrypt data on a platform other than the platform on which encryption took place. This is useful for bcp, among other things. (i.e. test/development systems, where you might move databases over, including encryption key database(s), and give select but not decrypt permissions to the testers/developers).

Changing encryption keys

You will likely make a security decision to change encryption keys on a periodic basis. This is implemented easily with the *alter table* command.

Because changing the encryption key requires decryption from the old key and corresponding reencryption, you'll want to set aside plenty of time for the activity. (i.e. table scan + encryption time)

This is likely to preclude frequent CEK changes in large-scale and/or high volume production environments.

A Final Design Note

Encrypted columns take up more storage space because of the data column changes which add 16-byte encryption information + offset lengths. If encrypted data has a significant number of rows, be sure to take this into account at capacity planning time.

PERFORMANCE IMPLICATIONS OF ENCRYPTION

Encryption is CPU-intensive. How CPU-intensive is something you're going to have to benchmark on your own application/hardware/load combination, but will vary based upon:

- Number of CPUs
- ASE engines
- System load
- Concurrent sessions
- Encryption per session
- Encryption key size
- Length of data

In general, it's hard to say that the larger the key size and the wider the data, the higher your CPU utilization will be. As a result, you will want to ensure that you only encrypt columns that require the extra security. There are other considerations other than encryption and decryption of data.

INDEXES ON ENCRYPTED COLUMNS

ASE avoids table scans by using an index to access data. If you are searching for data based upon the content of an encrypted column, as opposed to simply retrieving the information, we have to be cognizant of a few things:

- 1) The index lookups are efficient because they look up and compare ciphertext values.
- 2) Noting that encryption happens when data goes in or comes out, things like range searches would end up using the index to get a range of ciphertext, rather than data, which is less than useful. But, indexes on the encrypted data are just fine for equality/inequality searches.
- 3) Same with sorts: any sort of encrypted data is going to incur additional overhead because the data needs to be decrypted prior to the sort. In other words, the index will not help avoid a sort).
- 4) In order to index the columns, the encryption key must have been created with both `init_vector` and `pad` (random padding) turned off.
- 5) Joins of encrypted columns are optimized if the following are true:
 - a. Indexing is permitted (i.e. `init_vector` and `pad` set to NULL).
 - b. The joining columns have the same datatype. Char and varchar are considered to be the same datatypes, as are binary and varbinary.
 - c. For int and float types, the columns have the same length.
 - d. For numeric and decimal types, the columns have the same precision and scale.

- e. The joining columns are encrypted with the same key.
 - f. The joining columns are not part of an expression.
 - g. The join operator is '=' or '<>'.
 - h. The data has the default sort order.
- 6) In order to use an index, we have to have a search argument (sarg), which is a where clause of the format *encrypted_column operator constant*.
- 7) You can define referential integrity constraints between two encrypted columns when:
- a. Indexing is permitted (i.e. init_vector and pad set to NULL).
 - b. Both referencing and referenced columns are encrypted.
 - c. The referenced and referencing column are encrypted with the same key.

Referential integrity checks are efficient because they are performed on ciphertext values.

RETURNING DEFAULT VALUES

Protection exceptions are sloppy, unless your business requirement specifically states that it wants an exception thrown. In order to avoid protection errors, you simply set up a default at table creation time.

```
create table secure_table (ssnum char(11) encrypt with Key1
decrypt_default 'If you get caught looking at this data you are out of a job')
```

It's that easy.

RECOMMENDED PRACTICES & OTHER RECOMMENDATIONS

- CEK selection (absent indexing needs)

Columns characteristics	Recommended encryption key properties
Low cardinality data	Key with Initialization Vector or random padding
High cardinality data (SSN, Phone#, Credit Card#)	Key without Initialization Vector and random padding
Primary key columns and indexed columns	Key without Initialization Vector and random padding
Foreign key columns	Same key as referenced primary key (fully qualified name of the key should match)
Columns used in joins	Same key without Initialization Vector and random padding

- You might also consider using a single key for a single “type” of data. For example, a social security number might get the same key regardless of the table it's in. That way, if you have a specific standard “Change SSN keys monthly” you know where to start looking.
- Key locations
 - Keep keys in separate databases, so that a stolen database doesn't contain the decryption key
 - Make sure dumps of CEK databases and the data itself are synchronized
- Change keys periodically
 - Makes key attacks harder
- Encrypt data during movement
 - Use BCP with the –c option to keep the ciphertext in character format
 - Alternatively, use replication
 - Also, use SSL to send data
- Beware impact of range searches (including foreign key searches) on encrypted data
 - Consider using a (noncompromising) prefix, rather than the entire key, as an unencrypted, indexed, searchable column

- As the output may contain secure data, disable/do not use capture:
 - monSysSQLText and monProcessSQLText
 - query metrics capture
 - statement cache
 - monitor server
 - dbcc sqltext
- Audit access to tables with encrypted data
- Consider using central key management, and replicating keys out to CEK database on the target servers
- Protect keys using explicit passwords. Do not give key custodians select permission on encrypted data
- Use the decrypt default feature to preserve application transparency with stored procedures which may start returning scrambled results from suddenly-encrypted columns

CONCLUSION

Database encryption is in and of itself fairly cookbook. You install the encryption option; enable it; create a system encryption password; a system encryption key; and then you create or alter a table column to use the encryption.

It is also fairly flexible; you can create a system-wide encryption key, or a separate encryption key for every column you encrypt.

There are a several options to consider when encrypting data, and these decisions are going to be driven based upon your security needs and procedures, in conjunction with your knowledge of resource on your server, as well as your performance needs.

For example, using 128-bit encryption is probably more than adequate, but if you're going to use 256-bit encryption, make sure your CPU capacity is up to it prior to rolling this set of decisions out to production. Or, if you are going to be searching on an encrypted column, you need to make sure you can index that column, and that limits the type of encryption key you can use on that column.

Jeff Garbus has 20 years of expertise in architecture, tuning and administration of Sybase ASE, Oracle, and Microsoft SQL Server databases with an emphasis on assisting clients in migrating from existing systems to pilot and enterprise projects. He has co-authored 15 books and has published dozens of articles on the subject. Mr. Garbus is the CEO of Soaring Eagle Consulting, an organization that specializes in assisting businesses maximize database performance www.soaringeagle.biz.