

# Encrypted Columns in ASE 15

By Jeffrey Garbus – Soaring Eagle Consulting

## EXECUTIVE SUMMARY

In today's security conscious world, you need to know that critical business and personal information is protected from prying eyes. In fact, often you need to demonstrate to your own customers that you've taken steps to guarantee their account information. With Column-level encryption, Sybase ASE 15 takes encryption out of the middle tier where it's a potential performance problem and puts it in the database layer where it belongs. This article discusses the basics of why and how you encrypt data in ASE 15.

## INTRODUCTION

In the early days of Sybase Adaptive Server Enterprise (ASE), there was one level of security: either permission is granted on a column, or it is not. If it is granted, you can query the data with a select statement; if permission is not granted, you may not query the data with a select statement.

This was the mechanism with which Sybase protected login passwords. Server logins all had permission to use the master database, and select from the syslogins table, but would not have permission to select from the password column; therefore the password was protected; wasn't it?

There are two catches here. First, the system administrator (SA) had permission to select from that column; the SA has permission to do most everything on the server. The second problem was that a dump or database file which was stolen from the company could be mined, and the password pulled from that dump.

Sybase's answer 15 years ago is the answer that has become commonplace today: The data in the column is encrypted. Now, the SA can still select data from the column, but without the encryption key, the data is meaningless.

Data encryption meets both needs in data security: It protects data from "friendly" eyes, as well as "unfriendly" eyes, who have perhaps stolen a backup tape.

With the advent of ASE 12.5.4, Sybase gave customers the ability to create encryption keys and encrypt any other data in the database(s). This met an immediate market need, because there had been a lot of high-profile cases in the news where databases had been stolen, and this had led to identity theft on a massive scale. With ASE 15, encryption got better & easier, with the ability to recover lost encryption keys, specify default values for data a user has no permission for, and additional datatypes that may be encrypted. (Note: this article focuses on ASE 15, if you are still on 12.5.x and care about encryption, it is probably worth upgrading for this additional functionality in and of itself.)

There are / have been third-party products to perform the encryption, but performing the encryption at the database level is an improvement in performance and reduces the need for a security tier (don't we already have enough tiers in our environments?).

## DATA ENCRYPTION IN ASE 15

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.

### Encrypting the Data – the Encryption Key

ASE 15 introduces several new concepts, which work together to create a consistent and usable encrypted data security model. The primary focus is the management of and access to an encryption key.

<sup>1</sup> You may ask why this is a bad thing. After all, the SA has permission to do anything / change anything on the server anyway. It was nicely explained to me (about 18 years ago) by a Pentagon employee attending a Systems Administration class I was teaching that people tend to use the same password for all their systems; so, if you are able to read folks' passwords on Sybase, you may have access to their email, etc., as well. Lesson learned.

Note that ASE manages the security of keys by keeping keys encrypted. There are actually two keys between the user and the data: the column encryption key (CEK) and the key encryption key (KEK). The CEK encrypts data and users must have access to it before they can access the encrypted data. For security it is stored in encrypted form. ASE encrypts the CEK with a KEK when you create or alter an encryption key. The KEK is also used to decrypt the CEK before you can access decrypted data. The KEK is derived internally from the system encryption password, a user-specified password, or a login password, depending on how you specify the key's encryption with the create and alter encryption key statements. CEKs are stored in encrypted form in sysencryptkeys. ASE 15 provides the ability to move keys between systems in a secure manner.

### **Key Custodian**

Somebody needs to manage the encryption keys, which includes creating, dropping, and modifying them, distributing passwords, and providing for key recovery in the event of a lost password (important feature!).

In order to separate the management of the encryption key from the day-to-day administration of security (logins, etc.), Sybase has introduced the concept of Key Custodian. The keycustodian\_role is automatically granted to the sso\_role. It can also be granted to any login by a user with the sso\_role. You can have multiple key custodians, who each own a set of keys.

The key custodian can:

- Create and alter encryption keys
- Assign a database default key a key
- Set up key copies for designated users, allowing each user access to the key through a chosen password or a login password
- Share key encryption passwords
- Grant schema owners select access to encryption keys
- Set the system encryption password
- Recover encryption keys
- Drop encryption keys they own
- Change ownership of keys they own

There are three options for passwords, a system password, an encryption key password, and a login password.

### **System Encryption Password**

The key custodian sets the system encryption password using:

```
sp_encryption system_encr_passwd, 'password'
```

Using a system encryption password simplifies the administration of encrypted data because:

- Key management is restricted to setting up and changing the system encryption password
- You need not specify passwords on create and alter encryption key statements
- Password distribution and recovery from lost passwords are not required
- Privacy of data is enforced through decrypt permission on the column
- Restricted decrypt permission reinforces this privacy against the power of the administrator

The downside to System encryption passwords is that you may need to synchronize passwords across multiple systems, such as in a Replication Server environment. There are vulnerabilities, and given that different people may have the keycustodian\_role, we might not want both arbitrarily able to change the system encryption password, as this could add a layer of confusion.

### **User-Specific Encryption Passwords**

A login with the keycustodian\_role or the sso\_role can restrict access to data even from the SA or DBO by specifying passwords on keys through the create encryption key or alter encryption key statements.

If keys have explicit passwords, users need Decrypt permission on the column and the encryption key's password. Users must also have knowledge of the password to run DML commands that encrypt data.

Use the *create encryption key command* to associate a password with a key:

```
create encryption key [db.[owner].]keyname [as default]
[for algorithm_name] [with {[keylength num_bits]
[passwd 'password_phrase'] [init_vector {NULL | random}]
[pad {NULL | random}]]}
```

- for algorithm\_name – (optional for the 15.0.2 release) specifies the algorithm you are using. The default is the Advanced Encryption Standard (AES) algorithm
- password\_phrase – is a quoted alphanumeric string of up to 255 bytes in length that ASE uses to generate the KEK.

Note that ASE doesn't save the password. Instead, it saves a string of validating bytes known as the "salt" in sysencryptkeys.eksalt, which allows it to recognize whether a password used on a subsequent encryption or decryption operation is legitimate for a key.

This example shows how to use passwords on keys, and the key custodian's function in setting up encryption. Here, the password would be shared among all the users.

- 1) Key custodian Jeff creates an encryption key:  
create encryption key key1 with passwd 'Th1s1smyP@ssword'
- 2) The Key custodian distributes the password to all users who need access to encrypted data
- 3) Each user enters the password before processing tables with encrypted columns:  
set encryption passwd 'Th1s1smyP@ssword' for key jeff.key1
- 4) If somebody quits, or the password is hacked / guessed / business rules require us to change the key, key custodian Jeff alters the key to change the password.

Alter encryption key key1 with passwd 'Th1s1smyNewP@ssword'

Note: all this changes is the KEK. The data is not decrypted and re-encrypted as when you change a key. Only the KEK is changed so only the key is re-encrypted. Also note that this method requires an application change as the application needs to be able to submit the 'set encryption passwd...' phrase.....and any associated restrictions/issues with app servers and 'set proxy/session\_authorization'.

### Encryption Summary

The SSO starts out as a Key Custodian, and can create additional Key Custodians. The Key Custodian is able to manage Keys, but not necessarily see the data, as he may not have select permission on the target tables, but only permission on the keys.

Keys may be protected with passwords at the system level, which become part of individual users' own passwords, or they may be protected with user-specific passwords, which may then be shared at the user level.

### Decrypting (querying) the Data

Once the data has been encrypted using keys with user-defined passwords, several things have to happen in order for a user to see data in the clear:

- You need select permission on the column to read the information, insert/update/delete permission (as appropriate) to modify the information
- You need decrypt permission on the column to read (or if you're going to use it for comparison purposes in a query predicate). If you have select permission, and do not have decrypt permission, you're going to get either the default (if it has been applied) or a permissions error (if no default has been applied)<sup>2</sup>.
- You need to supply a password, unless the system encryption password or a login password.

The encryption password is set using the "set encryption password" syntax above on a per session basis; simple.

This example illustrates how Adaptive Server determines the password when it must encrypt or decrypt data. It assumes that the ssn column in the employee and payroll tables is encrypted with key1, as shown in these simplified schema creation statements:

```
create encryption key key1 with passwd "MyPw2008"
create table customer (
    ssn char (11) encrypt with key1,
    name varchar(30),
    credit_card varchar(20) encrypt with key1)
```

The key custodian shares the password required to access customer.ssn with Penny. He doesn't need to disclose the name of the key to do this.

- 1) If Penny has select and decrypt permission on customer, she can select customer data using the password given to her for customer.ssn:

```
set encryption passwd "MyPw2008" for column customer.ssn
```

```
select name from customer where ssn = '111-22-3456'
name
```

```
-----
Soaring Eagle Consulting, Inc.
```

- 2) Note that even though Penny is not retrieving the column, because she's using it as a reference, it needs to be decrypted.
- 3) If Penny attempts to select data from payroll without specifying the password for customer.ssn, the following select fails (even if Penny has select and decrypt permission on customer)

```
select credit_card from customer where ssn = '111-22-3456'
```

You cannot execute 'SELECT' command because the user encryption password has not been set.

<sup>2</sup> This may lend itself to some interesting programming practices at many shops. For example, many of us write defaults into stored procedures, and test the parameters to see if they have default values, and make decisions based upon the absence of a value. You can do the same thing here, and identify that the column has a default value, identifying that the user does not have permission.

4) To avoid this error, Penny must first enter:

```
set encryption passwd " MyPw2008" for column customer.ssn
```

The key custodian may choose to share passwords on a column-name basis and not on a key-name basis to avoid users hard coding key names in application code, which can make it difficult for the DBO to change the keys used to encrypt the data. However, if one key is used to encrypt several columns, it may be convenient to enter the password once. If one key is used to encrypt several columns and the user is setting a password for the column, they need to set password for all the columns they want to process.

The need to share the passwords and embed it in an application is part of the downfall of this approach - but it could be done via application asking for tokens (i.e. asking for both the key name and the key password from the user) - or by looking it up in a catalog.

The column name approach is not necessarily a great solution - if using an encryption password, an alternative such as a key catalog is viable and would allow keys to be changed without an application change.

### **Key Copies**

Key copies enable users to access encrypted columns using their own copy of a single key. This provides accountability for data because a key copy is designated for an individual user with a private password known only to the user. Without knowledge of the passwords protecting the key and its copies, not even the SA can access the data.

To preserve application transparency, you may encrypt key copies with your login password, avoiding application changes to supply the key copy's password.

### **Using login passwords on key copies**

The easiest way to manage key copies is by associating the password with the user's login. Users whose login password is associated with a key can access encrypted data without supplying a password. This is a good thing, because the fewer passwords a user has to remember, the less likely he is to put it on a post-it note and stick it to his monitor. In addition, applications don't have to prompt for a password.

Sample syntax:

```
alter encryption key [database_name.[owner_name].]key_name  
with password 'base_key_password'  
add encryption for user 'user_name' for login_association
```

### **RECOVERING LOST KEYS**

What happens when a key is lost? Or a Key custodian quits? In short, and without a bundle of syntax, ASE provides the ability to recover from the loss of a user-specified encryption password on a key copy, a login password, or loss or unavailability of password on base key.

### **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.

### **CONCLUSION**

Data is encrypted to keep its contents protected from prying eyes, both from within and without.

ASE 15 provides a detailed and complete approach to encrypting, managing, and retrieving data in a consistent manner, while also creating capabilities to manage and secure keys in a flexible yet secure way.

---

A 20-year veteran of Sybase ASE database administration, design, performance, and scaling, Jeff Garbus has written over a dozen books, many dozens of magazine articles, and has spoken at dozens of user's groups on the subject over the years. He is CEO of Soaring Eagle Consulting, and can be reached at [jeff@soaringeagle.biz](mailto:jeff@soaringeagle.biz).