SQL 2005 Symmetric Encryption

Introduction

One of the most exciting new features of SQL Server 2005 is the built-in encryption functionality. With this new version of SQL Server, the SQL Server Team has added encryption tools, certificate creation and key management functionality directly to T-SQL. For anyone who males their living securing data in SQL Server tables because of business requirements or regulatory compliance, these new features are a godsend. For those trying to decide whether to use encryption to secure their data, the choice just got a lot easier. This article describes how the new encryption tools work, and how you can use them to your advantage.

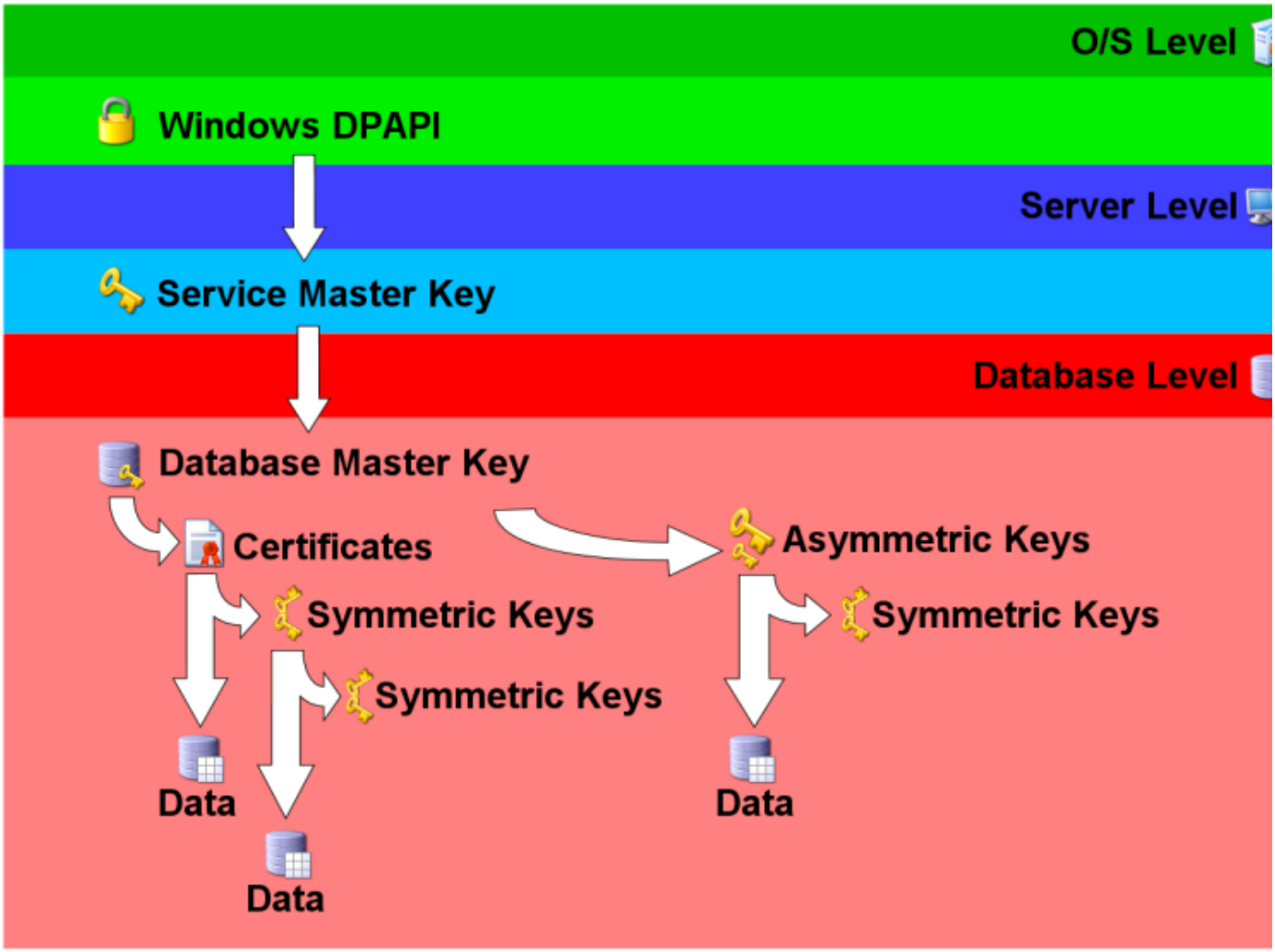
T-SQL now includes support for symmetric encryption and asymmetric encryption using keys, certificates and passwords. This article describes how to create, manage and use symmetric keys and certificates.

Because of the amount of information involved, I've divided this article into three sections:

Part 1: Service and Database Master Keys

The SQL 2005 Encryption Hierarchy

SQL Server 2005 encryption functionality uses a hierarchical model that looks like this:



SQL Server 2005 Encryption Hierarchy

Service Master Key

Each SQL Server 2005 installation has exactly one Service Master Key (SMK), which is generated at install time. The SMK directly or indirectly secures all other keys on the server, making it the "mother of all SQL Server encryption keys." The Windows Data Protection API (DPAPI), at the higher O/S level, uses the SQL Server service account credentials to automatically encrypt and secure the SMK.

Because it is automatically created and managed by the server, Service Master Keys require only a few administrative tools. The SMK can be backed up via the `BACKUP SERVICE MASTER KEY` T-SQL statement. This statement has the following format:

```
BACKUP SERVICE MASTER KEY TO FILE = 'path_to_file'
    ENCRYPTION BY PASSWORD = 'password'
```

Path_to_file is the local path or UNC network path to the file in which the SMK will be backed up. *Password* is a password which is used to encrypt the SMK backup file.

You should backup your Service Master Key and store the backup in a secure off-site location immediately after installing SQL Server 2005.

Should you ever need to restore the Service Master Key from the backup copy, you can use the `RESTORE SERVICE MASTER KEY` statement:

```
RESTORE SERVICE MASTER KEY FROM FILE = 'path_to_file'
    DECRYPTION BY PASSWORD = 'password' [FORCE]
```

The *`path_to_file`* is the UNC or local path to the backup file. *`Password`* is the same password previously

used to encrypt the backup. When restoring the SMK, SQL Server first decrypts all keys and other encrypted information using the current key. It then re-encrypts them with the new SMK. If the decryption process fails at any point, the entire restore process will fail. The FORCE option forces SQL Server to ignore decryption errors and force a restore.


If you have to use the FORCE option of the RESTORE SERVICE MASTER KEY statement, you can count on losing some or all of the encrypted data on your server.

If your Service Master Key is compromised, or you want to change the SQL Server service account, you can regenerate or recover the SMK with the ALTER SERVICE MASTER KEY statement. The format and specific uses of the ALTER SERVICE MASTER KEY statement are available in Books Online.

Because it is automatically generated by SQL Server, there are no CREATE or DROP statements for the Service Master Key.

Database Master Keys

While each SQL Server has a single Service Master Key, each SQL database can have its own Database Master Key (DMK). The DMK is created using the CREATE MASTER KEY statement:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'password'
```

This statement creates the DMK, encrypts it using the supplied password, and stores it in the database. In addition, the DMK is encrypted using the Service Master Key and stored in the master database; a feature known as "automatic key management." We'll talk more about this feature later.

Like the Service Master Key, you can backup and restore Database Master Keys. To backup a DMK, use the BACKUP MASTER KEY statement. The syntax is analogous to backing up a Service Master Key.

```
BACKUP MASTER KEY TO FILE = 'path_to_file'
    ENCRYPTION BY PASSWORD = 'password'
```

Restoring the Database Master Key requires that you use the DECRYPTION BY PASSWORD clause, which specifies the password previously used to encrypt the backup file. In addition you must use the ENCRYPTION BY PASSWORD clause, which gives SQL Server a password to encrypt the DMK after it is loaded in the database.

```
RESTORE MASTER KEY FROM FILE = 'path_to_file'
    DECRYPTION BY PASSWORD = 'password'
    ENCRYPTION BY PASSWORD = 'password'
    [ FORCE ]
```

Like restoring the Service Master Key, the DMK restore statement has a FORCE option which will ignore decryption errors.


It is recommended that you immediately create backups of Database Master Keys and store them in a secure off-site location immediately after creating them. Also, the FORCE option of the RESTORE MASTER KEY statement can result in encrypted data loss.

To drop a DMK, use the DROP MASTER KEY statement:

```
DROP MASTER KEY
```

This statement drops the Database Master Key from the current database. Make sure you are in the correct database before using the DROP MASTER KEY statement.

Automatic Key Management

When you create a Database Master Key, a copy is encrypted with the supplied password and stored in the current database. A copy is also encrypted with the Service Master Key and stored in the master database. The copy of the DMK allows the server to automatically decrypt the DMK, a feature known as "automatic key management." Without automatic key management, you must use the OPEN MASTER KEY statement and supply a password

every time you wish to encrypt and/or decrypt data using certificates and keys that rely on the DMK for security. With automatic key management, the `OPEN MASTER KEY` statement and password are not required.

The potential downfall of automatic key management is that it allows every sysadmin to decrypt the DMK. You can override automatic key management for a DMK with the `DROP ENCRYPTION BY SERVICE MASTER KEY` clause of the `ALTER MASTER KEY` statement. `ALTER MASTER KEY` and all its options are described in full detail in Books Online.

Part 2: Certificates

Creating Certificates

Once you have your Service Master Key and Database Master Key configured, you're ready to begin making certificates. SQL Server 2005 has the ability to generate self-signed X.509 certificates. The flexible `CREATE CERTIFICATE` statement performs this function:

```
CREATE CERTIFICATE certificate_name [ AUTHORIZATION user_name ]
    { FROM <existing_keys> | <generate_new_keys> }
    [ ACTIVE FOR BEGIN_DIALOG =  { ON | OFF } ]

<existing_keys> ::=
    ASSEMBLY assembly_name
    | {
        [ EXECUTABLE ] FILE = 'path_to_file'
        [ WITH PRIVATE KEY ( <private_key_options> ) ]
      }

<generate_new_keys> ::=
    [ ENCRYPTION BY PASSWORD = 'password']
    WITH SUBJECT = 'certificate_subject_name'
    [ , <date_options> [ ,...n ] ]

<private_key_options> ::=
    FILE = 'path_to_private_key'
      [ , DECRYPTION BY PASSWORD = 'password' ]
      [ , ENCRYPTION BY PASSWORD = 'password' ]

<date_options> ::=
    START_DATE = 'mm/dd/yyyy' | EXPIRY_DATE = 'mm/dd/yyyy'
```

There are a lot of options associated with the `CREATE CERTIFICATE` statement. Fortunately few are needed most of the time. The following statement will create a certificate encrypted by password:

```
CREATE CERTIFICATE TestCertificate
   ENCRYPTION BY PASSWORD = 'thisIsAP@$$w0rd'
   WITH SUBJECT = 'This is a test certificate',
   START_DATE = '1/1/2006',
   EXPIRY_DATE = '12/31/2008';
```

If you leave off the `ENCRYPTION BY PASSWORD` clause, the Database Master Key is used to encrypt the certificate. Leaving the `START_DATE`
out will result in the current date being used as the default start date for your certificate.

You can also use the `CREATE CERTIFICATE` statement to import an existing certificate into your SQL Server.

In addition to `CREATE CERTIFICATE`, SQL Server provides additional statements to manage certificates. These include `DROP CERTIFICATE`, `ALTER CERTIFICATE`, and `BACKUP CERTIFICATE`.

There is no `RESTORE` statement for certificates. Use the `CREATE CERTIFICATE` statement to restore a backed-up certificate.

Encryption and Decryption by Certificate

Certificates can be used to encrypt and decrypt data directly by using the built-in `EncryptByCert`, `DecryptByCert` and `Cert_ID` functions. The `Cert_ID` function returns the ID of the certificate with the specified name. The format of the `Cert_ID` function is:

```
Cert_ID ( 'cert_name' )
```

The *`'cert_name'`* is the name of the certificate. The `EncryptByCert` function requires the Certificate ID and has the following format:

```
EncryptByCert ( certificate_ID , { 'cleartext' | @cleartext } )
```

The *`certificate_ID`* is acquired by using the `Cert_ID` function. *`'Cleartext'`* is the clear text string to encrypt. The clear text can be a char, varchar, nchar, nvarchar or wchar value. The EncryptByCert function returns a varbinary result of up to 8,000 bytes.

The `DecryptByCert` function is used to decrypt data that was previously encrypted by certificate. The format for `DecryptByCert` looks like this:

```
DecryptByCert (certificate_ID,
               { 'ciphertext' | @ciphertext }
               [ , { 'cert_password' | @cert_password } ]
               )
```

Like `EncryptByCert`, *`certificate_ID`* can be obtained using the `Cert_ID` function. *`'Ciphertext'`* is the previously encrypted text. If you created your certificate with the `ENCRYPT BY PASSWORD` clause, *`'cert_password'`* must be the same password you used when you created the certificate. If you did not use `ENCRYPT BY PASSWORD` to create the certificate, leave out *`'cert_password'`*.

The following sample script creates a Database Master Key, a test certificate and demonstrates how to encrypt/decrypt data using the certificate.

```
-- Sample T-SQL Script to demonstrate Certificate Encryption

-- Use the AdventureWorks database
USE AdventureWorks;

-- Create a Database Master Key
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'p@ssw0rd';

-- Create a Temp Table
CREATE TABLE Person.#Temp
(ContactID   INT PRIMARY KEY,
 FirstName   NVARCHAR(200),
 MiddleName  NVARCHAR(200),
 LastName    NVARCHAR(200),
 eFirstName  VARBINARY(200),
 eMiddleName VARBINARY(200),
 eLastName   VARBINARY(200));

-- Create a Test Certificate, encrypted by the DMK
CREATE CERTIFICATE TestCertificate
   WITH SUBJECT = 'Adventureworks Test Certificate',
   EXPIRY_DATE = '10/31/2009';

-- EncryptByCert demonstration encrypts 100 names from the Person.Contact table
INSERT
INTO Person.#Temp (ContactID, eFirstName, eMiddleName, eLastName)
SELECT ContactID, EncryptByCert(Cert_ID('TestCertificate'), FirstName),
        EncryptByCert(Cert_ID('TestCertificate'), MiddleName),
        EncryptByCert(Cert_ID('TestCertificate'), LastName)
FROM Person.Contact
WHERE ContactID <= 100;

-- DecryptByCert demonstration decrypts the previously encrypted data
UPDATE Person.#Temp
SET FirstName = DecryptByCert(Cert_ID('TestCertificate'), eFirstName),
        MiddleName = DecryptByCert(Cert_ID('TestCertificate'), eMiddleName),
        LastName = DecryptByCert(Cert_ID('TestCertificate'), eLastName);

-- View the results
```

```
SELECT *
FROM Person.#Temp;

-- Clean up work:  drop temp table, test certificate and master key
DROP TABLE Person.#Temp;
DROP CERTIFICATE TestCertificate;
DROP MASTER KEY;
```

Part 3: Symmetric Keys

Creating Symmetric Keys

You can use certificates to create symmetric keys for encryption and decryption within the database. The `CREATE SYMMETRIC KEY` statement has the following syntax:

```
CREATE SYMMETRIC KEY key_name [ AUTHORIZATION owner_name ]
    WITH <key_options> [ , ... n ]
    ENCRYPTION BY <encrypting_mechanism> [ , ... n ]

<encrypting_mechanism> ::=
    CERTIFICATE certificate_name |
    PASSWORD = 'password' |
    SYMMETRIC KEY symmetric_key_name |
    ASYMMETRIC KEY asym_key_name

<key_options> ::=
    KEY_SOURCE = 'pass_phrase' |
    ALGORITHM = <algorithm> |
    IDENTITY_VALUE = 'identity_phrase'

<algorithm> ::=
    DES | TRIPLE_DES | RC2 | RC4 | DESX | AES_128 | AES_192 | AES_256
```

Like the `CREATE CERTIFICATE` statement, `CREATE SYMMETRIC KEY` is very flexible. In most situations you will probably use a small subset of the available options. As an example, this statement creates a symmetric key and encrypts it with the Test Certificate created in the previous section:

```
CREATE SYMMETRIC KEY TestSymmetricKey
    WITH ALGORITHM = TRIPLE_DES
    ENCRYPTION BY CERTIFICATE TestCertificate;
```

Symmetric keys can be secured via other symmetric keys, asymmetric keys and passwords, as well as by certificates. SQL Server also provides `ALTER SYMMETRIC KEY` and `DROP SYMMETRIC KEY` statements to manage your symmetric keys. Specific syntax for these statements can be found in Books Online.

When dropping keys and certificates, the order is important. SQL 2005 will not allow you to `DROP` certificates or keys if they are being used to encrypt other keys within the database.

Symmetric Key Encryption

SQL Server provides a set of functions to encrypt and decrypt data by symmetric key. These functions are `EncryptByKey`, `DecryptByKey` and `Key_GUID`. The `Key_GUID` function returns the unique identifier assigned to a specific symmetric key. The format of the function is:

```
Key_GUID( 'Key_Name' )
```

The `EncryptByKey` function requires a reference to the symmetric key GUID in order to encrypt data. The format of the `EncryptByKey` function is:

```
EncryptByKey( key_GUID, { 'cleartext' | @cleartext }
             [ , { add_authenticator | @add_authenticator }
```

```
            , { authenticator | @authenticator } ]
        )
```

The *key_GUID* is the symmetric key GUID, *'cleartext'* is the plain text to be encrypted. *Add_authenticator* and *authenticator* are optional parameters that can help eliminate post-encryption patterns from your data.

The DecryptByKey function performs the reverse of EncryptByKey. This function decrypts your previously encrypted data. The format for DecryptByKey is:

```
DecryptByKey( { 'ciphertext' | @ciphertext }
            [ , add_authenticator
              , { authenticator | @authenticator } ]
        )
```

'Ciphertext' is the encrypted text. *Add_authenticator* and *authenticator*, if present, must match the values used in the EncryptByKey function. The DecryptByKey function doesn't require you to explicitly specify the symmetric key GUID. The symmetric key used previously to encrypt the data must be open, however. The OPEN SYMMETRIC KEY statement is used to open a symmetric key.

Here is a sample T-SQL script demonstrating encryption and decryption by symmetric key:

```
-- Use the AdventureWorks database
USE AdventureWorks;

-- Create a Database Master Key
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'p@ssw0rd';

-- Create a Temp Table
CREATE TABLE Person.#Temp
(ContactID   INT PRIMARY KEY,
 FirstName   NVARCHAR(200),
 MiddleName  NVARCHAR(200),
 LastName    NVARCHAR(200),
 eFirstName  VARBINARY(200),
 eMiddleName VARBINARY(200),
 eLastName   VARBINARY(200));

-- Create a Test Certificate
CREATE CERTIFICATE TestCertificate
   WITH SUBJECT = 'Adventureworks Test Certificate',
   EXPIRY_DATE = '10/31/2009';

-- Create a Symmetric Key
CREATE SYMMETRIC KEY TestSymmetricKey
        WITH ALGORITHM = TRIPLE_DES
    ENCRYPTION BY CERTIFICATE TestCertificate;
OPEN SYMMETRIC KEY TestSymmetricKey
   DECRYPTION BY CERTIFICATE TestCertificate;

-- EncryptByKey demonstration encrypts 100 names from the Person.Contact table
INSERT
INTO Person.#Temp (ContactID, eFirstName, eMiddleName, eLastName)
SELECT ContactID,
        EncryptByKey(Key_GUID('TestSymmetricKey'), FirstName),
        EncryptByKey(Key_GUID('TestSymmetricKey'), MiddleName),
        EncryptByKey(Key_GUID('TestSymmetricKey'), LastName)
FROM Person.Contact
WHERE ContactID <= 100;

-- DecryptByKey demonstration decrypts the previously encrypted data
UPDATE Person.#Temp
SET FirstName = DecryptByKey(eFirstName),
        MiddleName = DecryptByKey(eMiddleName),
        LastName = DecryptByKey(eLastName);

-- View the results
SELECT *
FROM Person.#Temp;

-- Clean up work:  drop temp table, symmetric key, test certificate and master key
DROP TABLE Person.#Temp;
CLOSE SYMMETRIC KEY TestSymmetricKey;
DROP SYMMETRIC KEY TestSymmetricKey;
```

```
DROP CERTIFICATE TestCertificate;
DROP MASTER KEY;
```

Conclusions

SQL Server 2005 includes several new functions to securely create, manage and use encryption keys and certificates to secure sensitive data. Taking advantage of this new functionality can greatly enhance your database and application security.

Reference

SQL Books Online References:

- ALTER CERTIFICATE
- ALTER MASTER KEY
- ALTER SERVICE MASTER KEY
- ALTER SYMMETRIC KEY
- BACKUP CERTIFICATE
- BACKUP MASTER KEY
- BACKUP SERVICE MASTER KEY
- Cert_ID
- CLOSE MASTER KEY
- CLOSE SYMMETRIC KEY
- CREATE CERTIFICATE
- CREATE MASTER KEY
- CREATE SYMMETRIC KEY
- DecryptByCert
- DecryptByKey
- DROP CERTIFICATE
- DROP MASTER KEY
- DROP SYMMETRIC KEY
- EncryptByCert
- EncryptByKey
- Key_GUID
- OPEN MASTER KEY
- OPEN SYMMETRIC KEY
- RESTORE MASTER KEY
- RESTORE SERVICE MASTER KEY