Environment set up

Let’s prepare the environment for this article.

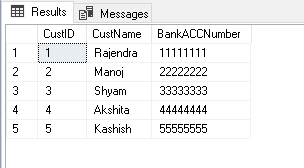
* Create a new database and create **CustomerInfo** table

|  |  |
| --- | --- |
|  | CREATE DATABASE CustomerData;              Go              USE CustomerData;              GO                CREATE TABLE CustomerData.dbo.CustomerInfo              (CustID        INT PRIMARY KEY,               CustName     VARCHAR(30) NOT NULL,               BankACCNumber VARCHAR(10) NOT NULL              );              GO |

* Insert sample data into **CustomerInfo** table

|  |  |
| --- | --- |
|  | Insert into CustomerData.dbo.CustomerInfo (CustID,CustName,BankACCNumber)              Select 1,'Rajendra',11111111 UNION ALL              Select 2, 'Manoj',22222222 UNION ALL              Select 3, 'Shyam',33333333 UNION ALL              Select 4,'Akshita',44444444 UNION ALL              Select 5, 'Kashish',55555555 |

* View the records in **CustomerInfo** table



We use the following steps for column level encryption:

1. Create a database master key
2. Create a self-signed certificate for SQL Server
3. Configure a symmetric key for encryption
4. Encrypt the column data
5. Query and verify the encryption

We will first use these steps and later explain the overall process using Encryption Hierarchy in SQL Server using the following image (Reference – [Microsoft Docs](https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/encryption-hierarchy?redirectedfrom=MSDN&view=sql-server-ver15)):

Create a database master key for column level SQL Server encryption

In this first step, we define a database master key and provide a password to protect it. It is a symmetric key for protecting the private keys and asymmetric keys. In the above diagram, we can see that a service master key protects this database master key. SQL Server creates this service master key during the installation process.

We use **CREATE MASTER KEY** statement for creating a database master key:

|  |  |
| --- | --- |
|  | USE CustomerData;  GO  CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'SQLShack@1'; |

We can use **sys.symmetric\_keys** catalog view to verify the existence of this database master key in SQL Server encryption:

|  |  |
| --- | --- |
|  | SELECT name KeyName,      symmetric\_key\_id KeyID,      key\_length KeyLength,      algorithm\_desc KeyAlgorithm  FROM sys.symmetric\_keys; |

In the output, we can notice that it creates a **##MS\_DatabaseMasterKey##** with key algorithm AES\_256. SQL Server automatically chooses this key algorithm and key length:

symmetric key 

Create a self-signed certificate for Column level SQL Server encryption

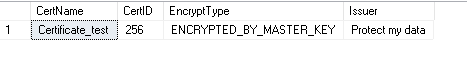
In this step, we create a self-signed certificate using the CREATE CERTIFICATE statement. You might have seen that an organization receives a certificate from a certification authority and incorporates into their infrastructures. In SQL Server, we can use a self-signed certificate without using a certification authority certificate.

Execute the following query for creating a certificate:

|  |  |
| --- | --- |
|  | USE CustomerData;  GO  CREATE CERTIFICATE Certificate\_test WITH SUBJECT = 'Protect my data';  GO |

We can verify the certificate using the catalog view **sys.certificates**:

|  |  |
| --- | --- |
|  | SELECT name CertName,      certificate\_id CertID,      pvt\_key\_encryption\_type\_desc EncryptType,      issuer\_name Issuer  FROM sys.certificates; |



In the output, we can note the following fields:

* **Encrypt Type:** In this column, we get a value **ENCRYPTED\_BY\_MASTER\_KEY,**and it shows that SQL Server uses the database master key created in the previous step and protects this certificate
* **CertName**: It is the certificate name that we defined in the CREATE CERTIFICATE statement
* **Issuer**: We do not have a certificate authority certificate; therefore, it shows the subject value we defined in the CREATE CERTIFICATE statement

Optionally, we can use **ENCRYPTION BY PASSWORD**and**EXPIRY\_DATE** parameters in the CREATE CERTIFICATE; however, we will skip it in this article.

Configure a symmetric key for column level SQL Server encryption

In this step, we will define a symmetric key that you can see in the encryption hierarchy as well. The symmetric key uses a single key for encryption and decryption as well. In the image shared above, we can see the symmetric key on top of the data. It is recommended to use the symmetric key for data encryption since we get excellent performance in it. For column encryption, we use a multi-level approach, and it gives the benefit of the performance of the symmetric key and security of the asymmetric key.

We use **CREATE SYMMETRIC KEY** statement for it using the following parameters:

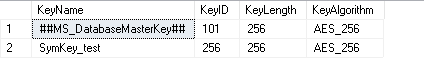
* **ALGORITHM:** AES\_256
* **ENCRYPTION BY CERTIFICATE:** It should be the same certificate name that we specified earlier using CREATE CERTIFICATE statement

|  |  |
| --- | --- |
|  | CREATE SYMMETRIC KEY SymKey\_test WITH ALGORITHM = AES\_256 ENCRYPTION BY CERTIFICATE Certificate\_test; |

Once we have created this symmetric key, check the existing keys using catalog view for column level SQL Server Encryption as checked earlier:

|  |  |
| --- | --- |
|  | SELECT name KeyName,      symmetric\_key\_id KeyID,      key\_length KeyLength,      algorithm\_desc KeyAlgorithm  FROM sys.symmetric\_keys; |

We can see two key entries now as it includes both the database master key and the symmetric key:



We have created the required encryption keys in this demo. It has the following setup that you can see in the image shown above as well:

* SQL Server installation creates a Service Master Key (SMK), and Windows operating system Data Protection API (DPAPI) protects this key
* This Service Master Key (SMK) protects the database master key (DMK)
* A database master key (DMK) protects the self-signed certificate
* This certificate protects the Symmetric key

Data encryption

SQL Server encrypted column datatype should be **VARBINARY**. In our **CustomerData** table, the **BankACCNumber** column data type is Varchar(10). Let’s add a new column of VARBINARY(max) datatype using the ALTER TABLE statement specified below:

|  |  |
| --- | --- |
|  | ALTER TABLE CustomerData.dbo.CustomerInfo  ADD BankACCNumber\_encrypt varbinary(MAX) |

Let’s encrypt the data in this newly added column.

* In a query window, open the symmetric key and decrypt using the certificate. We need to use the same symmetric key and certificate name that we created earlier

|  |  |
| --- | --- |
|  | OPEN SYMMETRIC KEY SymKey\_test          DECRYPTION BY CERTIFICATE Certificate\_test; |

* In the same session, use the following UPDATE statement. It uses **EncryptByKey** function and uses the symmetric function for encrypting the **BankACCNumber** column and updates the values in the newly created **BankACCNumber\_encrypt** column

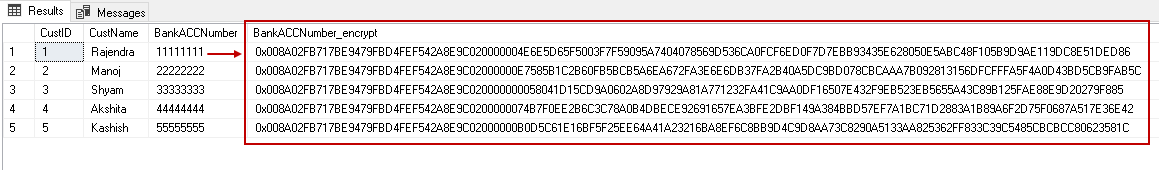
|  |  |
| --- | --- |
|  | UPDATE CustomerData.dbo.CustomerInfo          SET BankACCNumber\_encrypt = EncryptByKey (Key\_GUID('SymKey\_test'), BankACCNumber)          FROM CustomerData.dbo.CustomerInfo;          GO |

* Close the symmetric key using the **CLOSE SYMMETRIC KEY** statement. If we do not close the key, it remains open until the session is terminated

|  |  |
| --- | --- |
|  | CLOSE SYMMETRIC KEY SymKey\_test;              GO |

* Verify the records in the **CustomerInfo** table

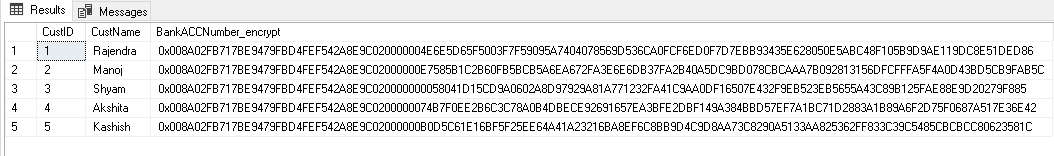
We can see the encrypted records in the newly added column. If the user has access to this table also, he cannot understand the data without decrypting it:

[](https://www.sqlshack.com/wp-content/uploads/2020/01/verify-the-records-in-the-customerinfo.png)

Let’s remove the old column as well:

|  |  |
| --- | --- |
|  | ALTER TABLE CustomerData.dbo.CustomerInfo DROP COLUMN BankACCNumber;  GO |

Now, we have only an encrypted value for the bank account number:

[](https://www.sqlshack.com/wp-content/uploads/2020/01/check-the-encrypted-data.png)

Decrypt column level SQL Server encryption data

We need to execute the following commands for decrypting column level encrypted data:

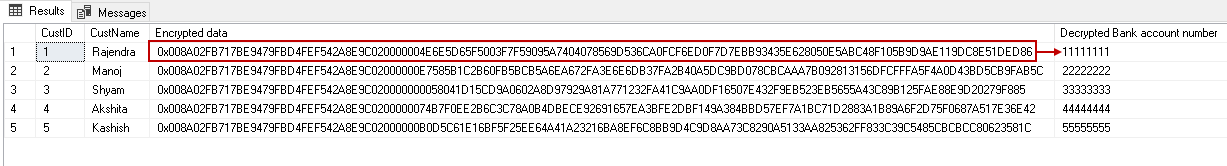
* In a query window, open the symmetric key and decrypt using the certificate. We need to use the same symmetric key and certificate name that we created earlier

|  |  |
| --- | --- |
|  | OPEN SYMMETRIC KEY SymKey\_test          DECRYPTION BY CERTIFICATE Certificate\_test; |

* Use the SELECT statement and decrypt encrypted data using the **DecryptByKey()** function

|  |  |
| --- | --- |
|  | SELECT CustID, CustName,BankACCNumber\_encrypt AS 'Encrypted data',              CONVERT(varchar, DecryptByKey(BankACCNumber\_encrypt)) AS 'Decrypted Bank account number'              FROM CustomerData.dbo.CustomerInfo; |

We can see both encrypted and decrypted data in the following screenshot:

[](https://www.sqlshack.com/wp-content/uploads/2020/01/encrypted-and-decrypted-data-using-column-level-sq.png)

Permissions required for decrypting data

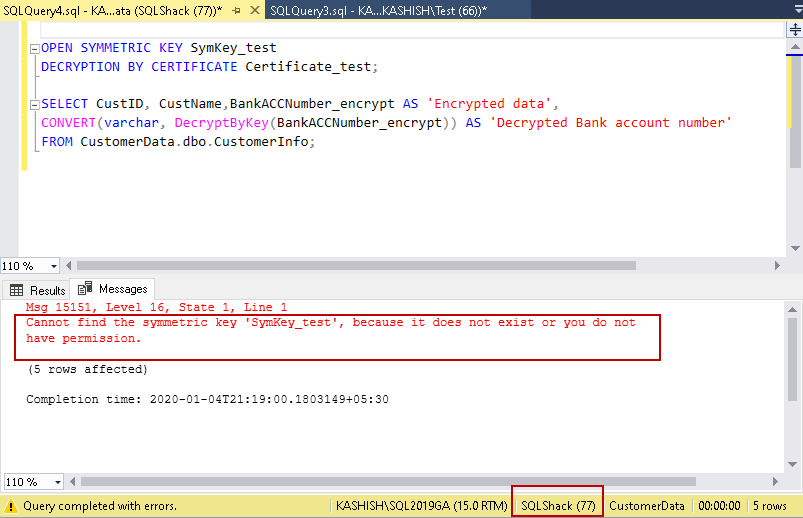
A user with the read permission cannot decrypt data using the symmetric key. Let’s simulate the issue. For this, we will create a user and provide **db\_datareader** permissions on **CustomerData** database:

|  |  |
| --- | --- |
|  | USE [master]  GO  CREATE LOGIN [SQLShack] WITH PASSWORD=N'sqlshack', DEFAULT\_DATABASE=[CustomerData], CHECK\_EXPIRATION=OFF, CHECK\_POLICY=OFF  GO  USE [CustomerData]  GO  CREATE USER [SQLShack] FOR LOGIN [SQLShack]  GO  USE [CustomerData]  GO  ALTER ROLE [db\_datareader] ADD MEMBER [SQLShack]  GO |

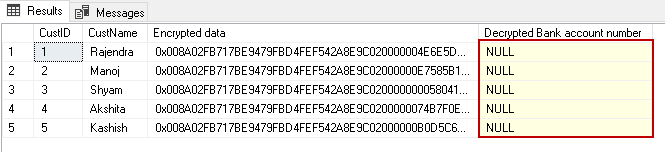
Now connect to SSMS using SQLShack user and execute the query to select the record with decrypting **BankACCNumber\_encrypt** column:

|  |  |
| --- | --- |
|  | OPEN SYMMETRIC KEY SymKey\_test  DECRYPTION BY CERTIFICATE Certificate\_test;    SELECT CustID, CustName,BankACCNumber\_encrypt AS 'Encrypted data',  CONVERT(varchar, DecryptByKey(BankACCNumber\_encrypt)) AS 'Decrypted Bank account number'  FROM CustomerData.dbo.CustomerInfo; |

In the output message, we get the message that the symmetric key does not exist, or the user does not have permission to use it:



Click on the results, and we get the NULL values in the decrypted column, as shown below:



We can provide permissions to the Symmetric key and Certificate:

* **Symmetric key permission**: GRANT VIEW DEFINITION
* **Certificate permission:**GRANT VIEW DEFINITION and GRANT CONTROL permissions

Execute these scripts with from a user account with admin privileges:

|  |  |
| --- | --- |
|  | GRANT VIEW DEFINITION ON SYMMETRIC KEY::SymKey\_test TO SQLShack;  GO  GRANT VIEW DEFINITION ON Certificate::[Certificate\_test] TO SQLShack;  GO  GRANT CONTROL ON Certificate::[Certificate\_test] TO SQLShack; |

Now, go back and re-execute the SELECT statement:

