An overview of the column level SQL Server encryption

January 14, 2020 by Rajendra Gupta

This article gives an overview of column level SQL Server encryption using examples.

Introduction

Data security is a critical task for any organization, especially if you store customer personal data such as Customer contact number, email address, social security number, bank and credit card numbers. Our main goal is to protect unauthorized access to data within and outside the organization. To achieve this, we start by providing access to relevant persons. We still have a chance that these authorized persons can also misuse the data; therefore, SQL Server provides encryption solutions. We can use these encryptions and protect the data.

It is a crucial aspect in classifying the data based on the information type and sensitivity. For example, we might have customer DOB in a column and depending upon the requirement, and we should classify it as confidential, highly confidential. You can read more about in the article SQL data classification – Add sensitivity classification in SQL Server 2019.

We have many encryptions available in SQL Server such as Transparent Data Encryption (TDE), Always Encrypted, Static data masking and Dynamic Data Masking. In this article, we will explore column level SQL Server encryption using symmetric keys.

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

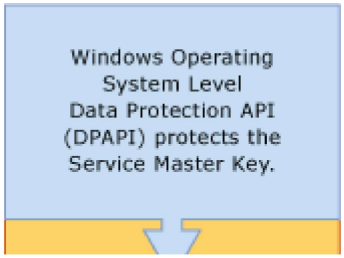
Results | Messages

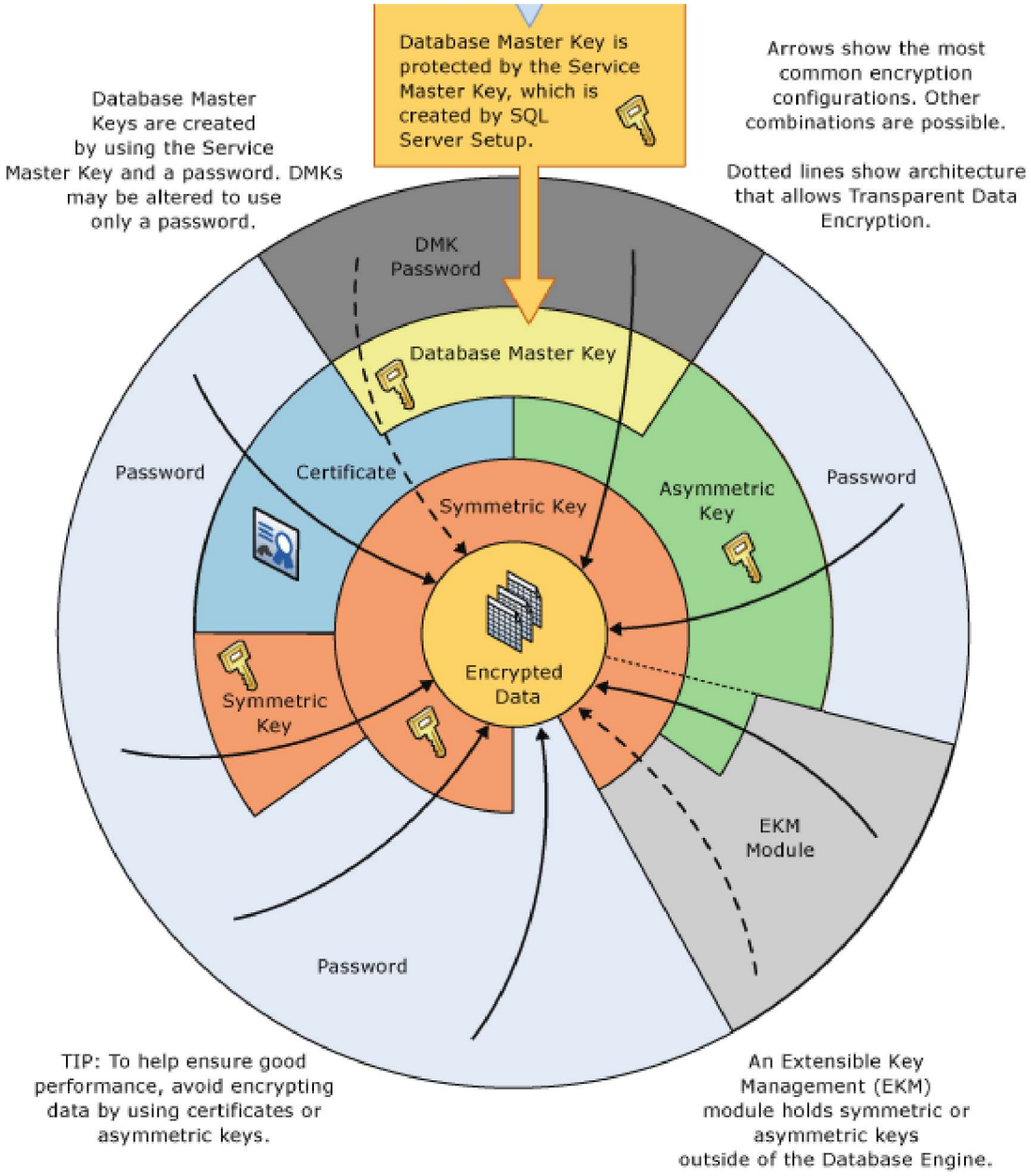
	CustID	CustName	BankACCNumber
1	1	Rajendra	11111111
2	2	Manoj	22222222
3	3	Shyam	33333333
4	4	Akshita	44444444
5	5	Kashish	55555555

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):





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:

	KeyName	KeyID	KeyLength	KeyAlgorithm
1	##MS_DatabaseMasterKey##	101	256	AES_256

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;
```

	CertName	CertID	EncryptType	Issuer
1	Certificate_test	256	ENCRYPTED_BY_MASTER_KEY	Protect my data

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 CERTIFICAT
E 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:

	KeyName	KeyID	KeyLength	KeyAlgorithm
1	##MS_DatabaseMasterKey##	101	256	AES_256
2	SymKey_test	256	256	AES_256

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

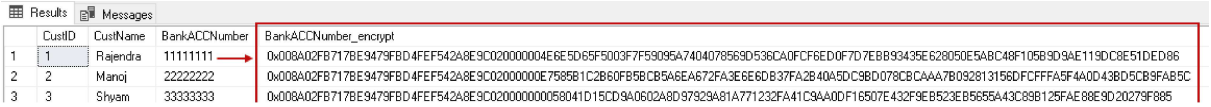
```
UPDATE CustomerData.dbo.CustomerInfo
        SET BankACCNumber_encrypt = EncryptByKey (Key_GUID('SymKey_test'), Ba
nkACCNumber)
        FROM CustomerData.dbo.CustomerInfo;
        GO
```

- 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:



4	4	Akshita	44444444	0x008A02FB717BE9479FBD4FEF542A8E9C0200000074B7F0EE2B6C3C78A0B4DBECE92691657EA3BFE2DBF149A384BBD57EF7A1BC71D2883A1B89A6F2D75F0687A517E36E42
5	5	Kashish	55555555	0x008A02FB717BE9479FBD4FEF542A8E9C02000000B0D5C61E16BF5F25EE64A41A23216BA8EF6C8BB9D4C9D8AA73C8290A5133AA825362FF833C39C5485CBCBCC80623581C

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:

Results Messages

	CustID	CustName	BankACCNumber_encrypt
1	1	Rajendra	0x008A02FB717BE9479FBD4FEF542A8E9C020000004E6E5D65F5003F7F59095A7404078569D536CA0FCF6ED0F7D7EBB93435E628050E5ABC48F105B9D9AE119DC8E51DED86
2	2	Manoj	0x008A02FB717BE9479FBD4FEF542A8E9C02000000E7585B1C2B60FB5BCB5A6EA672FA3E6E6DB37FA2B40A5DC9BD078CBCAAA7B092813156DFCFFFA5F4A0D43BD5CB9FAB5C
3	3	Shyam	0x008A02FB717BE9479FBD4FEF542A8E9C020000000058041D15CD9A0602A8D97929A81A771232FA41C9AA0DF16507E432F9EB523EB5655A43C89B125FAE88E9D20279F885
4	4	Akshita	0x008A02FB717BE9479FBD4FEF542A8E9C0200000074B7F0EE2B6C3C78A0B4DBECE92691657EA3BFE2DBF149A384BBD57EF7A1BC71D2883A1B89A6F2D75F0687A517E36E42
5	5	Kashish	0x008A02FB717BE9479FBD4FEF542A8E9C02000000B0D5C61E16BF5F25EE64A41A23216BA8EF6C8BB9D4C9D8AA73C8290A5133AA825362FF833C39C5485CBCBCC80623581C

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 'Decrypt
ed Bank account number'
            FROM CustomerData.dbo.CustomerInfo;
```

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

Results Messages

	CustID	CustName	Encrypted data	Decrypted Bank account number
1	1	Rajendra	0x008A02FB717BE9479FBD4FEF542A8E9C020000004E6E5D65F5003F7F59095A7404078569D536CA0FCF6ED0F7D7EBB93435E628050E5ABC48F105B9D9AE119DC8E51DED86	11111111
2	2	Manoj	0x008A02FB717BE9479FBD4FEF542A8E9C02000000E7585B1C2B60FB5BCB5A6EA672FA3E6E6DB37FA2B40A5DC9BD078CBCAAA7B092813156DFCFFFA5F4A0D43BD5CB9FAB5C	22222222
3	3	Shyam	0x008A02FB717BE9479FBD4FEF542A8E9C020000000058041D15CD9A0602A8D97929A81A771232FA41C9AA0DF16507E432F9EB523EB5655A43C89B125FAE88E9D20279F885	33333333
4	4	Akshita	0x008A02FB717BE9479FBD4FEF542A8E9C0200000074B7F0EE2B6C3C78A0B4DBECE92691657EA3BFE2DBF149A384BBD57EF7A1BC71D2883A1B89A6F2D75F0687A517E36E42	44444444
5	5	Kashish	0x008A02FB717BE9479FBD4FEF542A8E9C02000000B0D5C61E16BF5F25EE64A41A23216BA8EF6C8BB9D4C9D8AA73C8290A5133AA825362FF833C39C5485CBCBCC80623581C	55555555

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:

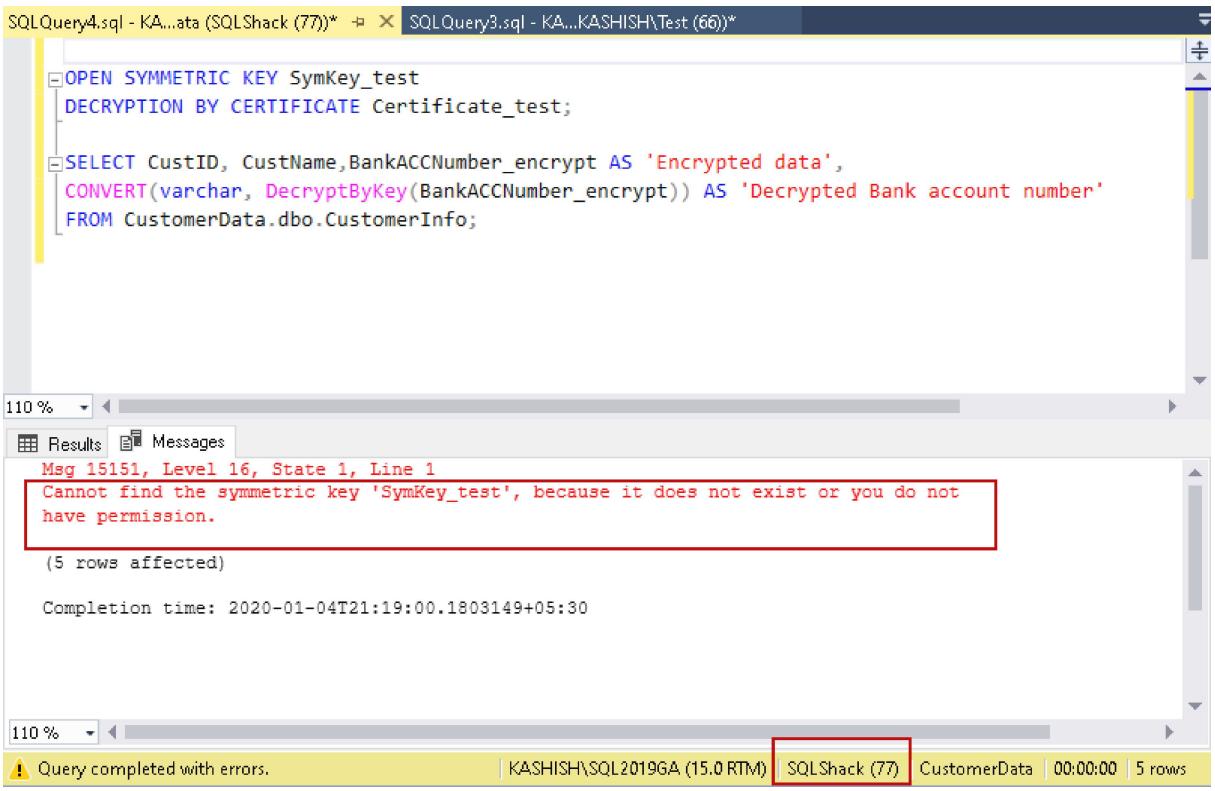
```
USE [master]
GO
CREATE LOGIN [SQLShack] WITH PASSWORD=N'sqlshack', DEFAULT_DATABASE=[CustomerData]
, CHECK_EXPIRATION=OFF, CHECK_POLICY=OFF
GO
USE [CustomerData]
GO
CREATE USER [SQLShack] FOR LOGIN [SQLShack]
GO
USE [CustomerData]
GO
ALTER ROLE [db_datareader] ADD MEMBER [SQLShack]
GO
```

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 n
umber'
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:

1	1	Rajendra	0x008A02FB717BE9479FBD4FEF542A8E9C020000004E6E5D...	NULL
2	2	Manoj	0x008A02FB717BE9479FBD4FEF542A8E9C02000000E7585B1...	NULL
3	3	Shyam	0x008A02FB717BE9479FBD4FEF542A8E9C020000000058041...	NULL
4	4	Akshita	0x008A02FB717BE9479FBD4FEF542A8E9C0200000074B7F0E...	NULL
5	5	Kashish	0x008A02FB717BE9479FBD4FEF542A8E9C02000000B0D5C6...	NULL

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:

```
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;

CLOSE SYMMETRIC KEY SymKey_test;
GO
```

110 %

Results | Messages

	CustID	CustName	Encrypted data	Decrypted Bank account number
1	1	Rajendra	0x008A02FB717BE9479FBD4FEF542A8E9C020000004E6E5D...	11111111
2	2	Manoj	0x008A02FB717BE9479FBD4FEF542A8E9C02000000E7585B1...	22222222
3	3	Shyam	0x008A02FB717BE9479FBD4FEF542A8E9C020000000058041...	33333333
4	4	Akshita	0x008A02FB717BE9479FBD4FEF542A8E9C0200000074B7F0E...	44444444
5	5	Kashish	0x008A02FB717BE9479FBD4FEF542A8E9C02000000B0D5C6...	55555555

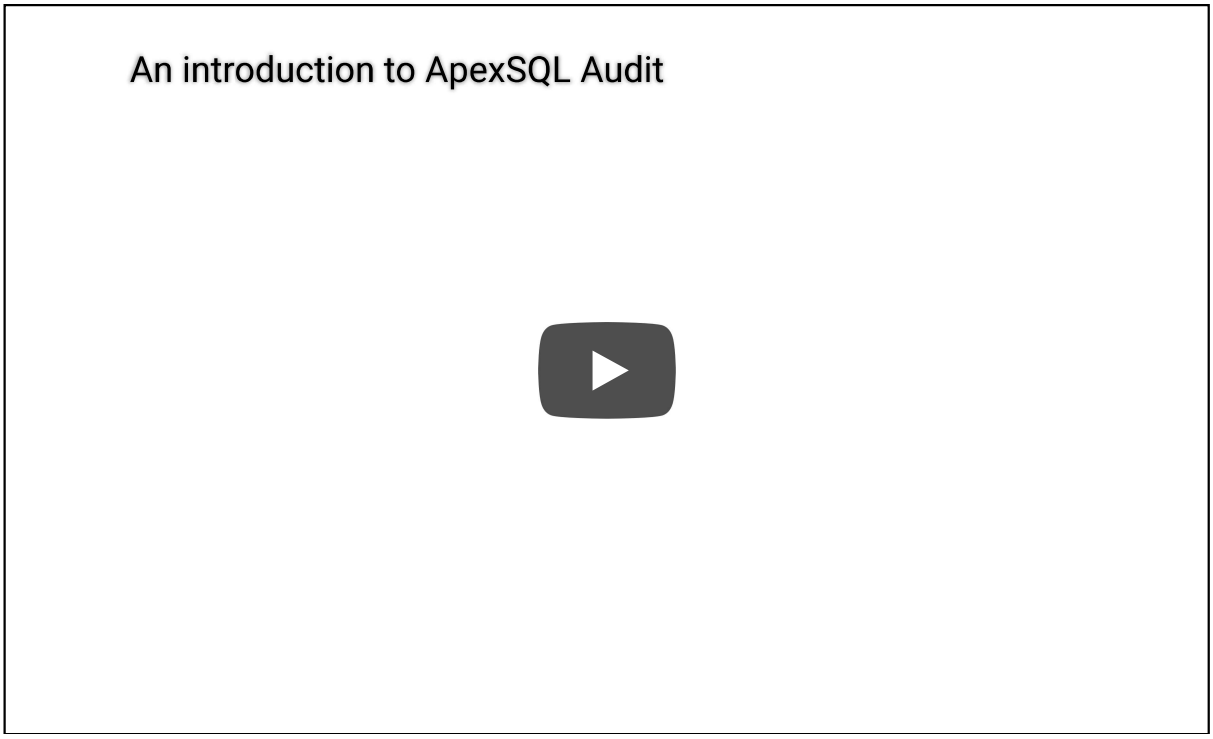
Query executed successfully. | KASHISH\SQL2019GA (15.0 RTM) | SQLShack (70) | CustomerData | 00:00:00 | 5 rows

Conclusion

In this article, we explored column level SQL Server encryption using the symmetric key. We can use the same key for encrypting other table columns as well. You should explore the encryption and decryption mechanism; however, you should consider the requirements first and then consider the appropriate encryption mechanism as per your need.

See more

Interested in an enterprise-level SQL Server audit and compliance solution for GDPR, HIPAA, PCI and more, including tamper-proof repository, fail-over/fault tolerant auditing, tamper-evident repository, sophisticated filters, alerting and reports? Consider ApexSQL Audit, a database auditing tool for SQL Server



Rajendra Gupta

Rajendra has 8+ years of experience in database administration having a passion for database performance optimization, monitoring, and high availability and disaster recovery technologies, learning new things, new features.

While working as a Senior consultant DBA for big customers and having certified with MCSA SQL 2012, he likes to share knowledge on various blogs.
He can be reached at rajendra.gupta16@gmail.com

View all posts by Rajendra Gupta

Related Posts:

1. **Restoring Transparent Data Encryption (TDE) enabled databases on a different server**
2. **Transparent Data Encryption (TDE) in AWS RDS SQL Server**
3. **SQL Server Confidential – Part II – SQL Server Cryptographic Features**
4. **SQL Server ALTER TABLE ADD Column overview**
5. **Overview of the SQL DELETE Column from an existing table operation**

Security

168 Views

ALSO ON SQL SHACK

Test-driven database hotfix development …

2 months ago • 1 comment

This article is about using tSQLt for test-driven database hotfix …

Install SQL Server 2019 on Windows Server …

4 months ago • 1 comment

In this article, we will discuss configuring SQL Server Always On Availability …

Lever T-SQL for Pinpoint Control of …

5 months ago • 1 comment

This article spotlights a clean, efficient, pinpoint T-SQL stored procedure …

Learn S SQL Se

5 months

Loops are basic, stil concepts