[How to Implement Row and Column Level Security in SQL Server (netwrix.com)](https://blog.netwrix.com/2019/06/27/how-to-implement-row-and-column-level-security-in-sql-server/)

SQL Server includes a range of security features. Two of the most valuable are column level security and row level security, which enable you to granularly grant users access to rows and columns in a table. This article explains how to implement these features in your environment.

**Securables, Principals, and Permissions**

Before we can dive into rows and column security, we need to review some [SQL Server security](https://blog.netwrix.com/2019/05/30/6-sql-server-security-basics-every-database-admin-should-know/) basics. Basically, [database security](https://blog.netwrix.com/2020/02/20/getting-started-with-database-security/) involves allowing someone to access a data stored in a database and perform actions on it.  The “someone” a *principal*, the actions they is allowed to take are their *permissions,* and the data on which the actions are performed is a *securable —* examples of securables include databases, tables, schemas, rows and columns.

Let’s take a look at the permissions that can be applied to securables:

* **SELECT** — Principals requires this permission to get or read data from the table.
* **INSERT** — Principals need this permission to add new rows to a table.
* **UPDATE** — Principals need this permission to modify data in a table.
* **DELETE** — Principals need this permission to remove rows from a table.
* **REFERENCES** — Principals need this permission to create a foreign key to a table if they have no other permissions on the table.

**Handpicked related content:**

* [[Free Guide] SQL Server Security Best Practices](https://www.netwrix.com/sql_server_security_best_practices.html?itm_source=blog&itm_medium=context&itm_campaign=sql&itm_content=none&cID=70170000000kgEZ)

**SQL Server Column Level Security**

There are two ways to protect columns in SQL Server: column level permissions and column level encryption. The great thing for administrators is that these two techniques can be combined.

**Column Level Permissions**

Column level permissions provide a more granular level of security for data in your database. You do not need to execute a separate GRANT or DENY statements for each column; just name them all in a query:

GRANT SELECT ON data1.table (column1, column2) TO user1;

GO

DENY SELECT ON data1.table (column3) TO user1;

GO

If you execute a DENY statement at table level to a column for a user, and after that you execute a GRANT statement on the same column, the DENY permission is removed and the user can have access to that column. Similarly, if you execute GRANT and then DENY, the DENY permission will be in force.

**Handpicked related content:**

* [How to Get User Permissions in SQL Server](https://www.netwrix.com/how_to_get_user_permissions_in_sql_server.html?itm_source=blog&itm_medium=context&itm_campaign=sql&itm_content=none&cID=70170000000kgEZ)

**Column Level Encryption**

Column level encryption is a direct encryption of data in a column. It requires code changes because it can be applied only while programming the database. It is a granular way of encrypting data so only specific columns, or parts of columns, can be encrypted. You can even use different encryption keys for different columns. Column level encryption requires binary columns to store encrypted data. Also, it stores the keys on the server, so SQL database needs to trust the server that stores the keys.

Column level encryption uses symmetric keys for encrypting the data because that helps maintain productivity, and each symmetric key is protected by an asymmetric key. You can use different algorithms for symmetric keys but AES is the best option. In addition, symmetric keys can also be protected with passwords or master keys.

In order to keep your keys safe, you can use SQL permissions to limit access to them to only trusted user accounts. Another important note is that encrypted data is not indexable; if you want to index it, you need to decrypt it first.

**Handpicked related content:**

* [SQL Server Encryption Explained: TDE, Column-Level Encryption and More](https://blog.netwrix.com/2019/06/13/sql-server-encryption-explained-tde-column-level-encryption-and-more/)

To implement column level encryption, we need to take the following steps:

1. First, we create the database master key to encrypt our symmetric key. This can be done using the **CREATE MASTER KEY** command and the **ENCRYPTION BY PASSWORD** argument, which defines the password to encrypt the key:

CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'P@$$w0rd'

GO

1. Then we have to create a symmetric key, but in order to secure it, we should make a digitally signed certificate. To create a certificate, use the **CREATE CERTIFICATE** command, which will be protected by our master key. Then we can create a symmetric key using the**CREATE SYMMETRIC KEY** command and the AES\_256 encryption algorithm.

CREATE CERTIFICATE SelfSignedCertificate WITH SUBJECT = 'Column Encryption';

GO

CREATE SYMMETRIC KEY SQLSymmetricKey

WITH ALGORITHM = AES\_256

ENCRYPTION BY CERTIFICATE SelfSignedCertificate;

GO

1. Next, we need to add a column to our table to store encrypted data.

USE DB;

GO

ALTER TABLE Testtable

ADD EncryptedColumn varbinary(MAX )NULL

GO

1. In order to encrypt the table data, we need to open our symmetric key and update the table.

OPEN SYMMETRIC KEY SQLSymmetricKey

DECRYPTION BY CERTIFICATE SelfSignedCertificate;

1. Once the symmetric key is open, we need to update the table.

UPDATE Testtable

SET [EncryptedColumn] = EncryptByKey(Key\_GUID('SQLSymmetricKey'), ColumnWithSensitiveData);

GO

select \* from Testtable

1. The encrypted column has been created; it contains the encrypted data of the *columnwithsensitivedata* To achieve our SQL Server security goal, we should drop the existing plain text *columnwithsensitivedata* column using the DROP command and retain only the encrypted column.

**SQL Server Row Level Security**

Row-level security (RLS) allows developers to control user access rights to information on a row by row basis; each user can see only that rows that they have access to. This security feature has high level set of permissions because it bypasses existing SQL Server security access lists. It consists of three parts:

* **Predicate function** — This is a schema function that regulates whether a user account executing the query has access to the row.
* **Security predicate** — A security predicate binds a predicate function to the table. There are two types of security predicates:
  + **Filter predicate** — This limits read operations (SELECT) on the row. This filtering is done silently without any event logs.
  + **Block predicate** — This limits write operations (INSERT) on the row. There are four types of BLOCK predicates: AFTER INSERT, BEFORE UPDATE, AFTER UPDATE and BEFORE DELETE operations.
* **Security policy** — This is a collection of security predicates grouped in a single new object.

**Handpicked related content:**

* [6 SQL Server Security Basics Every Database Admin Should Know](https://blog.netwrix.com/2019/05/30/6-sql-server-security-basics-every-database-admin-should-know/)

The logic is simple: A predicate function defines whether to allow or block rows a user can access based on certain parameters, and then the function is applied to a table with a [security policy](https://blog.netwrix.com/security-policy/) and becomes active. Let’s take a close look to this process:

1. First, we need to create a filter predicate For this example, we will allow access to a row for a user if it is matches the *privilegeuser* column of that row or if the user name is ADMIN.

CREATE FUNCTION dbo.NewPredicateFunction

( @privilegeuser AS SYSNAME )

RETURNS TABLE

WITH SCHEMABINDING

AS

RETURN SELECT 1 AS AccessRight

    WHERE @privilegeuser = USER\_NAME() OR USER\_NAME() = 'ADMIN'

GO

1. Next, we create a security policy by executing the following script, which adds the predicate function we just created as a filter predicate on our test table.

CREATE SECURITY POLICY PrivacyPolicy

ADD FILTER PREDICATE

dbo.NewPredicateFunction(privilegeuser) ON dbo.testtable

WITH (STATE = ON)

That’s all. Our security policy is active in our SQL database and will allow users to access only their row while allowing the ADMIN account to access all rows.