

Secure a Microsoft Fabric data warehouse

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/secure-data-warehouse-in-microsoft-fabric/1-introduction>

Introduction

Completed

Microsoft Fabric Data Warehouse is a complete platform for data, analytics, and AI (Artificial Intelligence). It refers to the process of storing, organizing, and managing large volumes of structured and semi-structured data.

In a warehouse, administrators have access to a suite of technologies aimed at safeguarding sensitive information. These security measures are capable of securing or masking data from users or roles without proper authorization, ensuring data protection across both Warehouse and SQL analytics endpoints. This ensures a smooth and secure user experience, with no need for alterations to the existing applications.

Understand security features

Data engineers who are often well-versed in the SQL engine and adept at using T-SQL, will find warehouses in Microsoft Fabric easy to use.

This is because warehouses are powered by the same SQL engine they're familiar with, enabling them to perform complex queries and data manipulations. The SQL engine's wide range of security features further allows for sophisticated security mechanism at the warehouse level.

- **Workspaces roles** – Designed to provide different levels of access and control within the workspace. You can assign users to the various workspace roles such as **Admin**, **Member**, **Contributor**, and **Viewer**. These roles are crucial for maintaining the security and efficiency of data warehousing operations within an organization.
- **Item permissions** – Individual warehouses can have item permissions assigned directly to them. The main intent behind assigning such permissions is to facilitate the sharing of the Warehouse for downstream use.

- **Data protection security** – For more precise control, you can use T-SQL to grant specific permissions to users. Warehouse supports a range of data protection features that enable administrators to shield sensitive data from unauthorized access. This includes object-level security for database objects, column-level security for table columns, row-level security for table rows using `WHERE` clause filters, and dynamic data masking to obscure sensitive data like email addresses. These features ensure data protection across Warehouses and SQL analytics endpoints without necessitating changes to applications.

In the next units, we'll explore various ways of enabling security in a warehouse, and how they can facilitate the tasks of keeping your data warehouse workload protected.

2. Explore dynamic data masking

<https://learn.microsoft.com/en-us/training/modules/secure-data-warehouse-in-microsoft-fabric/2-explore-dynamic-data-masking>

Explore dynamic data masking

Completed

Dynamic Data Masking (DDM) is a security feature that limits data exposure to nonprivileged users by obscuring sensitive information.

Dynamic data masking offers several key benefits that enhance the security and manageability of your data. One of the primary advantages is its real-time masking feature. When querying sensitive data, DDM applies dynamic masking to it in real time. This process means that the actual data is never exposed to unauthorized users, thus enhancing the security of your data. Furthermore, DDM is straightforward to implement. It doesn't require complex coding, making it accessible for users of all skill levels.

Another benefit of DDM is that the data in the database isn't changed when DDM is applied. Instead, the masking rules are applied to the query results. This benefit means that the actual data remains intact and secure, while nonprivileged users only see a masked version of the data.

Define masking rule

Dynamic data masking, which is set up at the column level, offers a suite of features including comprehensive and partial masking capabilities, along with a random masking function designed for numeric data.

Masking Type	Description	Use Case	Limitations	Masking Rule
Default	Full masking according to the data types of the designated fields.	Useful when you want to completely hide the actual data.	Completely mask the data. No information is visible.	<code>default()</code>
Email	Exposes the first letter of an email address and the constant suffix ".com"	Useful when you want to show that the data field contains an email without revealing the actual email.	Only applicable to email fields.	<code>email()</code>
Custom Text	Exposes the first and last 'n' characters and adds a custom padding string in the middle.	Useful when you want to partially hide the actual data.	Not suitable for numeric, date, and time data types.	<code>partial(prefix_padding, padding_string, suffix_padding)</code>
Random	Replaces any numeric or binary value with a random number within a specified range.	Useful when you want to hide the actual numeric or binary data.	Only applicable to numeric and binary data types.	<code>random(low, high)</code>

The `prefix_padding` and `suffix_padding` parameters in the `partial()` function specify the number of characters to expose at the beginning and end of the string, and the `padding_string` parameter specifies the string to use for masking the remaining characters.

The `low` and `high` parameters in the `random()` function specify the range of random numbers to generate.

These masking types help prevent unauthorized viewing of sensitive data by enabling administrators to specify how much sensitive data to reveal, with minimal effect on the application layer. They're applied to query results, so the data in the database isn't changed. This approach allows many applications to mask sensitive data without modifying existing queries.

Configure data masking

Let's consider an example of a warehouse that stores customer information. The warehouse contains a `Customer` table with fields such as `CustomerName`, `Email`, `PhoneNumber`, and `CreditCardNumber`.

To apply data masking on the `CustomerName`, `Email`, `PhoneNumber`, and `CreditCardNumber` columns, run the following command:

```
-- For Email
ALTER TABLE Customers
ALTER COLUMN Email ADD MASKED WITH (FUNCTION = 'email()');

-- For PhoneNumber
ALTER TABLE Customers
ALTER COLUMN PhoneNumber ADD MASKED WITH (FUNCTION = 'partial(0,"XXX-XXX-",4)');

-- For CreditCardNumber
ALTER TABLE Customers
ALTER COLUMN CreditCardNumber ADD MASKED WITH (FUNCTION = 'partial(0,"XXXX-XXXX-XX
```

View masked results

Without Dynamic Data Masking, if a nonprivileged user runs a query to fetch customer details, they might see something like this:

```
CustomerName: John Doe
Email: johndoe@contoso.com
PhoneNumber: 123-456-7890
CreditCardNumber: 1234-5678-9012-3456
```

However, with DDM applied to the `Email`, `PhoneNumber`, and `CreditCardNumber` fields, the same query would return:

```
CustomerName: John Doe
Email: j*****@contoso.com
PhoneNumber: XXX-XXX-7890
CreditCardNumber: XXXX-XXXX-XXXX-3456
```

As you can see, the sensitive data is hidden from the nonprivileged user, enhancing the security of your data. This scenario is a basic example of how Dynamic Data Masking works. It helps to ensure that sensitive data isn't exposed to users who don't have the necessary privileges to view it.

Note

Unprivileged users with query permissions can infer the actual data since the data isn't physically obfuscated.

DDM should be used as part of a comprehensive data security strategy that includes proper management of object-level security with SQL granular permissions and adherence to the principle of minimal required permissions.

3. Implement row-level security

Implement row-level security

Completed

Row-Level Security (RLS) is a feature that provides granular control over access to rows in a table based on group membership or execution context.

For example, in an e-commerce platform, you can ensure that sellers only have access to order rows that are related to their own products. This way, each seller can manage their orders independently, while maintaining the privacy of other sellers' order information.

If you have experience with SQL Server, you find that row-level security shares similar characteristics and features.

Protect your data

Row-Level Security (RLS) works by associating a function, known as a security predicate, with a table. This function is defined to return *true* or *false* based on certain conditions, typically involving the values of one or more columns in the table. When a user attempts to access data in the table, the security predicate function is invoked. If the function returns *true*, the row is accessible to the user; if it returns *false*, the row doesn't show up in the query results.

Depending on the business requirements, an RLS predicate can be as simple as `WHERE CustomerId = 29` or as complex as required.

This process is transparent to the user and is enforced automatically by SQL Server, ensuring consistent application of security rules.

Row-level security is implemented in two main steps:

- **Filter predicates** - It's an inline table-valued function that filters the results based on the predicate defined.

Access	Definition
SELECT	Can't view rows that are filtered.
UPDATE	Can't update rows that are filtered.
DELETE	Can't delete rows that are filtered.
INSERT	Not applicable.

- **Security policy** - It's a security policy that invokes an inline table-valued function to protect access to the rows in a table.

Because access control is configured and applied at the warehouse level, application changes are minimal - if any. Also, users can directly have access to the tables and can query their own data.

Configure row-level security

The T-SQL commands below demonstrate how to use RLS in a scenario where user access is segregated by tenant:

```
-- Create supporting objects for this example
CREATE TABLE [Sales] (SalesID INT,
    ProductID INT,
    TenantName NVARCHAR(50),
    OrderQty INT,
    UnitPrice MONEY)
GO

INSERT INTO [Sales] VALUES (1, 3, 'tenant1@contoso.com', 5, 10.00);
INSERT INTO [Sales] VALUES (2, 4, 'tenant2@contoso.com', 2, 57.00);
INSERT INTO [Sales] VALUES (3, 7, 'tenant3@contoso.com', 4, 23.00);
INSERT INTO [Sales] VALUES (4, 2, 'tenant4@contoso.com', 2, 91.00);
INSERT INTO [Sales] VALUES (5, 9, 'tenant5@contoso.com', 5, 80.00);

-- View all the rows in the table
SELECT * FROM Sales;
```

Next, we create a new schema, an inline table-valued function, and grant user access to the new function. The `WHERE @TenantName = USER_NAME() OR USER_NAME() = 'TenantAdmin'` predicate evaluates if the user name executing the query matches the *TenantName* column values.

```
--Create a schema
CREATE SCHEMA [Sec];
GO

--Create the filter predicate
CREATE FUNCTION sec.tvf_SecurityPredicatebyTenant(@TenantName AS NVARCHAR(10))
    RETURNS TABLE
WITH SCHEMABINDING
AS
    RETURN      SELECT 1 AS result
                WHERE @TenantName = USER_NAME() OR USER_NAME() = 'tenantAdmin'
GO

--Create security policy and add the filter predicate
CREATE SECURITY POLICY sec.SalesPolicy
ADD FILTER PREDICATE sec.tvf_SecurityPredicatebyTenant(TenantName) ON [dbo].[Sales]
WITH (STATE = ON);
GO
```

The `tenantAdmin@contoso.com` user should see all the rows. The `tenant1@contoso.com` to `tenant5@contoso.com` users should only see their own rows.

If you alter the security policy with `WITH (STATE = OFF);`, you notice that users see all the rows.

Note

There is a risk of information leakage if an attacker writes a query with a specially crafted `WHERE` clause and, for example, a divide-by-zero error, to force an exception if the `WHERE` condition is true. This is known as a *side-channel attack*.

Explore use cases

Row-level security is ideal for many scenarios, including:

- When you need to isolate departmental access at the row level.
- When you need to restrict customers' data access to only the data relevant to their company.
- When you need to restrict access for compliance purposes.

Apply best practices

Here are a few best practices to consider when implementing RLS:

- It's recommended to create a separate schema for predicate functions, and security policies.
- Whenever possible, avoid type conversions in predicate functions.
- To maximize performance, avoid using excessive table joins and recursion in predicate functions.

4. Implement column-level security

<https://learn.microsoft.com/en-us/training/modules/secure-data-warehouse-in-microsoft-fabric/4-implement-column-level-security>

Implement column-level security

Completed

Column-level security (CLS) allows you to restrict column access in order to protect sensitive data. It provides granular control over who can access specific pieces of data, enhancing the overall security of your data warehouse.

Secure sensitive data

Let's consider a practical example of column-level security (CLS) in the healthcare industry. Suppose we have a table named `Patients` with the following columns: `PatientID`, `Name`, `Address`, `DateOfBirth`, and `MedicalHistory`.

The `MedicalHistory` column contains sensitive health information about the patients. As per the healthcare regulations and privacy laws, this information should only be accessible to authorized medical personnel such as doctors or nurses.

Here's how you might implement column-level security in this scenario:

1. **Identify the sensitive columns:** In this case, the `MedicalHistory` column is identified as containing sensitive data.
2. **Define access roles:** Define roles such as `Doctor` and `Nurse` who are allowed to access the `MedicalHistory` column. Other roles such as `Receptionist` or `Patient` might be restricted from accessing this column.
3. **Assign roles to users:** Assign the appropriate roles to each user in the warehouse. For example, user `DrSmith` might be assigned the `Doctor` role, while user `JohnDoe` might be assigned the `Patient` role.
4. **Implement access control:** Restrict access to the `MedicalHistory` column based on the user's role.

Column-level security can help ensure that sensitive health information is only accessible to those who are authorized to see it, though protecting patient privacy and complying with healthcare regulations.

Configure column-level security

In the scenario we've recently explored, the syntax to use for implementing column-level security might look as follows:

```
-- Create roles
CREATE ROLE Doctor AUTHORIZATION dbo;
CREATE ROLE Nurse AUTHORIZATION dbo;
CREATE ROLE Receptionist AUTHORIZATION dbo;
CREATE ROLE Patient AUTHORIZATION dbo;
GO

-- Grant SELECT on all columns to all roles
GRANT SELECT ON dbo.Patients TO Doctor;
GRANT SELECT ON dbo.Patients TO Nurse;
GRANT SELECT ON dbo.Patients TO Receptionist;
GRANT SELECT ON dbo.Patients TO Patient;
GO
```

```
-- Deny SELECT on the MedicalHistory column to the Receptionist and Patient roles
DENY SELECT ON dbo.Patients (MedicalHistory) TO Receptionist;
DENY SELECT ON dbo.Patients (MedicalHistory) TO Patient;
GO
```

In this example, we first create the roles `Doctor`, `Nurse`, `Receptionist`, and `Patient`. We then grant `SELECT` permissions on all columns in the `Patients` table to all roles. Finally, we deny `SELECT` permissions on the `MedicalHistory` column to the `Receptionist` and `Patient` roles. This ensures that only users with the `Doctor` or `Nurse` role can access the `MedicalHistory` column.

Understand the benefits

In the realm of warehouse security, two commonly used techniques are column-level security and views. Both methods serve to restrict access to sensitive data, but they do so in different ways and offer different advantages. The following table provides a comparative analysis of these two techniques across various aspects such as granularity of access control, maintenance, performance, transparency, and flexibility.

This comparison can help you understand the strengths and weaknesses of each method and guide you in choosing the most suitable approach for your specific application requirements.

Aspect	Column-Level Security	Views
Granularity of Access Control	Allows control at a more granular level. Can specify different access rights for different users or roles on different columns within the same table.	Would need to create different views for different sets of permissions.
Maintenance	Permissions are tied to the columns themselves, so they automatically adapt to changes in table structure.	If the underlying table structure changes, views may need to be updated to reflect these changes.
Performance	Generally more efficient because it operates directly on the table data.	Can introduce performance overhead, especially if they are complex or if the underlying tables are large.
Transparency	The restrictions are transparent to the user. The user queries the table as usual, and the database engine takes care of applying the security rules.	The user needs to query a different object (the view instead of the table).
Flexibility	Less flexible than views.	Very flexible and can provide row-level security (by including a <code>WHERE</code> clause in the view definition) in addition to column-level security. They can also transform the data (for example, by calculating

Aspect	Column-Level Security	Views
		derived columns) which is not possible with column-level security.

The choice between using column-level security or views will depend on the specific requirements of your application. Always make sure to test any security changes in a safe environment before applying them to a production warehouse.

5. Configure SQL granular permissions using T-SQL

<https://learn.microsoft.com/en-us/training/modules/secure-data-warehouse-in-microsoft-fabric/5-configure-sql-granular-permissions>

Configure SQL granular permissions using T-SQL

Completed

If you're familiar with relational databases and enterprise warehouses, it's common knowledge that there are four fundamental permissions governing [Data Manipulation Language \(DML\)](#) operations. These permissions, namely `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, are universally applicable across all database platforms.

All of these permissions can be granted, revoked or denied on tables and views. If a permission is granted using the `GRANT` statement, then the permission is given to the user or role referenced in the `GRANT` statement. Users can also be denied permissions using the `DENY` command. If a user is granted a permission and denied the same permission, the `DENY` will always supersede the grant, and the user will be denied access to the specific object.

Table and view permissions

Tables and views represent the objects on which permissions can be granted within a warehouse. Within those tables and views, you can additionally restrict the columns that are accessible to a given security principal.

Permission	Definition
<code>SELECT</code>	Allows the user to view the data within the object (table or view). When denied, the user will be prevented from viewing the data within the object.

Permission	Definition
INSERT	Allows the user to insert data into the object. When denied, the user will be prevented from inserting data into the object.
UPDATE	Allows the user the update data within the object. When denied, the user will be prevented from updating data in the object.
DELETE	Allows the user to delete data within the object. When denied, the user will be prevented from deleting data from the object.

Function and stored procedure permissions

Like tables and views, functions and stored procedures have several permissions, which can be granted or denied.

Permission	Definition
ALTER	Grants the user the ability to change the definition of the object.
CONTROL	Grants the user all rights to the object.

Principle of least privilege

The basic idea of the principle of least privilege is that users and applications should only be given the permissions needed in order for them to complete the task. Applications should only have permissions that they need to do in order to complete the task at hand.

As an example, if an application accesses all data through stored procedures, then the application should only have the permission to execute the stored procedures, with no access to the tables.

Dynamic SQL

Dynamic SQL is a concept where a query is built programmatically. Dynamic SQL allows T-SQL statements to be generated within a stored procedure or a query itself. A simple example is shown below.

```
CREATE PROCEDURE sp_TopTenRows @tableName NVARCHAR(128)
AS
BEGIN
    DECLARE @query NVARCHAR(MAX);
    SET @query = N'SELECT TOP 10 * FROM ' + QUOTENAME(@tableName);
    EXEC sp_executesql @query;
END;
```

In this example, `@tableName` is the parameter that you can replace with the name of the table you want to inspect. The `QUOTENAME` function is used to safely quote the table name, preventing SQL

injection attacks. The `sp_executesql` stored procedure is then used to execute the dynamically built query.

Please note that this is a simple example and real-world data warehouse tasks might require more complex dynamic SQL queries. Always be cautious when using dynamic SQL due to the risk of SQL injection attacks. Always use parameterization methods like `sp_executesql` or `QUOTENAME` to sanitize inputs.

6. Exercise: Secure a warehouse in Microsoft Fabric

<https://learn.microsoft.com/en-us/training/modules/secure-data-warehouse-in-microsoft-fabric/6-exercise-secure-data-warehouse>

Exercise: Secure a warehouse in Microsoft Fabric

Completed

Now it's your chance to secure a warehouse in Microsoft Fabric.

In this exercise, you'll learn how to secure a warehouse using the concepts explored in this module.

Note

- You need a Microsoft Fabric trial license with the Fabric preview enabled in your tenant. See [Getting started with Fabric](#) to enable your Fabric trial license.
- This exercise includes optional steps that require a second user account to view the applied security measures. If you're unable to create a second account, you can still complete the exercise using your own account. In that case, please refer to the screenshots provided at the end of each section to see the expected results.

How to create a second user for this exercise

If you're part of an organization that has an Entra or Microsoft 365 tenant:

Work with your identity administrator to create the second user either in [Entra](#) or the [Microsoft 365 admin center](#).

If you're *not* a member of an organization with an Entra or Microsoft 365 tenant:

1. You're unable to sign up for a Fabric trial with your personal email address. You can sign up for a **Microsoft 365 trial** and a new organizational tenant will be created and you'll become the User and Billing administrator of the tenant and can then create users.
2. Create the second user in the **Microsoft 365 admin center** See: **Add users**
3. Enable the Fabric trial using **Getting started with Fabric**.

Launch the exercise and follow the instructions.

Launch Exercise

7. Module assessment

<https://learn.microsoft.com/en-us/training/modules/secure-data-warehouse-in-microsoft-fabric/7-knowledge-check>

Module assessment

Completed

8. Summary

<https://learn.microsoft.com/en-us/training/modules/secure-data-warehouse-in-microsoft-fabric/8-summary>

Summary

Completed

In this module, you learned about Dynamic Data Masking (DDM), Row-Level Security (RLS), Column-level security (CLS), and granular SQL permissions in Fabric warehouses.

The main takeaways from this module include understanding how DDM, RLS, and CLS work and their use cases. DDM operates at the column level, offering full, email, custom text, and random masking types. RLS works by associating a security predicate function with a table. CLS can be implemented by identifying sensitive columns, defining access roles, assigning roles to users, and implementing access control. In addition, you learned about the principle of least privilege, which suggests that users and applications should only be given the permissions necessary to complete their tasks.

For additional reading, you can refer to the following URLs:

- [Create a Warehouse in Microsoft Fabric](#)
- [Security for data warehousing in Microsoft Fabric](#)
- [Share your warehouse and manage permissions](#)