Lab: Secure Data in Synapse

Duration: 30 minutes

The main task for this exercise is as follows:

1. Row level Security
2. Column-level security
3. Dynamic data masking

Task 1: Row level Security

Row level Security (RLS) in Azure Synapse enables us to use group membership to control access to rows in a table.

Azure Synapse applies the access restriction every time the data access is attempted from any user.

Let see how we can implement row level security in Azure Synapse.

This lab uses the dedicated SQL pool. You should have paused the SQL pool at the end of the previous lab, so resume it by following these instructions:

1. Open Synapse Studio (https://web.azuresynapse.net/).
2. Select the **Manage** hub.
3. Select **SQL pools** in the left-hand menu. If the **SQLPool01** dedicated SQL pool is paused, hover over its name and select ▷.
4. When prompted, select **Resume**. It will take a minute or two to resume the pool.

Execute the following code in SQLPool01.

1. In Synapse Analytics Studio, navigate to the **Develop** hub.
2. In the **+** menu, select **SQL script**.
3. In the toolbar menu, connect to the **SQLPool01** database.
4. In the query window, replace the script with the following:

```
CREATE SCHEMA Sales;
GO;


CREATE TABLE Sales.Orders
(
        OrderID int,
        SalesRep nvarchar(50),
        Product nvarchar(50),
        Quantity smallint
);
GO;
```

```
INSERT INTO Sales.Orders  VALUES (1, 'SalesRep1', 'Valve', 5);
INSERT INTO Sales.Orders  VALUES (2, 'SalesRep1', 'Wheel', 2);
INSERT INTO Sales.Orders  VALUES (3, 'SalesRep1', 'Valve', 4);
INSERT INTO Sales.Orders  VALUES (4, 'SalesRep2', 'Bracket', 2);
INSERT INTO Sales.Orders  VALUES (5, 'SalesRep2', 'Wheel', 5);
INSERT INTO Sales.Orders  VALUES (6, 'SalesRep2', 'Seat', 5);
GO;

-- View the 6 rows in the table
SELECT * FROM Sales.Orders;
GO;

CREATE USER Manager WITHOUT LOGIN;
CREATE USER SalesRep1 WITHOUT LOGIN;
CREATE USER SalesRep2 WITHOUT LOGIN;
GO;

GRANT SELECT ON Sales.Orders TO Manager;
GRANT SELECT ON Sales.Orders TO SalesRep1;
GRANT SELECT ON Sales.Orders TO SalesRep2;
GO
```

Row-Level Security (RLS), 1: Filter predicates

We will define filter using what is called a Security Predicate. This is an inline table-valued function that allows us to evaluate additional logic.

-- Review any existing security predicates in the database

In the query window, replace the script with the following:

```
SELECT * FROM sys.security_predicates
```

Create a function.

In the query window, replace the script with the following:

```
CREATE FUNCTION Sales.tvf_securitypredicate(@SalesRep AS nvarchar(50))
    RETURNS TABLE
WITH SCHEMABINDING
AS
    RETURN SELECT 1 AS tvf_securitypredicate_result
WHERE @SalesRep = USER_NAME() OR USER_NAME() = 'Manager';
GO
```

Create a security policy adding the function as a filter predicate.

The state must be set to ON to enable the policy.

In the query window, replace the script with the following:

```
CREATE SECURITY POLICY SalesFilter
ADD FILTER PREDICATE Sales.tvf_securitypredicate(SalesRep)
ON Sales.Orders
WITH (STATE = ON);
GO
```

Review the security predicates in the database

In the query window, replace the script with the following:

```
SELECT * FROM sys.security_predicates
```

Now test the filtering predicate, by selected from the Sales table as each user.

In the query window, replace the script with the following:

```
EXECUTE AS USER = 'SalesRep1';
SELECT * FROM Sales.Orders;
REVERT;
```

In the query window, replace the script with the following:

```
EXECUTE AS USER = 'SalesRep2';
SELECT * FROM Sales.Orders;
REVERT;
```

The Manager should see all six rows. The Sales1 and Sales2 users should only see their own sales.

In the query window, replace the script with the following:

```
EXECUTE AS USER = 'Manager';
SELECT * FROM Sales.Orders;
REVERT;
```

Alter the security policy to disable the policy.

In the query window, replace the script with the following:

```
ALTER SECURITY POLICY SalesFilter
WITH (STATE = OFF);
```

Task 2: Column level Security

Column-level security feature in Azure Synapse simplifies the design and coding of security in application.

It ensures column level security by restricting column access to protect sensitive data.

In the query window, replace the script with the following:

```
REVOKE SELECT ON Sales.Orders from SalesRep1;

GRANT SELECT ON Sales.Orders([OrderID], [Product], [Quantity]) TO SalesRep1;
```

Now check if the security has been enforced, we execute the following query with current User As 'SalesRep1' this will result in an error since SalesRep1 doesn't have select access to the SalesRep column

In the query window, replace the script with the following:

```
EXECUTE AS USER ='SalesRep1';

select TOP 100 * from Sales.Orders;

REVERT
```

The following query will succeed since we are not including the SalesRep column in the query.

In the query window, replace the script with the following:

```
EXECUTE AS USER ='SalesRep1';

select [OrderID], [Product], [Quantity] from Sales.Orders;

REVERT
```

Task 3: Dynamic Data Masking

Dynamic data masking helps prevent unauthorized access to sensitive data by enabling customers to specify how much sensitive data to reveal with minimal impact on the application layer.

It is important to note that unprivileged users with ad-hoc query permissions can apply techniques to gain access to the actual data.

Ex: SELECT ID, Name, Salary FROM Employees WHERE Salary > 99999 and Salary < 100001;

This demonstrates that Dynamic Data Masking should not be used as an isolated measure to fully secure sensitive data from users running ad-hoc queries on the database.

Execute the following code in SQLPool01.

1. In Synapse Analytics Studio, navigate to the **Develop** hub.
2. In the **+** menu, select **SQL script**.
3. In the toolbar menu, connect to the **SQLPool01** database.
4. In the query window, replace the script with the following:

```
-- schema to contain user tables

CREATE SCHEMA Data;

GO
```

```
-- table with masked columns

CREATE TABLE Data.Membership(

   MemberID        int IDENTITY(1,1) NOT NULL,

   FirstName       varchar(100) MASKED WITH (FUNCTION = 'partial(1, "xxxxx", 1)') NULL,

   LastName        varchar(100) NOT NULL,

   Phone           varchar(12) MASKED WITH (FUNCTION = 'default()') NULL,

   Email           varchar(100) MASKED WITH (FUNCTION = 'email()') NOT NULL,

   DiscountCode    smallint MASKED WITH (FUNCTION = 'random(1, 100)') NULL

   );
```

```
-- inserting sample data
INSERT INTO Data.Membership VALUES ('Roberto', 'Tamburello', '555.123.4567', 'RTamburello@contoso.com', 10)
INSERT INTO Data.Membership VALUES ('Janice', 'Galvin', '555.123.4568', 'JGalvin@contoso.com.co', 5)
INSERT INTO Data.Membership VALUES ('Shakti', 'Menon', '555.123.4570', 'SMenon@contoso.net', 50)
INSERT INTO Data.Membership VALUES ('Zheng', 'Mu', '555.123.4569', 'ZMu@contoso.net', 40)
```

Let's see the Dynamic Data Masking (DDM) applied on columns.

```
SELECT c.name, tbl.name as table_name, c.is_masked, c.masking_function
FROM sys.masked_columns AS c
JOIN sys.tables AS tbl
    ON c.[object_id] = tbl.[object_id]
WHERE is_masked = 1;
```

A new user is created and granted the SELECT permission on the schema where the table resides.

```
-- Queries executed as the MaskingTestUser view masked data.
CREATE USER MaskingTestUser WITHOUT LOGIN;
GRANT SELECT ON SCHEMA::Data TO MaskingTestUser;
```

```
-- impersonate for testing:
EXECUTE AS USER = 'MaskingTestUser';
SELECT * FROM Data.Membership;
REVERT;
```

Adding or Editing a Mask on an Existing Column

```
ALTER TABLE Data.Membership
ALTER COLUMN LastName ADD MASKED WITH (FUNCTION = 'partial(2,"xxxx",0)');
```

The following example changes a masking function on the LastName column:

```
ALTER TABLE Data.Membership
ALTER COLUMN LastName varchar(100) MASKED WITH (FUNCTION = 'default()');
```

```
EXECUTE AS USER = 'MaskingTestUser';
SELECT * FROM Data.Membership;
REVERT;
```

Granting Permissions to View Unmasked Data

Granting the UNMASK permission allows MaskingTestUser to see the data unmasked.

```
GRANT UNMASK TO MaskingTestUser;
EXECUTE AS USER = 'MaskingTestUser';
SELECT * FROM Data.Membership;
REVERT;
```

```
-- Removing the UNMASK permission
REVOKE UNMASK TO MaskingTestUser;
```

Dropping a Dynamic Data Mask

The following statement drops the mask on the LastName column created in the previous example:

```
ALTER TABLE Data.Membership
ALTER COLUMN LastName DROP MASKED;
```

Let's confirm that the masking is removed on column LastName

```
SELECT c.name, tbl.name as table_name, c.is_masked, c.masking_function
FROM sys.masked_columns AS c
JOIN sys.tables AS tbl
    ON c.[object_id] = tbl.[object_id]
WHERE is_masked = 1;
```