

Dynamic Data Masking

Dynamic Data Masking (DDM) is a new feature introduced in SQL Server 2016. The purpose of DDM is to limit sensitive data exposure to the non-privileged users by masking it in a database. DDM helps prevent unauthorized access to sensitive data by enabling customers to designate how much of the sensitive data to reveal with minimal impact on the application layer. It's a policy-based security feature that hides the sensitive data in the result set of a query over designated database fields, while the data in the database is not changed.

This article explains how DDM works, how you can set it up, and manage permissions.

Adding Masking to a Column

Let's begin by creating a table with a mask over some of the data. Let's add a mask to one of the fields to start with in the table definition. Note that the way we do this is with a "MASKED WITH ()" format after the data type, but before the NULL and default options as shown below. Inside of the parenthesis we include FUNCTION = "", which specifies our function. Inside of the quotes, we specify the mask.

```
CREATE TABLE [dbo].[Customer]
  (SSN VARCHAR (10) MASKED WITH (FUNCTION = 'default()') DEFAULT ('000000000'))
, Name VARCHAR (200)
, Email VARCHAR (250)
)
GO
INSERT dbo.Customer
  (SSN , Name, Email)
VALUES
  ('1234567890', 'Bharath Malapati', 'Bharath.Malapati@Fiserv.com')
```

If I query this table as myself, I see a normal table. I get all the data, as it was inserted. This is because I am a privileged user. Those users with dbo privileges (db_owner or sysadmin roles), will not see masked data.

```
-- Execute as sysadmin/dbo
SELECT [SSN]
      ,[Name]
      ,[Email]
FROM [dbo].[Customer]
```



	SSN	Name	Email
1	1234567890	Bharath Malapati	Bharath.Malapati@Fiserv.com

Now I want to see a normal or non-privileged user. Let's create one test user without a login, but it works the same with any user. We need to grant normal SQL Server permissions to the test user to see the data in the table as shown below.

```
CREATE USER mytest WITHOUT LOGIN
GRANT SELECT ON [dbo].[Customer] TO mytest
```

When we now query the table with this user, we will see a different set of data.

```
-- Query as the test user
EXECUTE AS USER = 'mytest'
GO
SELECT [SSN]
       , [Name]
       , [Email]
FROM [dbo].[Customer]
GO
REVERT
GO
```



The screenshot shows a SQL Server query results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with three columns: 'SSN', 'Name', and 'Email'. The first row of data shows the SSN as 'xxxx', the Name as 'Bharath Malapati', and the Email as 'Bharath.Malapati@Fiserv.com'. The 'SSN' value is highlighted with a dashed border, indicating it is masked.

	SSN	Name	Email
1	xxxx	Bharath Malapati	Bharath.Malapati@Fiserv.com

We can see that the first column SSN contains masked data. Only x's appear in place of the data. This achieves what we want, which is hiding data from our non-privileged users. Note that the data is not changed on disk. The data is only masked when returned to non-privileged users.

You can see this occurring in the last part of the execution plan. We need to grant our test user rights to view the plan.

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 100%
 SELECT [SSN] , [Name] , [Email] FROM [dbo].[Customer]

Compute Scalar
 Compute new values from existing values in a row.

Physical Operation	Compute Scalar
Logical Operation	Compute Scalar
Estimated Execution Mode	Row
Estimated Operator Cost	0.0000001 (0%)
Estimated I/O Cost	0
Estimated CPU Cost	0.0000001
Estimated Subtree Cost	0.0032832
Estimated Number of Executions	1
Estimated Number of Rows	1
Estimated Row Size	245 B

There are other types of masks we can define on the table. There is a custom mask allowing control over what is shown, an email mask for email addresses, and a random mask for numbers.

Let's add masking to another column in our [dbo].[Customer] table. In this case, let's add a mask to the [Email] column and use the email mask. We will use MASKED WITH (FUNCTION='mask ()') format as shown below.

```
ALTER TABLE [dbo].[Customer]
ALTER COLUMN Email VARCHAR(250) MASKED WITH (FUNCTION='email()')
GO
```

Now if we query the table as test user, we'll see that both the SSN and Email columns are masked.

```
-- Query as the test user
EXECUTE AS USER = 'mytest'
GO
SELECT [SSN]
      , [Name]
      , [Email]
FROM [dbo].[Customer]
GO
REVERT
GO
```

Results Messages

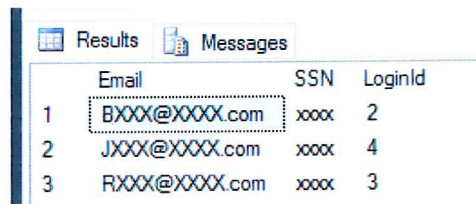
	SSN	Name	Email
1	xxxx	Bharath Malapati	BXXX@XXXX.com

We can certainly mask multiple columns in a table, each one separate from the others. Here we'll apply different masks to columns in a table.

```
CREATE TABLE [dbo].[User] (  
    Email VARCHAR (250) MASKED WITH (FUNCTION= 'email()')  
    , SSN VARCHAR (10) MASKED WITH (FUNCTION = 'default()')  
    , LoginId INT MASKED WITH (FUNCTION = 'random(1,4)')  
    )  
GO  
INSERT [dbo].[User]  
VALUES  
    ('Bharath.Malapati@Fiserv.com', '1234567890', 10000)  
    , ('John.Bishop@Fiserv.com', '0123456789', 55555)  
    , ('Rehan.Akhtar@Fiserv.com', '9876543210', 99999)  
GO  
GRANT SELECT ON [dbo].[User] TO mytest
```

Now let's query the table with our test user (mytest).

```
-- Query as the test user  
EXECUTE AS USER = 'mytest'  
GO  
SELECT [Email]  
    , [SSN]  
    , [LoginId]  
FROM [dbo].[User]  
GO  
REVERT  
GO
```



The screenshot shows a SQL Server query results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with three columns: 'Email', 'SSN', and 'LoginId'. The data is as follows:

	Email	SSN	LoginId
1	BXXX@XXXX.com	xxxx	2
2	JXXX@XXXX.com	xxxx	4
3	RXXX@XXXX.com	xxxx	3

As we can see, we get various masks from the different rows, each applied to the data in that particular row. You can experiment with the masks and defaults and can view the impact of masking.

If you execute this query multiple times, you will see different, random, values for the LoginId column as shown below.

```
-- Query as the test user  
EXECUTE AS USER = 'mytest'  
GO  
SELECT [Email]  
    , [SSN]  
    , [LoginId]  
FROM [dbo].[User]  
GO  
REVERT  
GO
```

	Email	SSN	LoginId
1	BXXX@XXXX.com	xxxx	1
2	JXXX@XXXX.com	xxxx	2
3	RXXX@XXXX.com	xxxx	1

Allowing Users to Unmask Data

There is a new permission in SQL Server 2016 for DDM. This is the UNMASK permission, which is granted like any other permission. Let's examine how this works.

Now we will grant the UNMASK permission to the test user as shown below and then re-query the table.

```
GRANT UNMASK TO mytest
GO
-- Query as the test user
EXECUTE AS USER = 'mytest'
GO
SELECT [Email]
      ,[SSN]
      ,[LoginId]
FROM [dbo].[User]
GO
REVERT
GO
```

	Email	SSN	LoginId
1	Bharath.Malapati@Fiserv.com	1234567890	10000
2	John.Bishop@Fiserv.com	0123456789	55555
3	Rehan.Akhtar@Fiserv.com	9876543210	99999

Now we can see that the data appears as it would for a privileged user. All the data is "unmasked" for the test user.

There is a downside to this, however. The UNMASK permission is granted database wide to users. There is no granularity by table or column. If a user has UNMASK, they can view all data in tables for which they have SELECT permission as it is stored in the database. We can see this by querying the first table with the test user.

```
-- Query as the test user
EXECUTE AS USER = 'mytest'
GO
SELECT [SSN]
      ,[Name]
      ,[Email]
FROM [dbo].[Customer]
GO
```



```
REVERT
GO
```



	SSN	Name	Email
1	1234567890	Bharath Malapati	Bharath.Malapati@Fiserv.com

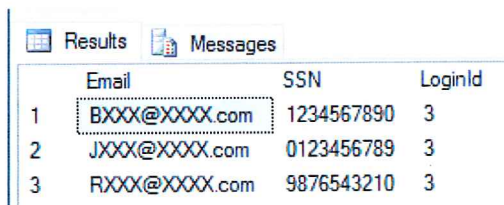
Removing Masks

If I decide that data no longer needs to be masked, the mask is removed with a simple ALTER TABLE statement as shown below.

```
ALTER TABLE [dbo].[User]
ALTER COLUMN [SSN] DROP MASKED;
```

Once we do this, our test user can see unmasked data for SSN column, as shown below.

```
DENY UNMASK TO mytest
GO
-- Query as the test user
EXECUTE AS USER = 'mytest'
GO
SELECT [Email]
       , [SSN]
       , [LoginId]
FROM [dbo].[User]
GO
REVERT
GO
```



	Email	SSN	LoginId
1	BXXX@XXX.com	1234567890	3
2	JXXX@XXX.com	0123456789	3
3	RXXX@XXX.com	9876543210	3

Note that the data for the SSN column is unmasked, but the data for Email and LoginId is still masked.

Conclusion

Dynamic Data Masking is a nice new feature designed to make protecting data from non-privileged users a bit easier for organizations. This can be implemented in the database, without any application code changes, allowing you to mask sensitive data from application users with a minimum of cost and effort.

I would also caution you that this is not really a security feature. The data stored on disk, and in your tables, is not changed in any way. This is still plain text data, and if your users have the ability to query the system, they can still potentially query your data and discover its value.

There are limitations in this first version of the feature, most notably around the UNMASK permission.