# Security, Permissions and Limitations with Dynamic Data Masking

In this article we will look at the various security permissions as well the limitations of dynamic data masking to hide data from users.

# The UNMASK Permission

Dynamic Data Masking follows the idea of providing security by default. We can see this in a table that has a mask applied to a column. If a user has SELECT permission to the table (or column), the default behaviour is that the mask is applied to the data. Users cannot get the original values unless they are explicitly granted permission to do so.

In the current version of Dynamic Data Masking (SQL Server 2016 as of this writing), there is only one permission associated with the feature. This is the UNMASK permission. When granted to a user, the user can see the original values in a table. The user does not need to change their query in any way. Just as Dynamic Data Masking masks data automatically in queries, the UNMASK permission automatically returns the original values of the data from any query.

The UNMASK permission is granted like any other user permission in SQL Server: with GRANT, REVOKE, and DENY. Let's see how this works in practice.

```
CREATE TABLE [dbo].[Devices](
	[device_id] [int] IDENTITY(1,1) NOT NULL,
	[user_id] [int] MASKED WITH (FUNCTION='random(500, 1000)') NOT NULL,
	[type] [varchar](50) NOT NULL,
	[label] [varchar](250) MASKED WITH (FUNCTION='default()') NOT NULL,
	[deleted] [char](1) NOT NULL,
	[delete_reason] [varchar](50) MASKED WITH (FUNCTION='partial(2, "xxx.xxx", 4)')
NULL,
	[create_time] [datetime] MASKED WITH (FUNCTION='default()') DEFAULT GETDATE()
NOT NULL,
 CONSTRAINT [PK_Devices] PRIMARY KEY CLUSTERED
(
	[device_id] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
INSERT dbo.Devices ([user_id],[type],[label],[deleted],[delete_reason])
  VALUES
    (1, 'Phone', '17041234000','N', NULL)
  , (2, 'Phone', '+64 220334880','Y','Device Inactive')
  , (3, 'Phone', '17042000000','Y','Device Disabled')
  , (4, 'Phone', '+919885523265','N', NULL)
  , (5, 'Phone', '+64 (223419149)','N', NULL)
GO
CREATE USER stest WITHOUT LOGIN
GRANT SELECT ON [dbo].[Devices] TO stest
GO
```

```
CREATE TABLE [dbo].[DeviceProperties](
	[device_id] [int] NOT NULL,
	[prop_key] [varchar](30) MASKED WITH (FUNCTION='partial(2, "xxx.xxx", 4)') NOT
NULL,
	[prop_val] [varchar](500) MASKED WITH (FUNCTION='default()') NOT NULL,
 CONSTRAINT [PK_DeviceProperties] PRIMARY KEY CLUSTERED
(
	[device_id] ASC,
	[prop_key] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
INSERT dbo.DeviceProperties
  VALUES
    (1, 'DeviceOS', 'Android')
  , (1, 'Firmware', '7.0.4')
  , (2, 'DeviceOS', 'iPhone OS')
  , (2, 'Firmware', '7.0.2')
GO
GRANT SELECT ON [dbo].[DeviceProperties] TO stest
GO
```

If we query this data as a non-privileged user, we find these masked data results:

```
-- Query as the test user
EXECUTE AS USER = 'stest'
GO
SELECT
	*
FROM
	dbo.Devices
GO
REVERT
GO
```

Results | Messages

| | device_id | user_id | type | label | deleted | delete_reason | create_time |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 791 | Phone | xxxx | N | NULL | 1900-01-01 00:00:00.000 |
| 2 | 2 | 632 | Phone | xxxx | Y | Dexxx xxxtive | 1900-01-01 00:00:00.000 |
| 3 | 3 | 864 | Phone | xxxx | Y | Dexxx xxxbled | 1900-01-01 00:00:00.000 |
| 4 | 4 | 674 | Phone | xxxx | N | NULL | 1900-01-01 00:00:00.000 |
| 5 | 5 | 644 | Phone | xxxx | N | NULL | 1900-01-01 00:00:00.000 |

```
-- Query as the test user
EXECUTE AS USER = 'stest'
GO
SELECT
	*
FROM
	dbo.DeviceProperties
GO
REVERT
GO
```

Results Messages

| | device_id | prop_key | prop_val |
|---|---|---|---|
| 1 | 1 | Dexxx xxxceOS | xxxx |
| 2 | 1 | Fixxx xxxware | xxxx |
| 3 | 2 | Dexxx xxxceOS | xxxx |
| 4 | 2 | Fixxx xxxware | xxxx |

As we can see the data is masked. Now let's allow our non-privileged test user stest to see the original data. Let's grant UNMASK as shown below.

```
GRANT UNMASK TO stest
GO
```

This time when we query the data as a non-privileged user stest, we find unmasked data results:

```
-- Query as the test user
EXECUTE AS USER = 'stest'
GO
SELECT
	*
FROM
	dbo.Devices
GO
REVERT
GO
```

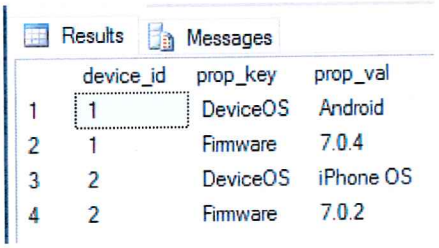
Results Messages

| | device_id | user_id | type | label | deleted | delete_reason | create_time |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | Phone | 17041234000 | N | NULL | 2016-09-13 19:28:43.313 |
| 2 | 2 | 2 | Phone | +64 220334880 | Y | Device Inactive | 2016-09-13 19:28:43.313 |
| 3 | 3 | 3 | Phone | 17042000000 | Y | Device Disabled | 2016-09-13 19:28:43.313 |
| 4 | 4 | 4 | Phone | +919885523265 | N | NULL | 2016-09-13 19:28:43.313 |
| 5 | 5 | 5 | Phone | +64 (223419149) | N | NULL | 2016-09-13 19:28:43.313 |

Nothing has changed in our code; only permissions were changed for this user.

However, we have a slight issue. There is also data masked in the DeviceProperties table. The prop_key and prop_val columns. If our non-privileged test user stest queries this table, this user can see original data.

```
-- Query as the test user
EXECUTE AS USER = 'stest'
GO
SELECT
	*
FROM
	dbo.DeviceProperties
GO
REVERT
GO
```



Results | Messages

| | device_id | prop_key | prop_val |
|---|---|---|---|
| 1 | 1 | DeviceOS | Android |
| 2 | 1 | Firmware | 7.0.4 |
| 3 | 2 | DeviceOS | iPhone OS |
| 4 | 2 | Firmware | 7.0.2 |

In fact, all masks are "removed" when the UNMASK permission is granted. When we add this permission to a database user, we don't specify any object (unlike for the SELECT permission).

We would have hoped for more granularity, and I expect this will be added in future versions, but for now our users either see masked data from all the tables or no masked data.

# Data Leakage

Dynamic Data Masking is really an application programming convenience feature, not a security feature. Despite it being marketed and documented in the security section, and despite the perspective of most of users, this feature only really limits access to data; it doesn't protect the data.

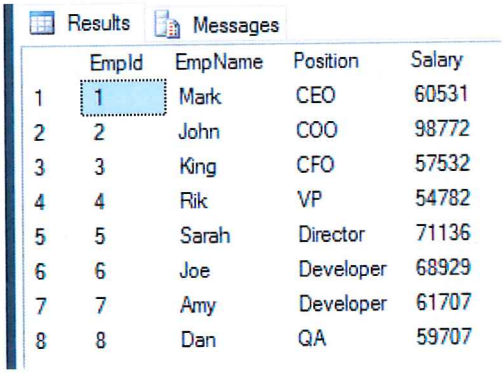
Any user who has the SQL knowledge can guess the masked data and expose the original values. There are a variety of ways this can be done, and it can be complex, but let's look at a simple example: employee salaries in a table.

Let's assume that we have an Employee table with the salary for each worker stored in the table. Let's look at the table DDL and sample data. Note: we are randomly masking the data.

```
CREATE TABLE [dbo].[Employee]
    (
      EmpId INT
    , EmpName VARCHAR(200)
    , Position VARCHAR(50)
    , Salary INT MASKED WITH (FUNCTION= 'random( 50000, 100000)' )
    );
GO
INSERT [dbo].[Employee]
    VALUES
        (1, 'Mark', 'CEO', 200000)
 ,      (2, 'John', 'COO', 150000)
 ,      (3, 'King', 'CFO', 145000)
 ,      (4, 'Rik', 'VP', 124000)
 ,      (5, 'Sarah', 'Director', 121000)
 ,      (6, 'Joe', 'Developer', 50000)
 ,      (7, 'Amy', 'Developer', 50000)
 ,      (8, 'Dan', 'QA', 40000);
GO
CREATE USER Joe WITHOUT LOGIN
GRANT SELECT ON [dbo].[Employee] TO Joe
GO
```

None of these employees has been granted the UNMASK permission. Therefore, if any of them query the table, they will get random values returned for the salary. We can see this if user, Joe, SELECTs all the data from the table.

```
EXECUTE AS USER = 'Joe'
GO
SELECT
	*
FROM
	dbo.Employee
GO
REVERT
GO
```



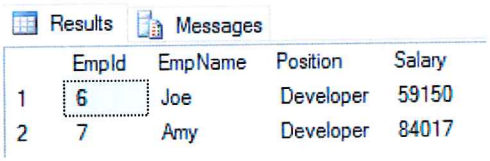
Results | Messages

| | EmpId | EmpName | Position | Salary |
|---|---|---|---|---|
| 1 | 1 | Mark | CEO | 60531 |
| 2 | 2 | John | COO | 98772 |
| 3 | 3 | King | CFO | 57532 |
| 4 | 4 | Rik | VP | 54782 |
| 5 | 5 | Sarah | Director | 71136 |
| 6 | 6 | Joe | Developer | 68929 |
| 7 | 7 | Amy | Developer | 61707 |
| 8 | 8 | Dan | QA | 59707 |

Joe knows that the Salary column in the dbo.Employee table is masked. Now Joe wants to know if Amy is making the same salary as him, so he decides to query the table. He knows his salary, so he uses this in a WHERE clause as shown below.

```
EXECUTE AS USER = 'Joe'
GO
SELECT
	*
FROM
	dbo.Employee
WHERE
	Salary = 50000
GO
REVERT
GO
```

He gets the following results, which show something we might not want Joe to know:



Results | Messages

| | EmpId | EmpName | Position | Salary |
|---|---|---|---|---|
| 1 | 6 | Joe | Developer | 59150 |
| 2 | 7 | Amy | Developer | 84017 |

Without knowing the value of Amy's salary, he can determine what it is. Even though the mask is in place, Joe can see whose salary matches with his salary.

Now, he modifies his query as shown below to see who makes more than he does.

```
EXECUTE AS USER = 'Joe'
GO
SELECT
	*
```

```
FROM
	dbo.Employee
WHERE
	Salary > 50000
GO
REVERT
GO
```

These results list all the employees who have higher salaries than Joe.

Results | Messages

| | EmpId | EmpName | Position | Salary |
|---|---|---|---|---|
| 1 | 1 | Mark | CEO | 54782 |
| 2 | 2 | John | COO | 71136 |
| 3 | 3 | King | CFO | 68929 |
| 4 | 4 | Rik | VP | 61707 |
| 5 | 5 | Sarah | Director | 59707 |

Once again, despite the mask, Joe has gained salary information about other employees. Even though the salary values aren't correct, Joe knows there's a domain of values that apply. Let's use this information further in a new query:

```
EXECUTE AS USER = 'Joe'
GO
CREATE TABLE #Numbers
	(
		n int NOT NULL PRIMARY KEY CLUSTERED
	)

	;WITH
	Digits (d) AS
	(
		SELECT 0 UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION
ALL SELECT 4 UNION ALL
		SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION
ALL SELECT 9
	),
	vii (d) AS
	(
		SELECT 0 UNION ALL SELECT 1000000
	)
	INSERT #Numbers WITH (TABLOCK) (n)
	SELECT
		n
	FROM
		(
			SELECT
				I.d
				+ 10 * II.d
				+ 100 * III.d
				+ 1000 * IV.d
				+ 10000 * V.d
				+ 100000 * VI.d
				+ VII.d
				AS n
			FROM
				Digits I
				CROSS JOIN Digits II
				CROSS JOIN Digits III
```

```
                        CROSS JOIN Digits IV
                        CROSS JOIN Digits V
                        CROSS JOIN Digits VI
                        CROSS JOIN VII
            ) AS N

SELECT e. EmpId
     , e. EmpName
     , e. Position
     , e. Salary
     , 'Real Salary' = t .n
  FROM dbo.employee e
    INNER JOIN #Numbers t
 on t .n = e.salary
WHERE salary > 50000

DROP TABLE #Numbers
GO
REVERT
GO
```

Now Joe has determined the actual value of all employees' salaries. This is because when the domain of value is known, or can be guessed, a non-privileged user can still query the actual values.

Results | Messages

| | EmpId | EmpName | Position | Salary | Real Salary |
|---|---|---|---|---|---|
| 1 | 1 | Mark | CEO | 90573 | 200000 |
| 2 | 2 | John | COO | 68908 | 150000 |
| 3 | 3 | King | CFO | 94367 | 145000 |
| 4 | 4 | Rik | VP | 55362 | 124000 |
| 5 | 5 | Sarah | Director | 67472 | 121000 |

In the results, you see the random, masked value, as well as the actual value in the far right column.

While this is a simple example, this could be extended. The domains for credit card numbers, social security numbers, and various other Personally Identifiable Information (PII) are known. With a tally table, a user can easily circumvent dynamic data masking to determine the actual values of data. This isn't limited to numbers. Tally tables (or a list of any data), can be used to determine the value of string data.

There is also data leakage in other ways. While a user copying data to a temp table or exporting data maintains the mask, data in statistics and CDC contains the actual value. We can see this by creating specific statistics on this table and examining them.

```
CREATE STATISTICS mystat ON dbo.Employee(Salary)

DBCC SHOW_STATISTICS (Employee, mystat)
```

When we examine the data, note that we see there are rows with various values. You can see that actual data is exposed. While this might not be as disturbing in a larger table, especially one that had substantially more rows than the 200 steps in the statistics histogram, there is still the potential for data leakage.

Results Messages

| | Name | Updated | Rows | Rows Sampled | Steps | Density | Average key length | String Index | Filter Expression | Unfiltered Rows |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | mystat | Sep 22 2016 6:56PM | 8 | 8 | 7 | 0 | 4 | NO | NULL | 8 |

| | All density | Average Length | Columns |
|---|---|---|---|
| 1 | 0.1428571 | 4 | Salary |

| | RANGE_HI_KEY | RANGE_ROWS | EQ_ROWS | DISTINCT_RANGE_ROWS | AVG_RANGE_ROWS |
|---|---|---|---|---|---|
| 1 | 40000 | 0 | 1 | 0 | 1 |
| 2 | 50000 | 0 | 2 | 0 | 1 |
| 3 | 121000 | 0 | 1 | 0 | 1 |
| 4 | 124000 | 0 | 1 | 0 | 1 |
| 5 | 145000 | 0 | 1 | 0 | 1 |
| 6 | 150000 | 0 | 1 | 0 | 1 |
| 7 | 200000 | 0 | 1 | 0 | 1 |

# Summary

There is only one permission associated with Dynamic Data Masking: the UNMASK permission. However this permission is globally applied at the database level, meaning that if a user has this permission, they have the ability to read the actual data in any column for which they have SELECT permission.

Dynamic Data Masking is also not really a security feature. As we showed, a user can determine the actual value of a row with a brute force attack by querying the domain of possible values. Even with a large, but known, domain of possible values, a patient hacker can submit multiple queries and slowly map the actual values of the data.

There are also other potential areas where data leakage can occur, the full list of which is not documented at this time. However users should be aware, again, that this is really a convenience feature for applications to obfuscate data, not a comprehensive security feature that protects data from unauthorized viewing.