SQL Server Security and Compliance

SQL 2016 new innovations

Ivan Kosyakov
Microsoft Technology Center, New York
Always Encrypted
Data disclosure prevention

Client-side encryption of sensitive data using keys that are *never* given to the database system

Queries on encrypted data

Support for equality comparison, including join, group by, and distinct operators

Application transparency

Minimal application changes via server and client library enhancements

*Allows customers to securely store sensitive data outside of their trust boundary.*

*Data remains protected from high-privileged, yet unauthorized, users.*
How it works
Help protect data at rest and in motion, on-premises and in the cloud

Encrypted sensitive data and corresponding keys are never seen in plaintext in SQL Server

SQL Server or SQL Database

"SELECT Name FROM Customers WHERE SSN = @SSN", 0x7ff654ae6d

Result Set

Name
Wayne Jefferson

Client

"SELECT Name FROM Customers WHERE SSN = @SSN", "111-22-3333"

Result Set

Name
0x19ca706fbd9a

dbo.Customers

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x19ca706fbd9a</td>
<td>0x7ff654ae6d</td>
<td>USA</td>
</tr>
</tbody>
</table>

trust boundary
Types of encryption for Always Encrypted

Randomized encryption
Encrypt('123-45-6789') = 0x17cfd50a
Repeat: Encrypt('123-45-6789') = 0x9b1fcf32
Allows for transparent retrieval of encrypted data but NO operations
More secure

Deterministic encryption
Encrypt('123-45-6789') = 0x85a55d3f
Repeat: Encrypt('123-45-6789') = 0x85a55d3f
Allows for transparent retrieval of encrypted data AND equality comparison
  E.g. in WHERE clauses and joins, distinct, group by

Two types of encryption available

Randomized encryption uses a method that encrypts data in a less predictable manner

Deterministic encryption uses a method that always generates the same encrypted value for any given plaintext value
Key provisioning

1. Generate CEKs and master key
2. Encrypt CEK
3. Store master key securely
4. Upload encrypted CEK to DB
Example

Client (trusted)

```
using (SqlCommand cmd = new SqlCommand(
    "SELECT Name FROM Customers WHERE SSN = @SSN",
    conn))
{
    cmd.Parameters.Add(new SqlParameter(
        "@SSN", SqlDbType.VarChar, 11).Value = "111-22-3333");
    SqlDataReader reader = cmd.ExecuteReader();
}
```

CMK store

```
EXEC sp_describe_parameter_encryption
@params = N'@SSN VARCHAR(11)' , @tsql = N'SELECT * FROM Customers WHERE SSN = @SSN'
```

SQL Server (untrusted)

```
exec sp_execute_sql
N'SELECT * FROM Customers WHERE SSN = @SSN' , @params = N'@SSN VARCHAR(11)' , @SSN = 0x7ff654ae6d
```

Result set (plaintext)

<table>
<thead>
<tr>
<th>Name</th>
<th>Jim Gray</th>
</tr>
</thead>
</table>

Result set (ciphertext)

| Name | 0x19ca706fbd9 |
Existing application setup

User experience: SSMS or SSDT (Visual Studio)

- Select candidate columns to be encrypted
- Analyze schema and application queries to detect conflicts and identify optimal encryption settings
- Select desired encryption settings for selected columns
- UI for selecting columns (no automated data classification)
- Schema/workload analysis tool analyzing schema and profiler logs
- UI for configuring encryption settings on selected columns (accepting/editing recommendations from analysis tool)
- Encrypt selected columns while migrating database to target server such as SQL Database
- Encryption tool creating new (encrypted) columns, copying data from old (plaintext) columns, swapping columns, and re-creating dependencies
- Set up keys
- Key setup tool to streamline selecting CMK, generating and encrypting CEK, and uploading key metadata to database
Select columns to be encrypted

UI for selecting columns (no automated data classification)

Set up keys: Master and CEK

Key setup tool to automate selecting CMK, generating and encrypting CEK, and uploading key metadata to database

Analyze schema and application queries to detect conflicts (build time)

Static schema analysis tool (SSDT only)

Setup (SSMS or SSDT)
User experience: SSMS or SSDT (Visual Studio)
Summary: Always Encrypted

Protect data at rest and in motion, on-premises and in the cloud

Capability

ADO.NET client library provides transparent client-side encryption, while SQL Server executes T-SQL queries on encrypted data.

Benefits

Sensitive data remains encrypted and queryable at all times on-premises and in the cloud.

Unauthorized users never have access to data or keys.

No application changes.
Dynamic Data Masking

SQL Server 2016
SQL Database
Dynamic Data Masking

Prevent abuse of sensitive data by hiding it from users

Configuration made easy in new Azure portal

Policy-driven at table and column level, for defined set of users

Data masking applied in real time to query results based on policy

Multiple masking functions available, such as full or partial, for various sensitive data categories (credit card numbers, SSN)
Benefits of Dynamic Data Masking

**Regulatory compliance**
A strong demand for applications to meet **privacy standards** recommended by regulating authorities.

**Sensitive data protection**
Protects against unauthorized access to sensitive data in application, and against exposure to developers or DBAs who need access to production database.

**Agility and transparency**
Data is masked on the fly, with underlying data in database remaining intact (transparent to application and applied according to user privilege).

*Limit access to sensitive data by defining policies to obfuscate specific database fields, without affecting database integrity.*
How it works

Limit sensitive data exposure by obfuscating it to non-privileged users

On-the-fly obfuscation of data in query results
Policy-driven on table and column
Multiple masking functions available for various sensitive data categories
Flexibility to define a set of privileged logins for unmasked data access
By default, database owner is unmasked

Dynamic data masking walkthrough

1) Security officer defines dynamic data masking policy in T-SQL over sensitive data in Employee table
2) Application user selects from Employee table
3) Dynamic data masking policy obfuscates the sensitive data in the query results

```
ALTER TABLE [Employee] ALTER COLUMN [SocialSecurityNumber]
ADD MASKED WITH (FUNCTION = 'SSN()')

ALTER TABLE [Employee] ALTER COLUMN [Email]
ADD MASKED WITH (FUNCTION = 'EMAIL()')

ALTER TABLE [Employee] ALTER COLUMN [Salary]
ADD MASKED WITH (FUNCTION = 'RANDOM(1,20000)')

GRANT UNMASK to admin1
```
Summary: Dynamic Data Masking

Capability

- Protects against unauthorized disclosure of sensitive data in application

Benefits

- Enables setup of policies at table and column level that provide multiple masking functions
- Allows certain privileged logins to see data unmasked
Row-Level Security
SQL Server 2016
SQL Database
The need for Row-Level Security

Protect data privacy by ensuring appropriate access across rows

Fine-grained access control over specific rows in database table

Blocking of unauthorized access when multiple users share tables, or connection filtering in multi-tenant applications

Administration via SQL Server Management Studio or SQL Server Data Tools

Enforcement logic inside database, with schema bound to table

Status: public preview
Benefits of Row-Level Security

**Fine-grained access control**
Keeping multi-tenant databases secure by limiting access by other users who share same tables

**Application transparency**
RLS works transparently at query time, without requiring app changes
Compatible with RLS in other leading products

**Centralized security logic**
Enforcement logic inside database that is schema-bound to protect table
Reduced application maintenance and complexity

*Store data intended for many consumers in a single database/table while also restricting row-level read-and-write access based on user execution context*
Common RLS use cases

**Traditional RLS workloads**

Custom business logic to determine which rows each user can SELECT, INSERT, UPDATE, and DELETE based on role, department, and security level

Target sectors: Finance, insurance, health care, energy, and government

**Multi-tenant databases**

Ensuring tenants can access only their own rows of data in a shared database, with enforcement logic in database rather than app tier

For example: Multi-tenant shards with elastic database tools in SQL Database

**Reporting, analytics, and data warehousing**

Different users access same database through various reporting tools, and work with different subsets of data based on their identity/role
**RLS concepts**

**Predicate function**

User-defined inline table-valued function (iTVF) implementing security logic

Can be arbitrarily complicated, containing joins with other tables

**Security predicate**

Binding of a predicate function to a particular table, applying it for all queries

Two types: filter predicates and blocking predicates (coming soon)

**Security policy**

Collection of security predicates for managing security across multiple tables

**CREATE SECURITY POLICY**

```
CREATE SECURITY POLICY mySecurityPolicy
ADD FILTER PREDICATE dbo.fn_securitypredicate(wing, startTime, endTime)
ON dbo.patients
```
CREATE FUNCTION dbo.fn_securitypredicate(@wing int)
    RETURNS TABLE WITH SCHEMABINDING AS
return SELECT 1 as [fn_securitypredicate_result] FROM
    StaffDuties d INNER JOIN Employees e
    ON (d.EmpId = e.EmpId)
    WHERE e.UserSID = SUSER_SID()
    AND @wing = d.Wing;

CREATE SECURITY POLICY dbo.SecPol
    ADD FILTER PREDICATE
dbo.fn_securitypredicate(Wing)
    ON Patients
    WITH (STATE = ON)

Fine-grained access control over rows in a table based on one or more pre-defined filtering criteria, such as user’s role or clearance level in organization

Concepts:
Predicate function
Security policy
RLS in three steps

1. Policy manager creates a filter predicate and security policy.
2. App user (e.g., nurse) selects from Patients table.

CREATE FUNCTION dbo.fn_securitypredicate(@wing int)
RETURNS TABLE WITH SCHEMABINDING AS
BEGIN
    RETURN SELECT 1 as [fn_securitypredicate_result] FROM
    StaffDuties d INNER JOIN Employees e
    ON (d.EmpId = e.EmpId)
    WHERE e.UserSID = SUSER_SID() AND @wing = d.Wing;
END;

CREATE SECURITY POLICY dbo.SecPol
ADD FILTER PREDICATE dbo.fn_securitypredicate(Wing) ON Patients
WITH (STATE = ON)

Filter Predicate:
INNER JOIN...

SELECT * FROM Patients
SELECT * FROM Patients
SEMIJOIN APPLY dbo.fn_securitypredicate(patients.Wing);

SELECT Patients.* FROM Patients,
    StaffDuties d INNER JOIN Employees e ON (d.EmpId = e.EmpId)
WHERE e.UserSID = SUSER_SID() AND Patients.wing = d.Wing;
Create security policy

-- The following syntax creates a security policy with a filter predicate for the Customer table, and leaves the security policy disabled
CREATE SECURITY POLICY [FederatedSecurityPolicy]
    ADD FILTER PREDICATE [rls].[fn_securitypredicate]([CustomerId])
    ON [dbo].[Customer];

-- Create a new schema and predicate function, which will use the application user ID stored in CONTEXT_INFO to filter rows.
CREATE FUNCTION rls.fn_securitypredicate (@AppUserId int)
    RETURNS TABLE
    WITH SCHEMABINDING
AS
RETURN (SELECT 1 AS fn_securitypredicate_result
    WHERE
        DATABASE_PRINCIPAL_ID() = DATABASE_PRINCIPAL_ID('dbo') -- application context
        AND CONTEXT_INFO() = CONVERT(VARBINARY(128), @AppUserId);
GO

Creates security policy for row-level security

The following examples demonstrate use of CREATE SECURITY POLICY syntax

For an example of a complete security policy scenario, see Row-Level Security
Summary: Row-Level Security

Capability

Row-Level Security provides fine-grained access control over rows in a table based on conditions you set up

Benefits

Store data for many users in same databases and tables while limiting access by other users who share same tables
Security enhancements

Built-in tools for enabling compliance: SQL Server audit tools

Create server audits, with server audit specifications (audited events can be written to event logs or to audit files)

**User-defined audit:** Allows middle-tier application to write custom events into audit log, which enables more flexibility to store audit information

**Audit filtering:** Provides greater flexibility to filter wanted events in audit log

**Audit resilience:** Audit logging is now tolerant to loss of connectivity to target directory and will recover automatically once network connection is re-established
Azure Key Vault support

Azure Key Vault:

Central key management that leverages hardware security modules (HSMs), separation of key management from data management

Support for AKV available through SQL Server Connector for AKV

Extensible Key Management (EKM) provider for SQL Server
Leverage Azure Key Vault for managing encryption keys
Both on-premises and SQL Server-in-a-VM users can assume control of encryption keys for Transparent Data Encryption (TDE), Column Level Encryption (CLE), and Backup Encryption while leveraging additional security benefits of Azure Key Vault
Transparent Data Encryption (TDE)

Encrypt data when it is stored on disk, and decrypt it when read into memory

- Developers can encrypt database files, log files, and backup files without changing existing applications
- Intel AES-NI hardware encryption acceleration
- Support for storage of memory-optimized OLTP tables (new)
- Exclusive to SQL Server Enterprise edition
Transparent Data Encryption (TDE)

Encrypt backup by specifying encryption algorithm and encryptor
Supports on-premises and Azure storage locations
Configurable for Managed Backup to Windows Azure

Backup encryption now supported with compression, using AES-NI hardware acceleration
Summary: Security

Always encrypted: Sensitive data always encrypted (and queryable)

Dynamic Data Masking: Real-time obfuscation of data

Row-Level Security: Fine-grained access control of table rows

Audit success/failure of database operations

TDE support for storage of In-Memory OLTP tables

Enhanced auditing for OLTP with ability to track history of record changes