Web Services SQLEKM Provider

WS SQLEKM User Guide

02 October 2024

Table of Contents

1. Introduction

This guide applies to nShield Web Services SQLEKM, which provides data-at-rest encryption for sensitive information held by Microsoft SQL Server.

The product works in combination with the nShield Web Services Option Pack, and nShield Hardware Security Modules (HSMs), to provide a high quality SQL Extensible Key Management (SQLEKM) provider. It is designed to be integrated into a Microsoft SQL Server database infrastructure with minimal disruption.

The nShield Web Services SQLEKM provider supports Transparent Data Encryption (TDE) and Cell-Level Encryption (CLE), and the concurrent use of both TDE and CLE.

1.1. Product configurations

For details of supported and tested versions, see Release notes.

1.2. Contacting Support

To obtain support for your product, visit https://nshieldsupport.entrust.com.

2. Overview

This section provides an overview of how the Extensible Key Management (EKM) API, as provided for Microsoft SQL Server, can be used to protect databases when using nShield Web Services SQLEKM. A brief description of how to perform encryption using Microsoft SQL Server with the Web Services SQLEKM provider is also given.

Encryption should be part of a wider scheme of security practices to protect your database assets that should take into account any regulatory or legal requirements for data protection. Administration and management of encryption within any organization is a serious issue that requires appropriate training and resources.

A Microsoft SQL Server service permits the creation of one or more databases. When a client request is made to the SQL Server, it determines which of the databases are the subject of the query, and loads data that is the subject of the query into available memory from disk storage.

From a security perspective, the Microsoft SQL Server supports the use of keys to protect its databases. These keys can be used to perform two levels of encryption.

- **Transparent Data Encryption (TDE)** is used to encrypt an entire database in a way that does not require changes to existing queries and applications. A database encrypted with TDE is automatically decrypted when SQL Server loads it into memory from disk storage, which means that a client can query the database within the server environment without having to perform any decryption operations. The database is encrypted again when saved to disk storage. When using TDE, data is not protected by encryption whilst in memory. Only one key at a time per database can be used for TDE.
- To use **Cell-Level Encryption (CLE)**, you must specify the data to be encrypted and the key(s) with which to encrypt it. CLE uses one or more keys to encrypt individual cells or columns. It gives the ability to apply fine-grained access policies to the most sensitive data in a database. Only the specified data is encrypted: other data remains unencrypted. This mode of encryption can minimize data exposure within the database server and client applications. You can apply CLE to database tables that are also encrypted using TDE. Note that when using CLE, data is only decrypted in memory when required for use. Separate data can be encrypted using different keys within the same data table.

There may be performance trade-offs between speed and security, regarding use of TDE or CLE, but these issues are beyond the scope of this document.

The Web Services SQLEKM provider supports the generation and management of these

keys. Authorized access to the secure environment of a HSM, and the keys under its protection, is controlled by a protection domain. To use a protection domain, you must first set up a database credential.

To read from or write to an encrypted database, a user must have *all* of the following:

- An authorized database login, with password, that maps to an appropriate database credential.
- Knowledge of the correct protection domain(s) and associated passphrase(s).
- For CLE, knowledge of the keys in use, and their passwords (if any).
- Knowledge of the appropriate encrypted database to read or write to.

If Security World data (or keys) are lost, they can be securely recovered from a backup as authorized through secure administrative means. It is important to maintain an up-to-date backup of your data.

When use of keys is legitimately made available to the database, the continuing security of data protected by those keys becomes dependent on access offered through SQL Server in accordance with your organization's security policies.

For more information about configuring the Web Services SQLEKM provider to perform encryption, see Using the SQLEKM provider.

2.1. Querying encrypted data

When the client sends a query to SQL Server, the Web Services SQLEKM provider checks the level of encryption on the database that is the subject of the query. If SQL Server uses a database that employs TDE, the process of loading the assigned keys and encrypting the database when it is stored is done automatically. The reverse decryption operation is also automatic when a TDE encrypted database needs to be used and is loaded into memory. If a database is encrypted using TDE only, this is transparent to the client or user who does not need to be aware of the encryption status or specify any encryption or decryption operations when querying the database. Backup and transaction logs are similarly encrypted.

CLE can be used with or without TDE. In either case, when using CLE the target data must be explicitly encrypted in memory before being stored, or explicitly decrypted after being loaded into memory from storage. You must specify:

- The fields to be encrypted or decrypted.
- The (correct) key to be used.

CLE is not automatic. If you use it, you must be aware of the encryption or decryption process.

Note that if TDE is used in combination with CLE, then after the CLE has been performed, the encrypted cells will be additionally encrypted by the TDE process when the data is stored. When the TDE process decrypts, the cells are returned to memory in their original encrypted form and must be decrypted a second time using the appropriate cell-level key. The database-level TDE processes remain automatic.

2.1.1. Example queries

The following example queries use a database table of customer information that includes first names, second names and payment card numbers. The queries concern the details of customers whose first names are Joe.

2.1.1.1. Example 1: TDE encryption/decryption only

In this example, the entire database is encrypted with TDE.

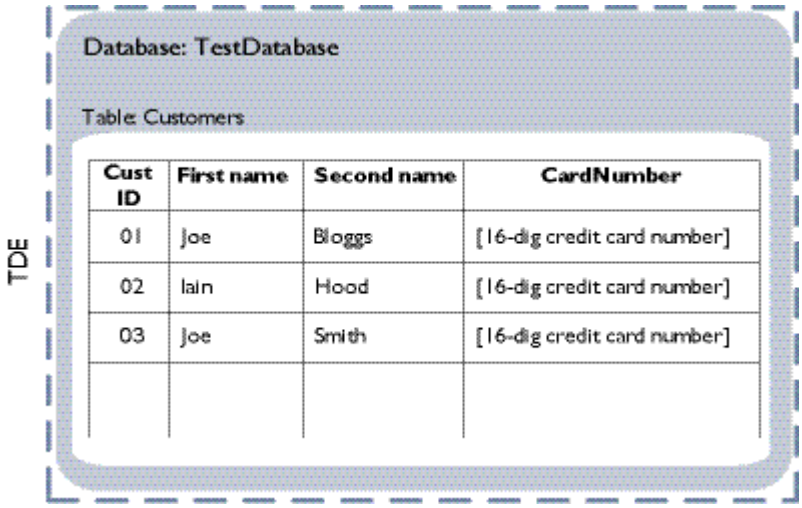

Figure 2.2: TDE encryption/decryption only

The database is decrypted when it is loaded into memory from disk storage. As this happens before the query is performed, the query does not have to specify any decryption operation:

```
USE TestDatabase
SELECT * FROM Customers WHERE
FirstName LIKE ('%Joe%');
```

2.1.1.2. Example 2: TDE combined with CLE/decryption

In this example, the database is encrypted with TDE, and the column of credit card numbers in the table of customers is additionally protected with CLE.

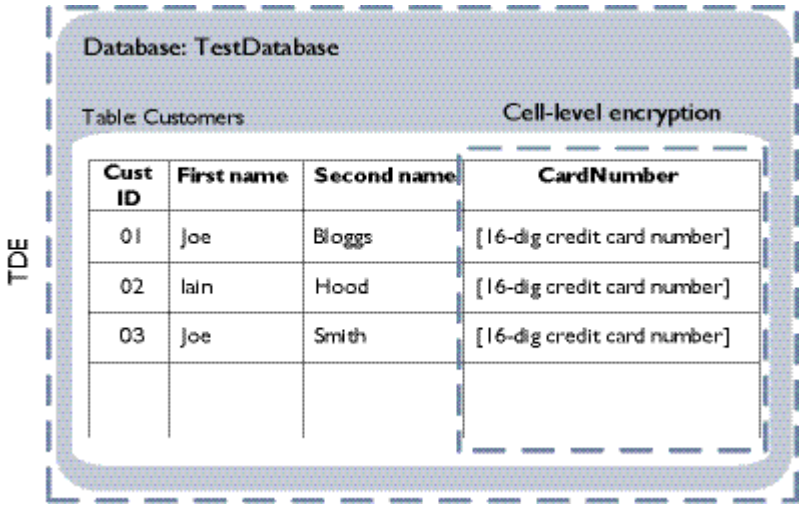

Figure 2.3: TDE and CLE/decryption

The query does not have to take account of TDE on the database because it is automatically decrypted on loading into memory from disk storage before the query is performed. However, the query must specify the (cell-level) decryption of the column of credit card numbers before the details of customers called 'Joe' can be returned.

```
USE TestDatabase
SELECT [FirstName], [SecondName], CAST(DecryptByKey(CardNumber) AS VARCHAR)
AS 'Decrypted card number'
FROM Customers WHERE [FirstName] LIKE ('%Joe%');
```

3. Installation

This section describes how to install and setup nShield Web Services SQLEKM for both standalone and clustered deployments.

3.1. Prerequisites to installing the nShield Web Services SQLEKM provider

- Ensure that `Microsoft Visual C++ 2015-2022 Redistributable (x64)` has been installed. A supported version is included on the nShield Web Services SQLEKM ISO under `redist` or can be downloaded from https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist?view=msvc-170.
- If you are using an SQL Server cluster, these installation steps should be repeated for each node in the cluster. The installation described here assumes Microsoft SQL Server and the nShield Web Services Option Pack are already installed.
- Ensure that all the latest service packs, updates and hotfixes for Microsoft SQL Server software have been added.
- SQL Server credentials and appropriate permissions are required for all users to install, configure, or use the Web Services SQLEKM provider.
- If you are intending to migrate keys from an existing nDSOP environment, see Migrating nDSOP keys.

3.2. Install nShield Web Services SQLEKM provider

1. Sign in as Administrator or as a user with local administrator rights.
2. Using the provided installation media, launch `setup.msi` manually.
3. Follow the onscreen instructions.
4. Accept the license terms and select **Next** to continue.
5. Specify the installation directory and select **Next** to continue.
6. Enter the Web Services Option Pack Server host and port number and select **Next** to continue.

   The host name must match the Web Services Option Pack Server's certificate's common name, see Install certificates for secure communication. The following registry entries can be used to change the host and port of the Web Services Option Pack Server:

   - `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\nCipher\SQLEKM\WebServices\Host` (type

REG_SZ)

- Computer\HKEY_LOCAL_MACHINE\SOFTWARE\nCipher\SQLEKM\WebServices\Port (type REG_DWORD)

7. Select **Install** to initiate installation.
8. Select **Finish** to complete the installation.

3.3. Install certificates for secure communication

The Web Services SQLEKM provider can only communicate securely with a Web Services Option Pack Server if the following certificates are installed:

- The Web Services Option Pack Server's CA certificate.
- An appropriate client certificate (with each SQL Server node in a cluster using its own client certificate).
- Any intermediate CA certificates that are to be used to form a complete chain to verify the client certificate on the Web Services Option Pack Server.

For information on these certificates, see nShield Web Services v3.3.1.

1. Install the Web Services Option Pack Server's CA certificate chain into the Root store using certutil.exe or a similar program.

```
certutil.exe -addstore Root <ca_certificate.pem>
```

2. Check that the certificate has been installed:

```
certutil.exe -store Root
```

3. Install any intermediate CA certificates for the client certificate:

```
certutil.exe -addstore CA <intermediate_ca_certificate.pem>
```

4. Install the client certificate and its private key with the ws-sqlekm-cert-install.exe tool. It is installed to the Web Services SQLEKM bin directory, for example C:\Program Files\nCipher\WebServices\SQLEKM\bin. It expects the client certificate to be a PFX file that contains a single certificate and the associated private key.

   The PFX must not contain the full certificate chain.

   To install the client certificate to a specific store, use the --install option, specify the PFX file, the associated password (if applicable), and the certificate store:

```
ws-sqlekm-cert-install.exe --install -p <password> -s My -x <client_certificate.pfx>
```

If the certificate is installed, the installer returns the `Operation successful` message.

The installation fails if:

- More than one certificate is found in the PFX file.
- A certificate containing the same subject name is found in the selected certificate store.

  Remove the existing certificate and reattempt installation.

If deploying a SQL Server cluster, each node should have its own client certificate.

3.4. Changes to the number of SQL Server instances

If the number of SQL Server instances changes, it will be necessary to ensure all instances are able to use the client certificate by running the following:

```
ws-sqlekm-cert-install.exe --pk-perms
```

4. Using the SQLEKM provider

This section shows you how to enable the Web Services SQLEKM provider, and provides examples showing how to encrypt and decrypt data.

To run these examples, open SQL Server Management Studio and connect to a SQL Server instance, then open a query window to execute a query. In the example T-SQL statements, the names used for keys (such as `dbAES256Key`) and databases (such as `TestDatabase`) are example names only. The exception is `master` database, which is a real database.

You must have a SQL Server login and appropriate permissions to configure or access Microsoft SQL Server or the Web Services SQLEKM provider.

4.1. Enabling the Web Services SQLEKM provider

1. To enable the Web Services SQLEKM provider, execute the following query:

```
sp_configure 'show advanced options', 1; RECONFIGURE;
GO
sp_configure 'EKM provider enabled', 1; RECONFIGURE;
GO
```

2. Register the Web Services SQLEKM provider by executing the following query:

```
CREATE CRYPTOGRAPHIC PROVIDER <provider name>
FROM FILE = '<provider path>';
GO
```

   Where:

   - `<provider name>` is the name that is used to refer to the Web Services SQLEKM provider, e.g. SQLEKM.
   - `<provider path>` is the fully qualified path to the `nShield.WebServices.SQLEKM.dll` file, e.g. `C:\Program Files\nCipher\WebServices\SQLEKM\provider\nShield.WebServices.SQLEKM.dll`.

3. To check that the Web Services SQLEKM provider is listed using SQL Server Management Studio.
   a. Go to **Security > Cryptographic Providers**. You should see **<provider name>**.

4.2. Creating a credential

A SQL Server credential represents the protection domain, and associated passphrase, that is used to authorize access to specific keys protected by the Security World. The protection domain must already exist before attempting to create a credential.

Keys are protected by a single protection domain. By implication, this also applies to the SQL Server credential that represents the protection domain.

Please refer to nShield Web Services v3.3.1 for further information about creating and managing protection domains.

We recommend the use of a strong passphrase. Please consult your organization's security policies.

Once created, the credential must in turn be associated with a login before it can be used. The owner of that login is then authorized to use that credential to create or use keys that are protected by the protection domain related to the credential.

It is by use of credentials and logins that access to keys for use in SQL Server can be controlled through the Web Services SQLEKM provider. For this reason, you should restrict who can use a credential. It is beyond the scope of this guide to deal with user access permissions. However, please be aware that if a valid credential and associated protection domain is available to an unauthorized user, who is then able to associate that credential with their login, this represents a security risk (the protection domain's password is stored in the credential and cannot be used to identify the user). This may be less of an issue when using TDE encryption, for which users authorized to access the database do not need an associated credential in any case, but it may be an issue with cell-level encryption.

You may use multiple credentials if you wish to simultaneously use TDE and cell-level encryption. You are advised to set up your cell-level credentials and associated keys first, before setting up the TDE login/credential and switching TDE on, see Transparent Data Encryption - TDE and Cell-Level Encryption (CLE).

To create a credential and map it to a login (also see Creating credentials):

1. In SQL Server Management Studio, navigate to **Security > Credentials**.
2. Right-click **Credentials**, then select **New Credential**.
3. Set **Credential name** to **<loginCredential>**.
4. Set **Identity** to `<protection domain id>`, where `<protection domain id>` matches the id of the protection domain. You must match the character case.
5. Set **Password** to `<protection domain passphrase>`, where `<protection domain`

passphrase> matches the passphrase of the protection domain. You must match the character case.

6. Ensure **Use Encryption Provider** is selected, then from the **Provider** drop-down list, choose **<provider name>**. Click **OK**.
7. Check that under **Security > Credentials** the name of the new credential appears. If necessary, right click and select **Refresh**.
8. In **SQL Server Management Studio**, navigate to **Security > Logins**.
9. Right-click to select the required login, then select **Properties**.
10. Ensure **Map to Credential** is selected, then select **loginCredential** from the drop down list. Click **Add**, then click **OK**.

4.3. Checking the configuration

To check that the Web Services SQLEKM provider was configured correctly:

1. Check that the Web Services SQLEKM provider was registered correctly by running the following query:

   ```
   SELECT * FROM sys.cryptographic_providers;
   ```

   A table is displayed with information about the registration of the Web Services SQLEKM provider. Check that:

   - The build version matches the `WS-SQLEKM` version number (found in the `version.json` file on the ISO).
   - The `.dll` path matches the path given when registering the provider (e.g. `C:\Program Files\nCipher\WebServices\SQLEKM\provider\nShield.WebServices.SQLEKM.dll`.)
   - The `is_enabled` column is set to `1`.
2. Check the Web Services SQLEKM provider properties by running the following query:

   ```
   SELECT * FROM sys.dm_cryptographic_provider_properties;
   ```

   A table is displayed with information about the properties of the Web Services SQLEKM provider. Check that:

   - `provider_version` matches the `WS-SQLEKM` version number (found in the `version.json` file on the ISO). The number may be in a different format, but digits should be the same.

- `friendly_name` is `nShield Web Services SQLEKM Provider`
- `authentication_type` is set to `BASIC`
- `symmetric_key_support` is set to `1`
- `asymmetric_key_support` is set to `1`

3. To check that the supported cryptographic algorithms can be queried, run the following query:

```
DECLARE @ProviderId int;
SET @ProviderId = (SELECT TOP(1) provider_id FROM sys.dm_cryptographic_provider_properties
WHERE friendly_name LIKE 'nShield Web Services SQLEKM Provider');
SELECT * FROM sys.dm_cryptographic_provider_algorithms(@ProviderId);
GO
```

A table is displayed with the supported cryptographic algorithms. For more information about the algorithms that should be displayed, see Supported cryptographic algorithms.

4.4. Encryption and keys

When you have configured the Web Services SQLEKM provider, and you have a suitable credential associated with your login, you can use the provider to:

- Manage keys within the nShield HSM.
- Encrypt or decrypt entire databases, or fields within tables, within SQL Server using TDE or cell-level encryption, or both at the same time.

Keys can be created in the Web Services SQLEKM provider and referenced by the appropriate database as required for use. When a reference of a key is no longer required for active use in the database, it should be deleted from the database while retaining the original copy of the key in the provider, which also acts as a secure backup. Storing original copies of keys in the Web Services SQLEKM provider is more secure than leaving key references and associated data together in the database. As long as the key is not deleted from the Web Services SQLEKM provider, its reference can be restored when next required for active use in the database.

Copying and deletion of keys does not apply to a TDE Database Encryption Key (TDEDEK), which is created as an integral part of a user database. On the other hand, this can apply to the wrapping key (TDEKEK) which is used to protect the TDEDEK. See Transparent Data Encryption - TDE.

It is recommended to regularly re-encrypt your data using fresh keys so that any persistent

attempts to decipher or compromise your encrypted data are impeded.

4.5. Key naming, tracking and other identity issues

Keys held in the database are really references to actual keys held in the Web Services SQLEKM provider. For the purpose of key tracking, it is suggested that you use the same name for both the database and the provider version of a key. Use a suffix or prefix to distin guish between the database and provider versions.

In a database there can be only one key with a specific name at any one time.

If you have a significant number of keys, you may wish to implement a key naming convention that helps you track which keys encrypt which data, backed up with some form of secure documentation. Note if a key naming convention incorporates a database identifier, a Security World can hold keys for more than one database at the same time, and a key can be used in more than one database at a time.

To use the examples in this document you will first need to create `TestDatabase` and `TestTable` as shown in Creating a database and Creating a table. Otherwise, provide your own database and table to perform encryption operations and adapt the examples accordingly. Refer to Verifying by inspection that TDE has occurred on disk before adapting any examples. See also T-SQL shortcuts and tips.

Keys created under a login that is mapped to a particular credential will be protected by that credential.

Please refer to nShield Web Services v3.3.1 for further information about listing keys.

4.6. Supported cryptographic algorithms

The Web Services SQLEKM provider supports the following algorithms: `AES_128`, `AES_192`, `AES_256`, `RSA_2048`, `RSA_3072` and `RSA_4096`.

4.6.1. Symmetric keys

4.6.1.1. Symmetric key GUIDs

When a new symmetric key is generated through the Web Services SQLEKM provider, it is associated in the database with a *Global Unique Identifier* or GUID. The database issues a

different and random GUID for every new key, and uses the GUID to identify the correct symmetric key for encryption or decryption purposes. As long as a copy of this key with the same GUID remains available to the database, it can be used indefinitely.

If the key is lost to the database, then a cryptographically equivalent duplicate can be gener ated through the Web Services SQLEKM provider from the copy stored in the HSM. The duplicate key, although cryptographically identical to the lost key, will be issued with a new GUID by the database. Because the GUID is different from the original key it will not be iden tified with the original key, and will not be allowed to perform encryption or decryption of the data with which the lost key was associated.

To avoid this issue, you should always specify an `IDENTITY_VALUE` when generating a sym- metric key. `IDENTITY_VALUE` is used to generate the key GUID in the database. The examples below create a symmetric key in the Web Services SQLEKM provider, and make available the same key for use in the database. The key does not have to share the same name between the Web Services SQLEKM provider and database.

4.6.1.2. Original key

To create a symmetric key with an identity value:

```
USE <database name>
CREATE SYMMETRIC KEY <key name in database> FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='<key name in provider>',
IDENTITY_VALUE='<unique GUID generator string>',
CREATION_DISPOSITION = CREATE_NEW, ALGORITHM=<symmetric algorithm>;
GO
```

Where

- `<database name>` is the name of the database for which you wish to provide encryption. See T-SQL shortcuts and tips for examples.
- `<provider name>` is the name of the provider you are using.
- `<key name in database>` is the name you wish to give the key in the database.
- `<key name in provider>` is the name you wish to give the key in the provider. Please note that there is a length restriction on this name of 31 characters maximum if created using a T-SQL query.
- `<unique GUID generator string>` is a unique string that will be used to generate the GUID.
- `<symmetric algorithm>` is a valid symmetric key algorithm descriptor.

If the value of the `<unique GUID generator string>` is known to an attacker, this will help them reproduce the symmetric key. Therefore, it

> should always be kept secret and stored in a secure place. We recommend the <unique GUID generator string> shares qualities similar to a strong passphrase. Check your organization's security policy.

Only one key that has been created using a particular IDENTITY_VALUE can exist at the same time in the same database.

4.6.1.2.1. Creating a duplicate key

This example shows how a duplicate of a lost symmetric key can be made through the Web Services SQLEKM provider and imported into the database.

To create a duplicate key:

```
USE <database name>
CREATE SYMMETRIC KEY <key name in database> FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='<key name in provider>',
IDENTITY_VALUE='<unique GUID generator string>',
CREATION_DISPOSITION = OPEN_EXISTING;
GO
```

Where <unique GUID generator string> is the same value as used to create the original key.

4.6.1.3. Creating and managing symmetric keys

This query generates a new symmetric key through the Web Services SQLEKM provider:

```
USE TestDatabase
CREATE SYMMETRIC KEY dbAES256Key
FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='ekmAES256Key',
IDENTITY_VALUE='Rg7n*9mnf29xl4',
CREATION_DISPOSITION = CREATE_NEW, ALGORITHM=AES_256;
GO
```

Where <provider name> is the name of the provider you are using.

In this example, the key is named dbAES256Key in the database and ekmAES256Key in the Web Services SQLEKM provider.

4.6.1.4. Listing symmetric keys in a database

To list the symmetric keys in a database using SQL Server Management Studio: . Go to **Databases > TestDatabase > Security > Symmetric Keys** (right-click to select **Refresh**).

Alternatively, you may check keys by following the methods shown in the section Checking keys.

4.6.1.5. Removing symmetric keys from the database only

To remove the symmetric key `dbAES256Key` from the database only (`TestDatabase`):

```
USE TestDatabase
DROP SYMMETRIC KEY dbAES256Key;
GO
```

After the above query completes, the key `dbAES256Key` is deleted from the database, but the corresponding key `ekmAES256Key` remains protected by the Web Services SQLEKM provider.

4.6.1.6. Re-importing symmetric keys

To re-import the symmetric key (`dbAES256Key`) that was removed from the database, where a corresponding copy (`ekmAES256Key`) exists in the Web Services SQLEKM provider:

```
USE TestDatabase
CREATE SYMMETRIC KEY dbAES256Key FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='ekmAES256Key',
IDENTITY_VALUE='Rg7n*9mnf29xl4',
CREATION_DISPOSITION = OPEN_EXISTING;
GO
```

This example uses the same `IDENTITY_VALUE` as in the original key generation. This regenerates the same GUID. Having the same GUID means that the key is logically identical to the key it replaces.

4.6.1.7. Removing symmetric keys from the database and provider

To remove a symmetric key (`dbAES256Key`) from both the database (`TestDatabase`) and the Web Services SQLEKM provider, execute the following query:

```
USE TestDatabase
DROP SYMMETRIC KEY dbAES256Key REMOVE PROVIDER KEY;
GO
```

Using this method means you do not have to name the corresponding key in the Web Services SQLEKM provider to remove it from there.

Refer to your security policies before considering deleting a key from

> the Web Services SQLEKM provider. You cannot import a key into the database once you have deleted that key from the provider.

4.6.2. Creating and managing asymmetric keys

4.6.2.1. Creating an asymmetric key

To generate a new asymmetric key through the Web Services SQLEKM provider:

```
USE TestDatabase
CREATE ASYMMETRIC KEY dbRSA2048Key FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='ekmRSA2048Key',
CREATION_DISPOSITION = CREATE_NEW, ALGORITHM=<asymmetric algorithm>;
GO
```

Where

- `<provider name>` is the name that is used to refer to the Web Services SQLEKM provider.
- `<asymmetric algorithm>` is a valid asymmetric key algorithm descriptor.

Please note that there is a length restriction on this name of 31 characters maximum if created using a T-SQL query.

This example names the key `dbRSA2048Key` in the database, and `ekmRSA2048Key` in the Web Services SQLEKM provider.

> `IDENTITY_VALUE` is not a supported argument for asymmetric key generation.

4.6.2.2. Listing asymmetric keys in a database

To list the asymmetric keys in a database using SQL Server Management Studio: . Go to **Databases > TestDatabase > Security > Asymmetric Keys** (right-click to select **Refresh**).

Alternatively, you may check keys by following the methods shown in the section Checking keys.

4.6.2.3. Removing an asymmetric key from the database only

To remove the asymmetric key `dbRSA2048Key` from the database only (`TestDatabase`):

```
USE TestDatabase
DROP ASYMMETRIC KEY dbRSA2048Key;
```

```
GO
```

After the above query completes, the key dbRSA2048Key is deleted from the database, but the corresponding key ekmRSA2048Key remains protected by the Web Services SQLEKM provider.

4.6.2.4. Re-importing an asymmetric key

To re-import a deleted asymmetric key (dbRSA2048Key) back into the database (TestDatabase), where a corresponding copy (ekmRSA2048Key) exists in the Web Services SQLEKM provider:

```
USE TestDatabase
CREATE ASYMMETRIC KEY dbRSA2048Key
FROM PROVIDER <provider name> WITH PROVIDER_KEY_NAME='ekmRSA2048Key',
CREATION_DISPOSITION = OPEN_EXISTING;
GO
```

4.6.2.5. Removing an asymmetric key from the database and provider

To remove the asymmetric key (dbAES256Key) from both the database (TestDatabase) and the Web Services SQLEKM provider, execute the following query:

```
USE TestDatabase
DROP ASYMMETRIC KEY dbRSA2048Key REMOVE PROVIDER KEY;
GO
```

Using this method means you do not have to name the corresponding key in the Web Services SQLEKM provider to remove it from there.

Refer to your security policies before considering deleting a key from the Web Services SQLEKM provider. You cannot import a key into the database once you have deleted that key from the provider.

4.6.2.6. Creating a symmetric wrapped key from an asymmetric wrapping key

To create a symmetric wrapped key (dbSymWrappedKey1) from an asymmetric wrapping key (dbAsymWrappingKey1), execute the following query:

```
USE TestDatabase
CREATE ASYMMETRIC KEY dbAsymWrappingKey1 FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='ekmAsymWrappingKey1',
CREATION_DISPOSITION = CREATE_NEW, ALGORITHM=RSA_2048;
CREATE SYMMETRIC KEY dbSymWrappedKey1
WITH ALGORITHM = AES_128,
```

```
IDENTITY_VALUE ='yr7s365$dfFJ901'
ENCRYPTION BY ASYMMETRIC KEY dbAsymWrappingKey1;
```

Where <provider name> is the name that is used to refer to the Web Services SQLEKM provider.

If you wish to delete the wrapped and wrapping keys, you will have to delete the wrapped key first.

4.6.3. Importing keys

By 'importing keys' we should distinguish between:

- Importing a key into the database that was created in the Web Services SQLEKM provider.
- Importing a (foreign) key that was created outside the Web Services SQLEKM provider.

Keys created in the Web Services SQLEKM provider can be imported into a database provided they are in simple format.

As regards keys created outside the Web Services SQLEKM provider, it is not recommended to import such keys into the Security World unless they are from a trustworthy source. Importing of externally created keys into the Security World may require format con version.

Please contact Entrust nShield Technical Support if you wish to pursue key import (or export) operations further.

To import an externally created symmetric key with an identity value:

```
USE <database name>
CREATE SYMMETRIC KEY <key name in database> FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='<key name in provider>',
IDENTITY_VALUE='<unique GUID generator string>',
CREATION_DISPOSITION = OPEN_EXISTING;
```

Where:

- <database name> is the name of the database for which you wish to provide encryption. See T-SQL shortcuts and tips for examples.
- <provider name> is the name of the provider you are using.
- <key name in database> is the name you wish to give the key in the database.
- <key name in provider> is the name of the externally created key in the provider. This

must be no more than 32 characters maximum.

- `<unique GUID generator string>` is a unique string that will be used to generate the GUID.

If the value of the `<unique GUID generator string>` is known to an attacker, this will help them reproduce the symmetric key. Therefore, it should always be kept secret and stored in a secure place. We recommend that the `<unique GUID generator string>` shares qualities similar to a strong passphrase. Check your organization's security policy.

Only one key that has been created using a particular `IDENTITY_VALUE` can exist at the same time in the same database.

To import an externally created asymmetric key

```
USE <database name>
CREATE ASYMMETRIC KEY <key name in database> FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='<key name in provider>',
CREATION_DISPOSITION = OPEN_EXISTING;
```

Parameters are the same as for the symmetric key. Note, for an externally created asymmetric key, name length restriction of 32 characters maximum applies for `<key name in provider>`

4.7. Transparent Data Encryption - TDE

These examples assume that both the `TestDatabase` and `TestTable` as described in T-SQL shortcuts and tips have been created, and are not currently encrypted.

When TDE encryption has been correctly set up and switched on, the database it is protecting will appear as normal to any user who has been granted suitable permissions to use the database. The user does not require a Web Services SQLEKM provider credential to access or modify TDE protected data.

Note that the person setting up or managing the TDE encryption keys must use the same protection domain for their login credential as used for the `tdeCredential` below.

The TDE Database Encryption Key (TDEDEK) is a symmetric key that is used to perform the actual encryption of the database. It is created by SQL Server and cannot be exported from the database meaning that it cannot be created or directly protected by the Web Services SQLEKM provider. In order to protect the TDEDEK within the database it may in turn be encrypted by a wrapping key. The wrapping key is called the TDE Key Encryption Key (TDEKEK). In this case, the Web Services SQLEKM provider can create and protect the

TDEKEK.

Before running the following examples, you should create a backup copy of the unencrypted database. See Backing up a database with SQL Server Management studio. Alternatively, you may prefer to adapt the T-SQL query shown in Making a database backup. Save the backup as `<Drive>:\<Backup_directory_path>\TestDatabase_TDE_Unencrypted.bak`.

If you are using a shared disk cluster, TDE should be configured on the active node.

4.7.1. Creating a TDEKEK

The TDEKEK must be protected under the same protection domain as that used to create the `tdeCredential` below.

To create a TDEKEK, or wrapping key, for database encryption:

```
USE master
CREATE ASYMMETRIC KEY dbAsymWrappingKey FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='ekmAsymWrappingKey', CREATION_DISPOSITION =
CREATE_NEW, ALGORITHM = RSA_2048;
GO
```

Where `<provider name>` is the name that is used to refer to the Web Services SQLEKM provider.

The TDEKEK is the only key you must create in the `master` database.

To check the TDEKEK, in SQL Server Management Studio navigate to **Databases > System Databases > Master > Security > Asymmetric Keys**. If necessary, right-click and select **Refresh**.

4.7.2. Setting up the TDE login and credential

1. In SQL Server Management Studio, navigate to **Security > Credentials**.
2. Right-click **Credentials**, then select **New Credential**.
3. Set **Credential name** to **tdeCredential** (for example).
4. Set **Identity** to `<protection domain id>`, where `<protection domain id>` matches the id of the protection domain. This must be the same key protector as that used to protect the `ekmAsymWrappingKey` created above.
5. Set **Password** to `<passphrase>`, where `<passphrase>` matches the passphrase of the protection domain.

6. Set **Use Encryption Provider** to `<provider name>`, where `<provider name>` is the name of the provider you are using. Click **OK**.
7. In SQL Server Management Studio, navigate to **Security > Logins**.
8. Right-click **Logins**, then select **New Login**.
9. Set **Login name** to **tdeLogin** (for example).
10. Ensure **Mapped to asymmetric key** is selected, then select `dbAsymWrappingKey` (the TDEKEK created previously) from the drop down list.
11. Ensure **Map to Credential** is selected, then select **tdeCredential** from the drop down list. Click **Add**, then click **OK**.
12. In SQL Server Management Studio, check that the **tdeCredential** exists by navigating to **Security > Credentials**. If necessary, right-click and select **Refresh**. You should see the credential name listed.
13. In SQL Server Management Studio, check that the **tdeLogin** exists by navigating to **Security > Logins**. If necessary, right-click and select **Refresh**. You should see the login name listed.

4.7.3. Creating the TDEDEK and switching on encryption

Only one TDEDEK per database can be used at a time.

To create the TDEDEK using the `dbAsymWrappingKey` (TDEKEK) created above for database encryption, and enable TDE on the database (`TestDatabase`):

1. In SQL Server Management Studio, navigate to **Databases > TestDatabase**.
2. Right-click `TestDatabase`, then select **Tasks > Manage Database Encryption...**
3. Set **Encryption Algorithm** to the desired algorithm.
4. Ensure that **Use server asymmetric key** is selected, then select `dbAsymWrappingKey` from the drop down list.
5. Ensure **Set Database Encryption On** is selected, then click **OK**.

After successfully setting up the TDE encryption, the person performing the set up no longer needs to use the same protection domain for their login credential as used for the `tdeCredential`.

4.7.4. Verifying by inspection that TDE has occurred on disk

Note that the inspection method will only work for data that can be backed up in the database (on disk) as human-readable character strings.

To check the encryption state of the database, refer to the section How to check the TDE encryption/decryption state of a database. If the TDE has been successful, then an 'Encrypted' state should be indicated.

Querying the `TestTable` or database contents will not indicate whether the table was encrypted on disk, because it will be automatically decrypted when loaded into memory. TDE encryption on disk can be verified by inspecting backup copies of the `TestDatabase` from before and after the TDE encryption.

After TDE encryption has been set up and checked to be functioning, make a backup copy of the encrypted `TestDatabase`: see Backing up a database with SQL Server Management studio for instructions.

You should now have the following unencrypted and encrypted backup copies of the `TestDatabase`:

- `<Drive>:\<Backup_directory_path>\TestDatabase_TDE_Unencrypted.bak`
- `<Drive>:\<Backup_directory_path>\TestDatabase_TDE_Encrypted.bak`

These backup files can be inspected using a text editor, provided you have appropriate access permissions.

1. Open `TestDatabase_TDE_Unencrypted.bak` in a text editor and search for a known value. It should be possible to find the plaintext `FirstName` or else `LastName` of anyone mentioned in the original and unencrypted `TestTable`.
2. Open `TestDatabase_TDE_Encrypted.bak` in a text editor and search for the same value. It should not be possible to find any plaintext names or other values in the encrypted file.

The backup files circumvent the automatic TDE decryption of the database, allowing direct inspection of the contents as stored on disk. Although this inspection has been carried out on backup files, these should contain information similar enough to the actual database disk contents to demonstrate whether the TDE encryption is working on disk or not.

4.7.5. To replace the TDEKEK

1. Follow the procedure above (see Creating a TDEKEK) to create a new asymmetric TDEKEK called `dbAnotherAsymWrappingKey`.
2. Create a new credential called `anotherTdeCredential`.
3. Create a new TDE login called `anotherTdeLogin`. Map it to `dbAnotherAsymWrappingKey` and the new `anotherTdeCredential`.
4. In SQL Server Management Studio, navigate to **Databases > TestDatabase**.

5. Right-click TestDatabase, then select **Tasks > Manage Database Encryption...**
6. Select **Re-Encrypt Database Encryption Key** and **Use server asymmetric**. Select `dbAnotherAsymWrappingKey` from the drop down list.
7. Ensure **Regenerate Database Encryption Key** is not selected.
8. Ensure **Set Database Encryption On** is selected, then click **OK**.

4.7.6. To replace the TDEDEK

1. In SQL Server Management Studio, navigate to **Databases > TestDatabase**.
2. Right-click **TestDatabase**, then select **Tasks > Manage Database Encryption...**
3. Ensure **Re-Encrypt Database Encryption Key** is not selected.
4. Ensure **Regenerate Database Encryption Key** is selected, then select **AES 256** from the drop down list.
5. Ensure **Set Database Encryption On** is selected, then click **OK**.

4.7.7. Switching off and removing TDE

See Uninstalling and upgrading.

4.7.8. How to check the TDE encryption/decryption state of a database

The following `encryption_state` information applies to TDE encryption only.

You can use the following T-SQL queries to find the current encryption state of a database. This can be particularly useful where large amounts of data have to be processed and you wish to check progress before attempting any further operations on the database.

First, find the database ID from the database name by using the following query:

```
SELECT DB_ID('<Database name>') AS [Database ID];
GO
```

Where `<Database name>` is the name of the database you are interested in.

List database encryption states by using the following query:

```
SELECT * FROM sys.dm_database_encryption_keys
```

The above query provides a table output that includes columns titled `database_id` and `encryption_state`.

Find the database ID you are interested in and look at the corresponding value for the encryption state.

Alternatively, you can use the composite query:

```
SELECT db_name(database_id), encryption_state
FROM sys.dm_database_encryption_keys
```

Where database_id is the ID number of the database you are interested in.

Values of `encryption_state` are as follows:

Value of `encryption_state`	**Meaning of value**
0	Encryption disabled (or no encryption key)
1	Unencrypted or Decrypted
2	Encryption in progress
3	Encrypted
4	Key change in progress
5	Decryption in progress
6	Protection change in progress (The certificate or asymmetric key that is encrypting the database encryption key is being changed.)

4.8. Cell-Level Encryption (CLE)

In CLE separate data fields in the same table can be encrypted under different keys. These keys can be protected by different credentials. Unlike TDE protection, the user will need to obtain keys from the Web Services SQLEKM provider, and must have the correct credential to authorize and load the key(s) for the specific encrypted data they wish to access. Non-encrypted data is not affected by this and is visible to any authorized user.

Cell-level encryption will only work on data stored in the database as VARBINARY type. You must provide any necessary type conversions so that data is in VARBINARY form before encryption is performed. Decryption will return the data to its original VARBINARY structure. It may then be necessary to reconvert to its original type for viewing in human-readable form.

Database backup files that use the VARBINARY type are not human-readable. Therefore, the previous inspection method, as used for TDE to directly check if data has been encrypted on disk, cannot be used for cell-level encryption.

If you have not already created the following keys and made them available in your current database copy, then create them now.

4.8.1. Symmetric key

```
USE TestDatabase
CREATE SYMMETRIC KEY dbAES256Key
FROM PROVIDER SQLEKM
WITH PROVIDER_KEY_NAME='ekmAES256Key',
IDENTITY_VALUE='Rg7n*9mnf29xl4',
CREATION_DISPOSITION = CREATE_NEW, ALGORITHM=AES_256;
GO
```

4.8.2. Asymmetric key

```
USE TestDatabase
CREATE ASYMMETRIC KEY dbRSA2048Key FROM PROVIDER SQLEKM
WITH PROVIDER_KEY_NAME='ekmRSA2048Key',
CREATION_DISPOSITION = CREATE_NEW, ALGORITHM=RSA_2048;
GO
```

4.8.3. Encrypting and decrypting a single cell of data

Before you start, make sure you have a fresh version of the `TestTable` that is unencrypted.

In the example below, the encrypted and decrypted data is stored separately. Normally, the original data would be overwritten with the processed data.

1. View `TestTable` by running the following query:

```
SELECT TOP 10 [FirstName]
,[LastName]
,CAST(NationalIdNumber AS decimal(16,0)) AS [NationalIDNumber]
,(NationalIdNumber) AS VarBinNationalIdNumber
,[EncryptedNationalIdNumber]
,[DecryptedNationalIdNumber]
FROM [TestDatabase].[dbo].[TestTable]
```

   You will see the column NationalIdNumber in its original decimal form, and the column

VarBinNationalIdNumber which shows the same number in its VARBINARY form (as stored in the database), and in which it will be encrypted.

The columns EncryptedNationalIdNumber and DecryptedNationalIdNumber should contain NULL.

2. To encrypt a single cell in the `TestTable`, run the following query:

```
USE TestDatabase
UPDATE TestTable
SET EncryptedNationalIDNumber = EncryptByKey(Key_GUID('dbAES256Key'),
NationalIDNumber)
WHERE FirstName = 'Kate' AND LastName = 'Austin';
GO
```

This query encrypts the `NationalIdNumber` for Kate Austin using the symmetric key `dbAES256Key`, and stores the result in the column `EncryptedNationalIDNumber`.

If encrypting using an asymmetric key, run the following query:

```
USE TestDatabase
UPDATE TestTable
SET EncryptedNationalIDNumber =
ENCRYPTBYASYMKEY(ASYMKEY_ID('dbRSA2048Key'), NationalIDNumber)
WHERE FirstName = 'Kate' AND LastName = 'Austin';
GO
```

3. Run the previous `View Table` query. The `EncryptedNationalIdNumber` will now contain the encrypted value against the name Kate Austin.
4. Run the following query to decrypt the information:

```
USE TestDatabase
UPDATE TestTable
SET DecryptedNationalIDNumber = DecryptByKey(EncryptedNationalIDNumber)
WHERE FirstName = 'Kate' AND LastName = 'Austin';
GO
```

If decrypting using an asymmetric key, run the following query:

```
USE TestDatabase
UPDATE TestTable
SET DecryptedNationalIDNumber =
DECRYPTBYASYMKEY(ASYMKEY_ID('dbRSA2048Key'), EncryptedNationalIDNumber)
WHERE FirstName = 'Kate' AND LastName = 'Austin';
GO
```

5. Run the previous `View Table` query. The `DecryptedNationalIdNumber` will now contain the decrypted value against the name Kate Austin.

Ensure that this value matches the corresponding value in the `VarBinNationalIdNumber` column. If the values match, then the decryption worked successfully.

6. To view the decrypted value in its original decimal form, run the following query:

```
SELECT TOP 10 [FirstName]
,[LastName]

,CAST(NationalIdNumber AS decimal(16,0)) AS [NationalIDNumber]
,(NationalIdNumber) AS VarBinNationalIdNumber
,[EncryptedNationalIdNumber]
,CAST(DecryptedNationalIdNumber AS decimal(16,0)) AS
[DecryptedNationalIdNumber]
FROM [TestDatabase].[dbo].[TestTable]
```

7. Reset the `EncryptedNationalIdNumber` and `DecryptedNationalIdNumber` columns by run ning the following query:

```
USE TestDatabase
UPDATE TestTable
SET EncryptedNationalIDNumber = NULL, DecryptedNationalIDNumber = NULL;
GO
```

4.8.4. Encrypting and decrypting columns of data

Before you start, make sure you have a fresh version of the `TestTable` that is unencrypted.

In the example below, the encrypted and decrypted data is stored separately. Normally, the original data would be overwritten with the processed data.

Perform the same steps as shown in the section Encrypting and decrypting a single cell of data, but in this case where encryption or decryption occurs, replace with the following queries.

- Encrypt an existing column of data using the symmetric key:

```
USE TestDatabase
UPDATE TestTable
SET EncryptedNationalIDNumber = EncryptByKey(Key_GUID('dbAES256Key'),
NationalIDNumber);
GO
```

- Decrypt an existing column of data using the symmetric key:

```
USE TestDatabase
UPDATE TestTable
SET DecryptedNationalIDNumber = DecryptByKey(EncryptedNationalIDNumber);
```

```
GO
```

- Encrypt an existing column of data using the asymmetric key:

```
USE TestDatabase
UPDATE TestTable
SET EncryptedNationalIDNumber = ENCRYPTBYASYMKEY(ASYMKEY_ID('dbRSA2048Key'), NationalIDNumber);
GO
```

- Decrypt an existing column of data using the asymmetric key:

```
USE TestDatabase
UPDATE TestTable
SET DecryptedNationalIDNumber = DECRYPTBYASYMKEY(ASYMKEY_ID('dbRSA2048Key'), EncryptedNationalIDNumber);
GO
```

4.8.5. Creating a new table and inserting cells of encrypted data

The following assumes you have available `TestDatabase` and the keys `dbAES256Key`, `dbRSA2048Key` as created previously.

4.8.5.1. Create a table with an encrypted field:

To create a new database table `Customers`, where individual cells of data held in the third column (`CardNumber`) will be encrypted, execute the following query:

```
USE TestDatabase
GO
CREATE TABLE Customers (FirstName varchar(MAX), SecondName varchar(MAX), CardNumber varbinary(MAX));
```

4.8.5.2. Insert encrypted data with the symmetric key:

The following query allows the user to enter the sensitive data (`CardNumber`) via the keyboard and then immediately encrypt using a symmetric key, sending the `CardNumber` directly into memory (and database) in an encrypted state.

```
USE TestDatabase
INSERT INTO Customers (FirstName, SecondName, CardNumber)
VALUES ('Joe', 'Bloggs', ENCRYPTBYKEY(KEY_GUID('dbAES256Key'),
CAST('<16 digit card number>' AS VARBINARY)));
INSERT INTO Customers (FirstName, SecondName, CardNumber)
VALUES ('Iain', 'Hood', ENCRYPTBYKEY(KEY_GUID('dbAES256Key'),
CAST('<16 digit card number>' AS VARBINARY)));
INSERT INTO Customers (FirstName, SecondName, CardNumber)
VALUES ('Joe', 'Smith', ENCRYPTBYKEY(KEY_GUID('dbAES256Key'),
CAST('<16 digit card number>' AS VARBINARY)));
GO
```

Where <16 digit card number> is a 16-digit payment card number to be encrypted.

4.8.5.3. View data encrypted with the symmetric key in plain text:

The following query allows the user to view, in plain text on screen, the sensitive data (CardNumber) for customers named 'Joe'. The data remains encrypted in memory and (database).

```
USE TestDatabase
SELECT [FirstName], [SecondName],
CAST(DecryptByKey(CardNumber) AS varchar) AS 'Decrypted card number'
FROM Customers WHERE [FirstName] LIKE ('%Joe%');
GO
```

If an asymmetric key (dbRSA2048Key) is used, similar actions can be achieved using the following queries.

4.8.5.4. Insert encrypted data with the asymmetric key:

```
USE TestDatabase
INSERT INTO Customers (FirstName, SecondName, CardNumber)
VALUES ('Joe', 'Connor', ENCRYPTBYASYMKEY(ASYMKEY_ID('dbRSA2048Key'),
CAST('<16 digit card number>' AS VARBINARY)));
INSERT INTO Customers (FirstName, SecondName, CardNumber)
VALUES ('Richard', 'Taylor', ENCRYPTBYASYMKEY(ASYMKEY_ID('dbRSA2048Key'),
CAST('<16 digit card number>' AS VARBINARY)));
INSERT INTO Customers (FirstName, SecondName, CardNumber)
VALUES ('Joe', 'Croft', ENCRYPTBYASYMKEY(ASYMKEY_ID('dbRSA2048Key'),
CAST('<16 digit card number>' AS VARBINARY)));
GO
```

Where <16 digit card number> is a 16-digit payment card number to be encrypted.

4.8.5.5. View data encrypted with the asymmetric key in plain text:

```
USE TestDatabase
SELECT [FirstName], [SecondName],
CAST(DECRYPTBYASYMKEY(ASYMKEY_ID('dbRSA2048Key'),CardNumber) AS varchar) AS 'Decrypted card number'
FROM Customers WHERE [FirstName] LIKE ('%Joe%');
GO
```

It is possible to encrypt separate table cells using different keys. When decrypting with a particular key, it should not be possible to see data that was encrypted using another key.

4.9. Viewing tables

4.9.1. Using SQL Server Management Studio

To check that data in a table was either encrypted or decrypted successfully, complete the following steps with SQL Server Management Studio: . Go to **Databases > TestDatabase > Tables**. . Right-click the table name and select **Select Top 1000 Rows** to view the encrypted or decrypted data.

4.9.1.1. Using SQL Query

To check that data in a table was either encrypted or decrypted successfully, execute the following SQL query:

```
Use TestDatabase
SELECT * FROM <table_name>
```

4.10. Checking keys

The following queries show how you can check the attributes of keys in your database and the Web Services SQLEKM provider.

- To view the symmetric keys in a database:

  ```
  Use TestDatabase
  SELECT * FROM sys.symmetric_keys
  ```

- To view the asymmetric keys in a database:

  ```
  Use TestDatabase
  SELECT * FROM sys.asymmetric_keys
  ```

- To list all the Web Services SQLEKM provider keys that are associated with the current credential:

  ```
  DECLARE @ProviderId int;
  SET @ProviderId = (SELECT TOP(1) provider_id
  FROM sys.dm_cryptographic_provider_properties
  WHERE friendly_name LIKE '<provider friendly_name>');
  SELECT * FROM sys.dm_cryptographic_provider_keys(@ProviderId);
  GO
  ```

  Where `<provider friendly_name>` can be found as shown in the section Checking the configuration for the provider you are using.

- To correlate symmetric keys between the database and the Web Services SQLEKM

provider:

```
DECLARE @ProviderId int;
SET @ProviderId = (SELECT TOP(1) provider_id FROM sys.dm_cryptographic_provider_properties
WHERE friendly_name LIKE '<provider friendly_name>');
SELECT * FROM sys.dm_cryptographic_provider_keys(@ProviderId)
FULL OUTER JOIN sys.symmetric_keys
ON sys.symmetric_keys.key_thumbprint = sys.dm_cryptographic_provider_keys.key_thumbprint
WHERE sys.dm_cryptographic_provider_keys.key_type = 'SYMMETRIC KEY'
GO
```

Where `<provider friendly_name>` can be found as shown in the section Checking the configuration for the cryptographic provider you are using.

- To correlate asymmetric keys between the database and the Web Services SQLEKM provider:

```
DECLARE @ProviderId int;
SET @ProviderId = (SELECT TOP(1) provider_id FROM sys.dm_cryptographic_provider_properties
WHERE friendly_name LIKE '<provider friendly_name>');
SELECT * FROM sys.dm_cryptographic_provider_keys(@ProviderId)
FULL OUTER JOIN sys.asymmetric_keys
ON sys.asymmetric_keys.thumbprint = sys.dm_cryptographic_provider_keys.key_thumbprint
WHERE sys.dm_cryptographic_provider_keys.key_type = 'ASYMMETRIC KEY'
GO
```

Where `<provider friendly_name>` can be found as shown in the section Checking the configuration for the cryptographic provider you are using.

- To correlate all keys (symmetric and asymmetric) between the database and cryptographic provider:

```
DECLARE @ProviderId int;
SET @ProviderId = (SELECT TOP(1) provider_id FROM sys.dm_cryptographic_provider_properties
WHERE friendly_name LIKE '<provider friendly_name>');
SELECT * FROM sys.dm_cryptographic_provider_keys(@ProviderId)
FULL OUTER JOIN sys.symmetric_keys
ON sys.symmetric_keys.key_thumbprint = sys.dm_cryptographic_provider_keys.key_thumbprint
FULL OUTER JOIN sys.asymmetric_keys
ON sys.asymmetric_keys.thumbprint = sys.dm_cryptographic_provider_keys.key_thumbprint
GO
```

Where `<provider friendly_name>` can be found as shown in the section Checking the configuration for the Web Services SQLEKM provider you are using.

The `key_thumbprint` returned by the T-SQL queries, maps to the `kid` when listing keys with the Web Services Option Pack REST API. Please refer to nShield Web Services v3.3.1 for further information about listing keys.

4.11. Restarting SQL Server

Changes to the Security World, or the Web Services Option Pack Server, may result in the need to restart SQL Server.

5. Database back-up and restore

It should be part of your corporate disaster recovery policy to perform regular back-ups of both your database and associated Security World such that the back-ups remain up to date and synchronized with each other.

If you are backing up a database that uses cell encryption keys, you should ensure that all sensitive data is encrypted first before back-up commences. Before back-up, remove the cell encryption key references from the database itself. If key references are not removed from the database, they will be stored within the database back-up. This should be avoided from a security point of view. If you are backing up a database that is both cell and TDE encrypted, perform the above instructions for the cell encryption keys before continuing with the following instructions for backing up a TDE encrypted database.

When backing up a TDE encrypted database, you must have the TDE credential (including protection domain) and database wrapping key (TDEKEK) present.

Once you have prepared the database as described above, you may back-up the database in a similar manner to an unencrypted database. If you are backing up a TDE encrypted data base, it will be backed up while remaining in its encrypted form, which is advantageous from a security point of view.

Your backup will include data content of your selected database, but may not include back-ups of SQL Server logins or credentials. Please refer to Microsoft SQL Server documentation for details of how to back these up. Otherwise, when later restoring the database, you may have to recreate suitable SQL Server logins and credentials, although this should not be a difficult task.

5.1. Backing up a database with SQL Server Management studio

This provides a basic example of how to backup a database. Please refer to Microsoft SQL Server documentation for a more thorough treatment of backup (and restoration) of a data base.

1. In SQL Server Management Studio, navigate to **Management**.
2. Right-click on **Management** and select **Back up**.
3. Set **Database_Name** using the pull down menu.
4. Set **Backup type** as **Full** using the pull down menu.
5. Set **Backup component** button as **Database**.

6. Under **Destination** select **Disk**.

Click **Remove** to set aside any previously named back-up file(s) that you do not want to keep. Click **Add** and provide a suitable path and name for the backup file, e.g. `<Drive>:<Backup_directory_path>\TestDatabase_TDE_[date].bak` Press **OK** to accept the file path and name. Press **OK** again. You must remove the existing entry as backup only allows a single entry to populate this field at any one time. Make sure that you rename with a meaningful and unique name for the backup and include the `bak` suffix.

7. When the back-up is complete, the message **The backup of database 'TestDatabase' completed successfully** is displayed. Press **OK**.
8. Make sure you can access the back-up file at the location given above.

If the database back-up fails with a message indicating that the transaction log is not up to date, repeat the above steps, but for step 4 select **Backup type** as **Transaction Log**. In step 6, provide a suitable **Log file name**. After this completes successfully, you should be able to perform the database back-up.

5.2. Restoring from a back-up

Restoring a database, including a TDE encrypted database, is similar in manner to an unencrypted database.

Once the database is restored, you will require suitable SQL Server logins and associated credentials to use the database and retrieve keys from the Security World. If these are not already present, or you have not restored them by some independent means, you will need to regenerate them. Once you have created a credential you must associate it with an authorized login. See Creating a credential for details of how to create a credential.

For cell encryption keys, once the database is restored with valid credentials and associated login, you can restore the cell encryption keys from the Web Services SQLEKM provider by reimporting them. But there is no need to do this until you need the keys. You must be using the correct credentials for the particular keys you wish to reimport, see Re-importing symmetric key or Re-importing an asymmetric key.

If you are restoring a database that uses both cell encryption and TDE encryption, then the database must first be restored for TDE encryption as shown below, before reimporting the cell encryption keys.

The following description focusses on restoring a TDE encrypted database. It assumes the database wrapping key (TDEKEK) has not been reimported into the master database.

Before proceeding to restore a TDE encrypted database:

- The user will need to use a personal login that is associated through a credential with the same protection domain that is protecting the TDEKEK for the database to be restored. If necessary, create a credential that uses this protection domain, and associate it with the user login.
- The database wrapping key (TDEKEK) should already exist and will need to be reimported into your master database using the 'OPEN_EXISTING' clause as in the example below.

```
USE master
CREATE ASYMMETRIC KEY dbAsymWrappingKey
FROM PROVIDER <provider name>
WITH PROVIDER_KEY_NAME='ekmAsymWrappingKey',
CREATION_DISPOSITION = OPEN_EXISTING;
GO
```

- You will need to recreate the TDE login and credential that was originally used with the database e.g.

```
Use master
CREATE LOGIN tdeLogin FROM ASYMMETRIC KEY dbWrappingKey;
CREATE CREDENTIAL tdeCredential WITH IDENTITY = '<protection domain id>', SECRET = '<protection domain
passphrase>'
FOR CRYPTOGRAPHIC PROVIDER SQLEKM;
ALTER LOGIN tdeLogin ADD CREDENTIAL tdeCredential;
```

- The user will need to use a personal login that is associated through a credential with the same protection domain that is protecting the TDEKEK for the database to be restored. If necessary, create a credential that uses this protection domain, and associate it with the user login.
- After setting up the TDEKEK and credentials above, you may now restore the TDE encrypted database in a similar manner to an unencrypted database. If the database was backed up in an encrypted state, it should be restored in an encrypted state, and you should not need to switch on encryption.

6. Uninstalling and upgrading

If you delete a provider login credential you will no longer be able to use it for the nShield Web Services SQLEKM provider.

If you delete an associated SQL Server login you will no longer be able to use it to access the SQL Server or the nShield Web Services SQLEKM provider and will be locked out.

6.1. Turning off TDE and removing TDE setup

You must turn off TDE on all your databases and remove TDE setup before uninstalling nShield Web Services SQLEKM. Otherwise, you will not be able to decrypt any databases encrypted with TDE.

Before disabling and removing TDE encryption you are advised to back up the encrypted database (see Backing up a database with SQL Server Management studio) and associated Security World.

1. In SQL Server Management Studio, navigate to **Databases > TestDatabase**.
2. Right-click **TestDatabase**, then select **Tasks > Manage Database Encryption...**
3. Ensure **Set Database Encryption On** is deselected, then click **OK**.
4. Wait for the decryption process to finish. Check this by referring to the section How to check the TDE encryption/decryption state of a database.
5. When the database has completed decryption, drop the encryption key using the following T-SQL query:

```
USE TestDatabase
DROP DATABASE ENCRYPTION KEY;
GO
```

6. Restart the database instance. If you are using a database failover cluster, you may have to do this directly on the active server. In SQL Server Management Studio right-click on the instance and select **Restart**.
7. In SQL Server Management Studio, navigate to **Security > Logins**, and select the TDE login you wish to remove (for example, **tdeLogin**). Right-click on the selected login and select **Properties**.
8. Ensure the associated credential (for example, **tdeCredential**) is highlighted then choose **Remove**. Untick the box **Map to credential**. Click **OK**.
9. In SQL Server Management Studio, navigate to **Security > Credentials**, and select the

same credential you previously removed from the login (for example, **tdeCredential**). Right-click on the credential and select **Delete**. In the following screen, select **OK**.

10. In SQL Server Management Studio, navigate to **Security > Logins**, and select the TDE login you wish to remove (for example, **tdeLogin**). Right-click on the selected login and select **Delete**. In the following screen, select **OK**.
11. In SQL Server Management Studio, navigate to **Databases > System Databases > master >Security > Asymmetric keys**.
    - Select the key you wish to remove (for example, **dbAsymWrappingKey**). Right-click on the key and select **Delete**.
    - Alternatively, you can use the following query:

```
USE master
DROP ASYMMETRIC KEY dbAsymWrappingKey REMOVE PROVIDER KEY;
GO
```

6.2. Uninstalling nShield Web Services SQLEKM

Do not uninstall nShield Web Services SQLEKM until you have:

- Decrypted any data encrypted using the nShield Web Services SQLEKM provider in all your databases.
- Turned off TDE.

To uninstall nShield Web Services SQLEKM:

1. Remove the loginCredential from the logged-in user:
    a. In SQL Server Management Studio, select **Security > Logins** and open up the properties of the logged-in user.
    b. Select **loginCredential**, then click **Remove**, then **OK**.
2. Select **Security > Credentials**, and delete the **loginCredential**.
3. Disable and remove the nShield Web Services SQLEKM provider:
    a. Select **Security > Cryptographic Providers**.
    b. Right-click to select the nShield Web Services SQLEKM provider and click **Disable Provider**.
    c. A dialog is displayed which shows that this action was successful. Click **Close**.
    d. Right-click to select the disabled nShield Web Services SQLEKM provider, then click **Delete**, then **OK**.
4. Select **Start > Control Panel > Administrative Tools > Services** (or **Start > Administrative Tools > Services**, depending on your version of Windows). Select **SQL Server**

(MSQLSERVER) and click **Action > Stop**.

5. Select **Start > Control Panel > Add/Remove programs** (or **Uninstall program**, depending on your version of Windows). Select **nShield Web Services SQLEKM** then click **Uninstall**.
6. A dialog is displayed asking if you want to continue with uninstalling nShield Web Services SQLEKM. Click **Yes**.
7. A setup status screen is displayed while nShield Web Services SQLEKM is uninstalled. Click **Finish** to complete the removal of the program from your system.
8. Select **Start > Control Panel > Administrative Tools > Services** (or **Start > Administrative Tools > Services**, depending on your version of Windows). Select **SQL Server (MSQLSERVER)** then click **Action > Start**.

6.3. Upgrading

Follow the instructions for your system:

- Upgrading a standalone system.
- Upgrading a clustered system.

6.3.1. Upgrading a standalone system

Enhancements will be made to nShield Web Services SQLEKM over time, and product upgrades made available to customers.

Upgrading is a service affecting operation.

Before upgrading, please confirm the following:

- The system and the database are in a working state.
- The version of the nShield Web Services SQLEKM provider:

```
SELECT * FROM sys.dm_cryptographic_provider_properties;
```

To upgrade the product:

1. Disable the nShield Web Services SQL provider using the following query:

```
ALTER CRYPTOGRAPHIC PROVIDER SQLEKM
DISABLE;
GO
```

2. Restart SQL Server.
3. Uninstall nShield Web Services SQLEKM:
    a. Ensure that you are signed in as **Administrator** or as a user with local administrator rights.
    b. Select **Start > Control Panel**.
    c. Depending on your version of Windows, select **Add/Remove programs** or **Uninstall program**.
    d. Select **nShield Web Services SQLEKM**, then click **Uninstall**.
    e. A dialog is displayed, asking if you want to continue with uninstalling nShield Web Services SQLEKM. Click **Yes**.
    f. A setup status screen is displayed while nShield Web Services SQLEKM is uninstalled. Click **Finish** to complete the removal of the program from your system.
4. Install the upgraded version of nShield Web Services SQLEKM, see Installation.
5. Specify the new nShield Web Services SQLEKM provider with the following query (specifying the path as appropriate):

```
ALTER CRYPTOGRAPHIC PROVIDER SQLEKM
FROM FILE = 'C:\Program Files\nCipher\WebServices\SQLEKM\provider\nShield.WebServices.SQLEKM.dll';
GO
```

6. Enable the new nShield Web Services SQLEKM provider using the following query:

```
ALTER CRYPTOGRAPHIC PROVIDER SQLEKM
ENABLE;
GO
```

7. Restart SQL Server and check that the system and database are in a working state. Confirm that the version of the nShield Web Services SQLEKM provider is as expected with the following query:

```
SELECT * FROM sys.dm_cryptographic_provider_properties;
```

6.3.2. Upgrading a clustered system

Enhancements will be made to the nShield Web Services SQLEKM over time, and product upgrades made available to customers.

Upgrading is a service affecting operation.

Before upgrading, please confirm the following:

- The system and the database are in a working state.
- Each of the nodes can become the active node.
- The version of the nShield Web Services SQLEKM provider:

```
SELECT * FROM sys.dm_cryptographic_provider_properties;
```

The procedure is based on upgrading a passive node. This means that the particular node will have to be taken out of service. The following steps are dependent on being able to pause a passive node in the cluster, using the Failover Cluster Manager via the Windows Server Manager (**Server Manager Dashboard > Tools > Failover Cluster Manager**). The paused node is the one to be upgraded. Windows Server 2016 and later versions are automatically cluster-aware. Therefore the Failover Cluster Manager should be available on all versions of Windows that can run SQL Server Enterprise.

To upgrade the product:

1. Select a passive node (Node U) that will be temporarily out of service while it is upgraded. At least one other node (Node A) must remain active to continue service.
2. Using the Failover Cluster Manager (**Server Manager Dashboard > Tools > Failover Cluster Manager**), open the Nodes tree, and select Node U. Pause Node U by selecting **Pause > Drain roles**.
3. On node U, disable the nShield Web Services SQL provider using the following query:

```
ALTER CRYPTOGRAPHIC PROVIDER SQLEKM
DISABLE;
GO
```

4. On node U, uninstall nShield Web Services SQLEKM:
   a. Ensure that you are signed in as **Administrator** or as a user with local administrator rights.
   b. Select **Start > Control Panel**.
   c. Depending on your version of Windows, select **Add/Remove programs** or **Uninstall program**.
   d. Select **nShield Web Services SQLEKM**, then click **Uninstall**.
   e. A dialog is displayed, asking if you want to continue with uninstalling nShield Web Services SQLEKM. Click **Yes**.
   f. A setup status screen is displayed while nShield Web Services SQLEKM is uninstalled. Click **Finish** to complete the removal of the program from your system.
5. On node U, install the upgraded version of nShield Web Services SQLEKM, see Installation.

6. On Node U, specify the new nShield Web Services SQLEKM provider with the following query (specifying the path as appropriate):

```
ALTER CRYPTOGRAPHIC PROVIDER SQLEKM
FROM FILE = 'C:\Program Files\nCipher\WebServices\SQLEKM\provider\nShield.WebServices.SQLEKM.dll';
GO
```

7. On Node U, enable the new nShield Web Services SQLEKM provider using the following query:

```
ALTER CRYPTOGRAPHIC PROVIDER SQLEKM
ENABLE;
GO
```

8. Using the Failover Cluster Manager (**Server Manager Dashboard > Tools > Failover Cluster Manager**), open the **Nodes** tree, and select Node U. Resume Node U by selecting **Resume > Do not fail roles**.
9. On Node U, check that the system and database are in a working state. Confirm that the version of the nShield Web Services SQLEKM provider is as expected with the following query:

```
SELECT * FROM sys.dm_cryptographic_provider_properties;
```

10. Check that all databases held in common by both Node A and Node U are synchronized. If so, Node U can now be used for active service in the cluster if required.
11. Repeat all steps above, except step 6, for all remaining nodes that you wish to upgrade.

6.4. Migrating nDSOP keys

If migrating keys from the nDSOP SQLEKM provider for use with the nShield Web Services SQLEKM provider, it will be necessary for:

- The Web Services Option Pack Server to be indoctrinated into the same Security World as the existing nDSOP installation.
- The existing `sqlekm` keys to be migrated and then renamed.

Before migrating keys to the nShield Web Services SQLEKM provider, the following should be considered:

- The algorithms that are supported by the nShield Web Services SQLEKM provider, see Supported cryptographic algorithms.
- That whilst symmetric keys without an `IDENTITY_VALUE` can be migrated, they will not

be able to be used to decrypt data encrypted by the nDSOP SQLEKM provider. See Symmetric keys for more information on the importance of specifying an `IDENTITY_VALUE`.

- If the Security World is either an unrestricted or FIPS 140-2 Level 2 Security World, it will be necessary to set the following registry entry: `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\nCipher\SQLEKM\WebServices\ForceLegacyRSAPKCS15Padding` (type `REG_DWORD`) to `1`. Setting this registry entry will require SQL Server to be restarted.
- As the Web Services Option Pack currently only supports softcards for protection domains, any nDSOP keys protected by an operator cardset (OCS) will first need to be transferred to a softcard, see the documentation for your HSM for further information.

To indoctrinate the Web Services Option Pack Server into the existing Security World, it will be necessary to make the `%NFAST_KMDATA%\local` folder available. The Security World can then be loaded using `new-world`. See the documentation for your HSM for further information about loading an existing Security World.

Once the Security World has been loaded and confirmed as usable, `dbmt` should be used to migrate the existing `sqlekm` keys, and protecting softcards, to the Web Services Option Pack database. See nShield Web Services v3.3.1 for further information about using the data base management tool `dbmt`.

After the keys which are supported have been successfully migrated, it is necessary to run the `ws-sqlekm-key-rename` tool to ensure the migrated keys are named appropriately. This tool is available on the nShield Web Services SQLEKM ISO under `linux`. The `ws-sqlekm-key-rename.tar.gz` file should be copied to the machine where `dbmt` was run.

To install `ws-sqlekm-key-rename`:

```
tar -xf ws-sqlekm-key-rename.tar.gz
cd opt/nfast/wsop/ws-sqlekm-key-rename
./install.sh
```

`ws-sqlekm-key-rename` is installed to the `PYTHONPATH` directory. (e.g. `/opt/nfast/python3/bin/ws-sqlekm-key-rename`)

The install script must be called with the correct user permissions.

Using the same `config.yaml` as used when running `dbmt`, the migrated keys can be renamed as follows:

```
ws-sqlekm-key-rename --config config.yaml
```

7. T-SQL shortcuts and tips

The following T-SQL queries provide assistance or alternative methods to perform some of the examples shown in this document.

7.1. Creating a database

To create a database called `TestDatabase`.

```
USE master
GO
CREATE DATABASE TestDatabase;
GO
```

7.2. Creating a table

To create the following example table called `TestTable` within a previously created `TestDatabase`.

```
USE TestDatabase
GO
CREATE TABLE TestTable (FirstName varchar(MAX), LastName varchar(MAX),
NationalIdNumber varbinary(MAX), EncryptedNationalIdNumber varbinary(MAX),
DecryptedNationalIdNumber varbinary(MAX));
GO
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('Jack', 'Shepard', 156587454525658);
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('John', 'Locke', 2365232154589565);
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('Kate', 'Austin', 332652021154256);
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('James', 'Ford', 465885875456985);
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('Ben', 'Linus', 5236566698545856);
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('Desmond', 'Hume', 6202366652125898);
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('Daniel', 'Faraday', 7202225698785652);
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('Sayid', 'Jarrah', 8365587412148741);
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('Richard', 'Alpert', 2365698652321459);
INSERT INTO TestTable (FirstName, LastName, NationalIdNumber) VALUES ('Jacob', 'Smith', 12545254587850);
GO
```

7.3. Viewing a table

To view the previously created `TestTable`:

```
SELECT TOP 10 [FirstName]
,[LastName]
,[NationalIDNumber]
,[EncryptedNationalIdNumber]
,[DecryptedNationalIdNumber]
FROM [TestDatabase].[dbo].[TestTable]
```

To view the previously created `TestTable` with the `NationalIDNumber` in the original decimal form:

```
SELECT TOP 10 [FirstName]
,[LastName]
,CAST(NationalIdNumber AS decimal(16,0)) AS [NationalIDNumber]
,[EncryptedNationalIdNumber]
,[DecryptedNationalIdNumber]
FROM [TestDatabase].[dbo].[TestTable]
```

To view the previously created `TestTable` with the `NationalIDNumber` in the original decimal form, and also show the `NationalIdNumber` in `VarBinary` form:

```
SELECT TOP 10 [FirstName]
,[LastName]
,CAST(NationalIdNumber AS decimal(16,0)) AS [NationalIDNumber]
,(NationalIdNumber) AS VarBinNationalIdNumber
,[EncryptedNationalIdNumber]
,[DecryptedNationalIdNumber]
FROM [TestDatabase].[dbo].[TestTable]
```

7.4. Making a database backup

To make a database backup:

```
USE TestDatabase;
GO
BACKUP DATABASE TestDatabase
TO DISK = '<Drive>:\<Backup_directory>\TestDatabase_SomeState.bak'
WITH NOFORMAT,
INIT,
NAME = 'TestDatabase_SomeState Backup',
SKIP,
NOREWIND,
NOUNLOAD,
STATS = 10
GO
```

Where: `<Drive>:\<Backup_directory>` is the path to the directory to store the backup.

7.5. Adding a credential

The following query will add a credential to the database:

```
CREATE CREDENTIAL <credential name> WITH IDENTITY = '<protection domain id>',
SECRET = '<protection domain passphrase>' FOR CRYPTOGRAPHIC PROVIDER <provider name>;
ALTER LOGIN "<domain>\<login name>" ADD CREDENTIAL <credential name>;
```

Where

- `<credential name>` is the name you wish to provide for the credential.
- `<protection domain id>` matches the id of the protection domain.
- `<protection domain passphrase>` matches the passphrase of the protection domain.
- `<provider name>` is the name of the provider you are using.
- `<domain>` is the relevant login domain.
- `<login name>` is the relevant login name (to the database host).

7.6. Removing a credential

To remove a credential from the database:

```
ALTER LOGIN "<domain>\<login name>" DROP CREDENTIAL <credential name>;
DROP CREDENTIAL <credential name>;
```

7.7. Creating a TDEDEK

To create a TDEDEK using `TestDatabase` and `dbAsymWrappingKey` as an example:

```
USE TestDatabase;
CREATE DATABASE ENCRYPTION KEY WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER ASYMMETRIC KEY dbAsymWrappingKey;
GO
```

7.8. Removing a TDEDEK

To remove a TDEDEK using `TestDatabase` as an example:

```
USE TestDatabase
DROP DATABASE ENCRYPTION KEY;
```

7.9. Switching on TDE

To switch on TDE using `TestDatabase` as an example:

```
ALTER DATABASE TestDatabase SET ENCRYPTION ON;
```

7.10. Switching off TDE

To switch off TDE using `TestDatabase` as an example:

```
ALTER DATABASE TestDatabase SET ENCRYPTION OFF;
```

7.11. Dropping a provider

To drop an existing provider:

```
DROP CRYPTOGRAPHIC PROVIDER <provider name>
```

Where

- `<provider name>` is the name of an existing provider.

7.12. Disabling existing providers

To disable all existing providers:

```
sp_configure 'show advanced options', 1; RECONFIGURE;
GO
sp_configure 'EKM provider enabled', 0; RECONFIGURE;
GO
```

7.13. Checking encryption state

To check the encryption state of your databases:

```
SELECT DB_NAME(e.database_id) AS DatabaseName, e.database_id, e.encryption_state, CASE e.encryption_state
WHEN 0 THEN 'No database encryption key present, no encryption'
WHEN 1 THEN 'Unencrypted'
WHEN 2 THEN 'Encryption in progress'
WHEN 3 THEN 'Encrypted'
WHEN 4 THEN 'Key change in progress'
WHEN 5 THEN 'Decryption in progress'
END AS encryption_state_desc, c.name, e.percent_complete FROM sys.dm_database_encryption_keys AS e
LEFT JOIN master.sys.certificates AS c ON e.encryptor_thumbprint = c.thumbprint
```