RAPPORT DU PROJET
D'ADMINISTRATION DE BASES DE DONNES

Cryptographie sur SQL Server 2012

Réalisé par :

AMARA Hala
TEKHZAZ Siham
TANTAOUI Mohamed
YACHAOUI Ayman

Encadré par :

Dr. EL KOUTBI

Résumé

Microsoft SQL Server devient un système de gestion de base de données de plus en plus mature, de plus en plus riche en terme de fonctionnalités et de plus en plus sécurisé. SQL Server 2012 est un gestionnaire de bases de données relationnelles de niveau entreprise. SQL Server étant un produit Microsoft, pensé et implémenté de façon à être le plus facile à installer et à prendre en main ; néanmoins cette apparente facilité ne devrait pas faire perdre de vue aux administrateurs, développeurs et autre ingénieurs de déploiement l'importance du travail à réaliser durant la gestion du SGBD.

Les données stockées sont certainement la propriété la plus précieuse de l'entreprise. Si une entreprise perd malencontreusement ses données, elle n'a plus qu'à mettre la clé sous le paillasson. Alors comment s'assurer de la solidité des dispositifs de sécurité mis en place pour protéger l'intégrité et la confidentialité des données?

Plusieurs considérations doivent être observées pour s'assurer de l'étanchéité de la sécurité des données et services SQL Server. Nous listons les plus pertinentes :

- SQL Server tourne sur Windows, ainsi sécuriser Windows est une nécessité.
- SQL Server est une application Client-Serveur ainsi la sécurisation du réseau sous-jacent est une nécessité.
- SQL Server a pour finalité d'assurer une disponibilité constante de ses services aux utilisateurs correctement authentifiés, il se doit aussi de lire et d'écrire des fichiers de sauvegarde qui doivent être sécurisés. Il peut également y avoir besoin que la donnée stockée au sein d'SQL Server soit protégée par cryptage.

Notre rapport se veut un guide en la matière de ce dernier point concernant le chiffrement des données. Nous adoptons une démarche simplificatrice pour exposer les procédures à suivre pour chiffrer les données sur SQL Server et en comprendre la technologie sous-jacente.

Ce guide est le fruit de notre travail sur la problématique : 'SQL Server et cryptographie' sujet de notre tâche assignée pour le projet d'Administration de Base de Données dispensé par Mr ELKOUTBI au sein de l'Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes de Rabat. Nous espérons avoir accompli notre dû.

Introduction

En informatique, le cryptage consiste en la transformation de data d'un état dit clair et lisible à un état binaire obscure par le biais d'un algorithme de chiffrage. L'algorithme utilise une clé de cryptage qui détermine le résultat final. La clé de cryptage garantie que le résultat du cryptage ne dépendra pas de l'algorithme uniquement. En fait la clé est l'élément le plus important du cryptage. La qualité de l'algorithme garantie qu'il est très difficile de décrypter les données sans connaître la clé. Mais il n'y a aucun secret derrière la clé et la plupart des algorithmes de cryptage sont open-source. Chose qui garantit que la sécurité du cryptage ne dépend pas de la divulgation de la méthode de cryptage. Le vrai élément secret et la clé. La clé est générée par l'utilisateur et l'algorithme et doit être assez longue pour prévenir une attaque de type brute-force. Les clés sont typiquement comprises entre 128 et 2048 bits. Plus longue est la clé, plus robuste est le cryptage ; mais ceci se fait au détriment des ressources de calcul. Chose qui se ressent d'avantage en matière de système de gestion de base de données ce qui implique une considération supplémentaire de ces paramètres. Sur SQL Server les clés peuvent avoir une taille de 128, 192 ou 256 bits, ce qui offre une protection suffisante pour la plupart des cas d'utilisation.

Il y'a plusieurs types de clés. Les plus simples sont les **clés symétriques**, parce que la même clé est utilisée pour crypter et décrypter les données. Elle offre de bonnes performances aux dépens du risque de faire fuiter la clé pendant son transit de l'endroit du cryptage à celui du décryptage. Sur Internet, les clés symétriques sont pratiquement inutilisables.
Pour résoudre ce problème, les **clés asymétriques** ont été introduites. Elles consistent en une paire de clés dites **publique** et **privée**. La clé publique ne sert qu'à crypter les données alors que la clé privée ne sert qu'à les décrypter. Les clés publiques ne sont pas confidentielles et peuvent être publiées sans risque. Il suffit donc de s'assurer de la confidentialité de la clé privée uniquement. Les clés asymétriques offrent aussi un meilleur chiffrage mais ceci est réalisé au détriment des ressources de calcul car elles sont plus complexes et plus lentes que les clés symétriques ce qui les rend impratiques pour du chiffrage massif. Particulièrement lorsqu'il s'agit de bases de données. Pour trouver un juste milieu entre la sécurité et les performances, il est d'usage d'utiliser des clés symétriques pour crypter les valeurs des colonnes d'une base de données. Un troisième type de clés sont les **certificats**. Un certificat est un type de cryptage asymétrique où la clé publique contient des informations sur le détenteur de la clé privée. Les certificats sont importants pour s'assurer non seulement que les données sont cryptées mais aussi qu'elles sont transmises à la bonne partie.

Nous détaillerons dans ce rapport comment SQL Server'12 utilise ces techniques.

Sommaire

Chiffrage de la session via SSL

Entre la machine client et SQL Server, la requête SQL et l'ensemble de résultat sont transmis sur les réseaux par des paquets qui peuvent être interprétés par l'œil humain en utilisant un sniffer de packet tel que **Wireshark.**

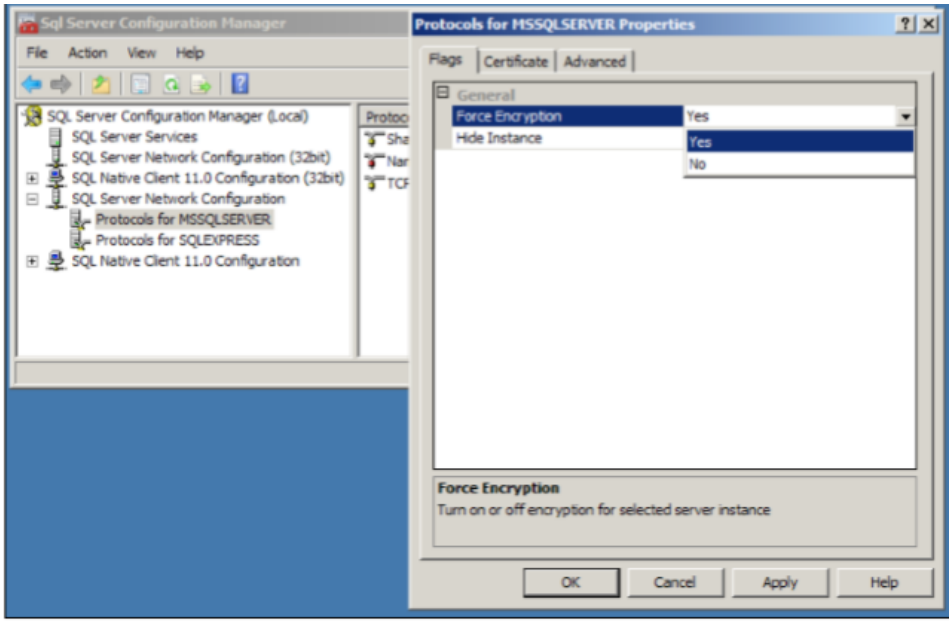
Préparation :

Pour assurer une connexion sécurisée via SSL, il est préférable d'acheter un certificat SSL de chez une autorité de certification (CA) telle que VeriSign, Comodo ou DigiCert. Bien qu'il soit aussi possible d'utiliser un certificat auto-signé, ceci n'est pas recommandé car il ne serait pas validé par une tierce partie de confiance.

Pour que le certificat soit visible par SQL Server, celui-ci doit être installé en utilisant le même compte exécutant SQL Server service. Si SQL Server est exécuté par a un compte système Windows, ou tout autre type de comptes virtuels, alors le certificat doit être installé en utilisant un compte d'utilisateurs bénéficiant de privilèges administrateurs sur le serveur.

Démarche :

1- Nous ouvrons **SQL Server Configuration Manager** et sélectionnons **SQL Server Network Configuration.**
2- Clic droit sur **Protocols for ''Nom de l'instance SQL Server''** puis **Propriétés.**
3- Sur l'onglet **Flags**, nous choisissons **Yes** pour **Force Encryption** pour interdire les connexions non cryptées.



4- Nous ajoutons notre certificat installé dans l'onglet **Certificate**.
5- **Ok** avant de redémarrer le service SQL Server pour que les modifications prennent lieu.

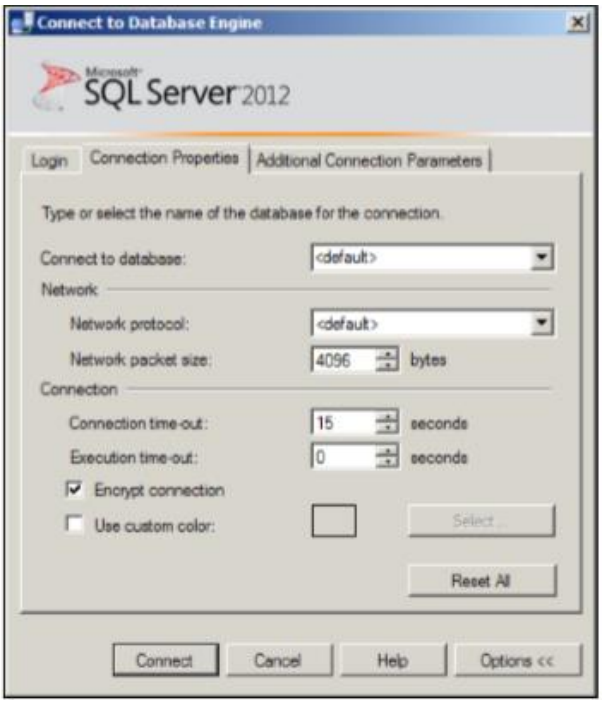
Plus de détails :

Si l'option **Force Encryption** est activée, le client se connectera automatiquement au serveur via SSL. Si non, dans le code client, la chaine de connexion suivante peut être utilisée :

Driver={SQL Server Native Client 11.0};Server=myServerAddress;Database=my DataBase; Trusted_Connection=yes;Encrypt=yes;

Pareil lors de la connexion à **SQL Server Management Studio** :

1- Dans la boite de dialogue **Connect to Database Engine**, nous cliquons sur **Options**.
2- Dans l'onglet **Connection Propreties**, nous cochons **Encrypt connection** :



3- **Connect**
4- Nous pouvons vérifier à l'intérieur d'SQL Server si la connection active est bien chiffrée depuis la vue dynamique de gestion **sys.dm_exec_connections** pour le paramètre **encrypt_option** qui est égale à 1 si la session est bien cryptée, 0 sinon :
SELECT encrypt_option FROM sys.dm_exec_connections
WHERE session_ id = @@SPID;

Utilisation de Transparent Database Encryption (TDE)

Un pirate avec des droits de lectures sur le repertoire où SQL Server stock ses fichiers (.mdf) n'aurait aucune difficulté d'accéder à l'ensemble de la base de données simplement en copiant les .mdf puis les attacher à un autre serveur SQL sur lequel il aurait des droits d'administration. La première protection contre cette menace consiste évidemment à une gestion imperméable des droits NTFS sur le répertoire Data d'SQL Server. Pour encore plus de sécurité, nous pouvons utiliser Transparent Data Encryption qui chiffre tous les fichiers de données de la base. Le pirate n'aurait donc aucun moyen de décrypter les donner même si celui-ci arrive à outrepasser les droits NTFS. La clé de cryptage est stockée dans la base système **master**.

Démarche :

1- Nous créons la clé **server encryption master key**. Sous **SQL Server Management Studio** :
USE master; CREATE MASTER KEY ENCRYPTION
BY PASSWORD = 'Strong_Password';

2- Nous sauvegardons la clé immédiatement dans un emplacement sécurisé:
BACKUP MASTER KEY TO FILE = '\\path\SQL1_master.key' ENCRYPTION BY PASSWORD = 'Very Strong p4ssw0rd'

3- Nous créeons un certificat server:
CREATE CERTIFICATE TDECert WITH SUBJECT = 'TDE Certificate';

4- Que nous sauvegardons dans un fichier backup:
BACKUP CERTIFICATE TDECert TO FILE = '\\path\SQL1_TDECert.cer' WITH PRIVATE KEY (
FILE = '\\path\SQL1_TDECert.pvk',
ENCRYPTION BY PASSWORD = 'Another Very Strong p4ssw0rd'
);

5- Nous sélectionnons la base de données **MyDatabase** que nous souhaitons crypter:
USE MyDatabase;
GO
CREATE DATABASE ENCRYPTION KEY WITH ALGORITHM = AES_128
ENCRYPTION BY SERVER CERTIFICATE TDECert;

6- FInalement, nous activons le chiffrage de la base de données **MyDatabase** :
ALTER DATABASE MyDatabase
SET ENCRYPTION ON;

Plus de détails :

TDE chiffre les données et les fichiers logs sur le disque de manière transparente sans recourir à aucun autre changement au niveau de la base de données (ni le code des applications sur des couches supérieures à SQL Server – d'où l'adjectif transparent) sans impact notable sur les performances de lecture / écriture

L'implémentation est très simple. Au sein de **master**, nous créeons d'abord une clé master et un certificat (ou une clé asymétrique). Puis dans la base de données, nous créeons une clé de chiffrage puis nous activons le cryptage de la base.

La clé de cryptage peut utiliser plusieurs algorithmes : **AES_128, AES_192 , AES_256** et **TRIPLE_DES_3KEY**. Plus la clé est longue, meilleure est la protection mais plus complexe les opérations de cryptage/décryptage deviennent. Cela dit, l'impact sur TDE reste minime.

Si nous souhaitons restaurer une base de données de sauvegarde cryptée sur un autre serveur, nous d'avons d'abord restaurer le certificat utilisé pour crypter les clés de cryptage de la base de données sur le nouveau serveur.

```
USE master;

CREATE CERTIFICATE TDECert FROM FILE = '\\path\SQL1_TDECert.cer'
WITH PRIVATE KEY (
                        FILE  = '\\path\SQL1_TDECert.pvk',
                       DECRYPTION BY PASSWORD = 'Another Very Strong p4ssw0rd'
 );
```

Utilisation des Service et Database Master Keys (SMK/DMK)

SQL Server permet la création et le stockage de clés de cryptage à l'intérieur d'une base de données. Le meilleur moyen de protéger les clés est de les crypter en utilisant une autre clé. Dans SQL Server cette clé est appelée **Master key**. Chaque base de données possède sa propre clé **master (DMK)**, et l'ensemble de ses clés et de nouveau crypté par la clé **master** de niveau serveur **(SMK)**.

Avant qu'une clé puisse être utilisée au sein d'SQL Server, elle doit être ouverte. Cela veut dire qu'elle est lue depuis une table système et décryptée. Après son usage, cette clé est refermée, soit explicitement en utilisant la commande *CLOSE*, soit lorsque la connexion est fermée.

La **SMK** est la racine dans la hiérarchie du chiffrement et est créée automatiquement lorsque SQL Server souhaite pour la première fois crypter une clé. La **DMK** doit être créée manuellement avant d'utiliser le cryptage au sein d'une base de données.

Démarche :

La **SMK** est automatiquement créée. Il est crucial de la sauvegarder :

BACKUP SERVICE MASTER KEY TO FILE = 'e:\encryption_keys\smk.key'
ENCRYPTION BY PASSWORD = 'a strong password';

Pour réstaurer la **SMK**, la procedure est similaire :

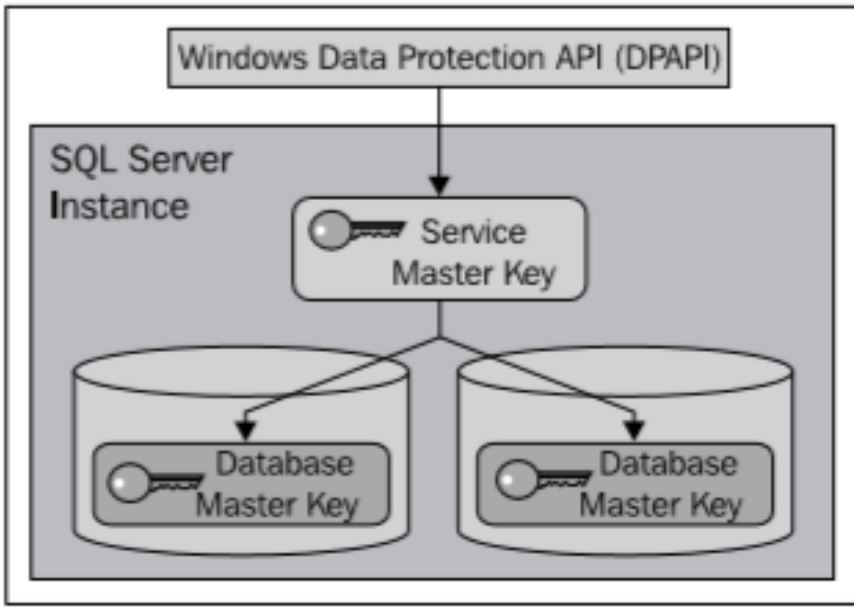
RESTORE SERVICE MASTER KEY FROM FILE = 'e:\encryption_keys\smk.key'
DECRYPTION BY PASSWORD = 'a strong password' ;

Par défaut, la **DMK** est protégée par la **SMK**. La **DMK** doit être manuellement créée :

USE MyDatabase;
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'a very strong password';

Plus de détails :

La **SMK** protégé toutes les autre clés. La **SMK** est protégée par **Windows Data Protection API (DPAPI)**, en utilisant l'algorithme **AES** en utilisant les identifiants compte d'utilisateurs et machine. **SMK** est néanmoins toujours utilisable en cas de changement de compte de service ou bien si la base de donnée **master** est importée depuis un autre serveur SQL utilisant le même compte de service.



La **DMK** doit être crée manuellement à l'intérieur d'une base de données. Deux copies de la **DMK** sont sauvegardées : la première au sein de la base de données et l'autre au sein de **master**, protégée par **SMK**. Ce qui est pratique, parce que cela ôte la nécessité d'ouvrir la clé explicitement au sein du code pour pouvoir l'utiliser ; cela aussi permet à tout utilisateur de la base de données d'utiliser la clé. Dans ce cas, on pourrait vouloir restreindre l'accès seulement aux utilisateurs fournissant un mot de passe. Pour cela il faut supprimer la protection **SMK** :

```
USE MyDatabase;
ALTER MASTER KEY DROP ENCRYPTION BY SERVICE MASTER KEY;
```

Pour utilizer la **DMK**, il n'y a qu'à l'ouvrir puis la refermer après avoir fini:

```
USE MyDatabase;
OPEN MASTER KEY DECRYPTION BY PASSWORD = 'a very strong password';
-- …// Code T-SQL ici
 CLOSE MASTER KEY;
```

Nous pouvons à tout moment voir quelles bases de données possèdent des **DMK** cryptés par **SMK** :

```
SELECT name, is_master_key_encrypted_by_server
FROM sys.databases ORDER BY name;
```

Pour savoir si la base de données actuelle possède une **DMK** :

```
SELECT * FROM sys.symmetric_keys WHERE symmetric_key_id = 101;
```

Création et utilisation de clés de chiffrage symétriques

Pour crypter les données, nous devons d'abord créer les clés. SQL Server nous permet de définir les deux types de clés précédemment introduites :

- Les clés symétriques : La même clé est utilisée pour crypter/décrypter les données
- Les clés asymétriques : Une paire de clé est utilisée – une clé publiquement utilisée exclusivement pour crypter et une clé privée utilisée exclusivement pour décrypter ce que la clé publique à produit.

Les clés symétriques sont plus rapides que les asymétriques, mais moins sécurisées. Nous parlerons dans ce chapitre de comment créer des clés symétriques puis de leur utilisation avant de nous intéresser à leur alternative.

Démarche :

1- Pour créer la clé symétrique *CleSymetriqueNumeroUn* au sein de la base de données *MyDatabase*. Nous pouvons définir des mots de passe différents pour différents utilisateurs. Dans le cas où l'accès pour un des utilisateurs doit être révoqué, seul sa définition de mot de passe doit être supprimée :

```
USE MyDatabase;
CREATE SYMMETRIC KEY CleSymetriqueNumeroUn
WITH ALGORITHM = AES_256
ENCRYPTION BY PASSWORD = 'I am a weak password';
PASSWORD = 'I am a weak password number 2';
```

2- Une fois la clée créée, nous pouvons l'ouvrir.

```
USE MyDatabase;
OPEN SYMMETRIC KEY CleSymetriqueNumeroUn
DECRYPTION BY PASSWORD = 'I am a weak password'; --//ou password numéro 2
-- // Ici nous pouvons crypter de la donnée en utilisant des fonctions que nous présenterons plus bas
CLOSE SYMMETRIC KEY CleSymetriqueNumeroUn;
```

Plus de détails :

Les algorithmes de cryptage qui peuvent être utilisé pour la génération de clés symétriques sont par ordre de robustesse sont **DES**, **TRIPLE_DES**, **TRIPLE_DES_3KEY**, **RC2**, **RC4**, **RC4_128**, **DESX**, **AES_128**, **AES_192**, et **AES_256**.

Une clé est stockée dans la base de données qui l'a créée. Pour ouvrir et utiliser la clé, nous devons nous positionner dans le contexte de la base de donnée. Sinon, nous obtenons un *object not found error*. Il n'y a aucun moyen de sauvegarder ni d'extraire une clé symétrique créée au sein d'SQL Server. Pour pouvoir partager la clé avec un autre server dans le but d'échanger des informations cryptées, nous devons recréer la même clé au sein d'un autre serveur. Pour cela il faut fournir une valeur aux options *KEY_SOURCE* et *IDENTITY_VALUE* au moment de la seconde création de la clé :

```
CREATE SYMMETRIC KEY SKeyToShare
WITH ALGORITHM = AES_256,
KEY_SOURCE = 'a complex passphrase #1',
IDENTITY_VALUE = 'a complex passphrase #2'
ENCRYPTION BY PASSWORD = 'a complex passphrase #3';
```

Deux clés symétriques générées par les *KEY_SOURCE* et *IDENTITY_VALUE* sont équivalentes et interchangeables. Ici nous protégeons la clé par un mot de passe mais nous aurions aussi bien pu le faire avec une autre clé asymétrique ou un certificat.

Création et Utilisation des clés de chiffrage asymétriques

Démarche :

1- Pour créer la clé asymétrique la plus simple qui soit et sans entrer de mot de passe (dans ce cas la clé est protégée par la **DMK**)
CREATE ASYMMETRIC KEY CleAsymetriqueNumeroUn
WITH ALGORITHM = RSA_2048 ;
2- Pour visualiser les clés asymétriques existantes dans la base de données actuelle :
*SELECT * FROM sys.asymmetric_keys;*

Plus de détails :

Dans le chapitre précédent nous avions discuté un code pour ouvrir une clé symétrique. Pourquoi n'utilisons nous pas quelque chose comme : *OPEN ASYMETRIC KEY* ? Parce qu'elle n'existe pas. Microsoft recommande l'utilisation de clé symétrique pour crypter les données principalement parce que le chiffrage asymétrique utilise beaucoup de ressources. L'usage est d'utiliser des clés symétriques pour crypter les données et de protéger les clés symétriques par des clés asymétrique (ou des certificats). C'est pour cela que l'on a choisi **RSA_2048**. Cette clé offre la meilleure protection et les considérations de performances ne sont pas de mise puisque le déchiffrage par la clé asymétrique n'aura lieu qu'une seule fois : à l'ouverture de la clé symétrique.

Nous donnons un exemple de code démontrant l'utilisation de la protection d'une clé symétrique par une clé asymétrique. Ici nous protégeons *CleSymetriqueNumero2* par une clé asymétrique plutôt qu'aver la **DMK**. Ce qui offre plus de control sur les personnes ayant le droit d'ouvrir la clé *CleSymetriqueNumero2.*

```
USE MyDatabase;


CREATE ASYMMETRIC KEY CleAsymetriqueNumero2
WITH ALGORITHM = RSA_2048
ENCRYPTION BY PASSWORD = 'We need a very strong password';

CREATE SYMMETRIC KEY CleSymetriqueNumero2
WITH ALGORITHM = AES_256 ENCRYPTION BY ASYMMETRIC KEY
CleAsymetriqueNumero2;
```

OPEN SYMMETRIC KEY CleSymetriqueNumero2 DECRYPTION BY ASYMMETRIC KEY CleAsymetriqueNumero2 WITH PASSWORD = 'We need a very strong password';

--//Code ici

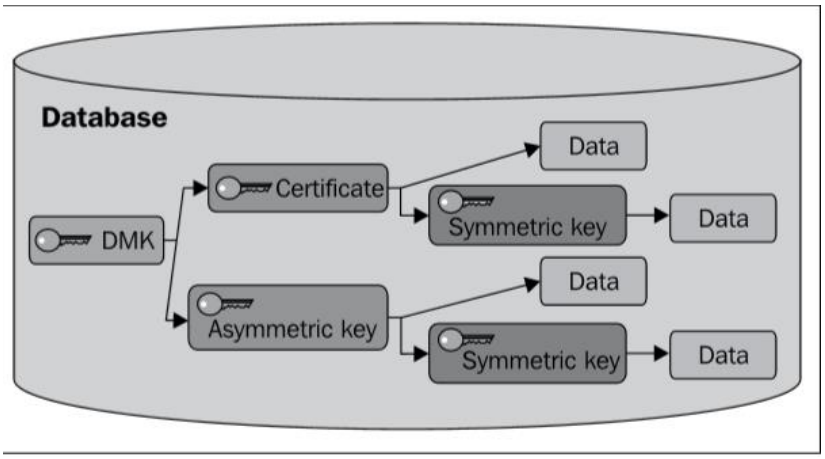
CLOSE SYMMETRIC KEY CleSymetriqueNumero2

Il faut noter que les clés asymétriques ne peuvent pas être exportées ou échangées entre différents serveurs. Si nous souhaitons réaliser un backup ou un export/import des clés vers une autre base de données ou un autre serveur, il faut utiliser un certificat. Objet de notre prochain chapitre.

Création et Utilisation des certificats

Dans le contexte d'SQL Server, les certificats auto-signés peuvent être utilisés au lieu des clés asymétriques. Le diagramme suivant résume les possibilités offertes par SQL Server pour crypter les données :

Démarche :



1- Nous créons un certificat auto-signé protégé par un mot de passe :
CREATE CERTIFICATE MyDatabaseCert
ENCRYPTION BY PASSWORD = 'I am a very strong password'
WITH SUBJECT = 'DataHero from ENSIAS',
EXPIRY_DATE = '20151231';

2- Nous pouvons également importer des certificats existants:
CREATE CERTIFICATE MyDatabaseCert2
FROM FILE = 'e:\encryption_keys\MyDatabaseCert.cer' WITH PRIVATE KEY
(FILE = 'e:\encryption_keys\ MyDatabaseCert.pvk',
DECRYPTION BY PASSWORD = 'the password set at backup');

3- Pour visualizer les certificats exitants dans la base de données actuelle :
*SELECT * FROM sys.certificates ;*

Plus de détails :

Les certificats peuvent être sauvegardés sous forme de fichiers :

BACKUP CERTIFICATE MyDatabaseCert
TO FILE = 'E:\certificates MyDatabaseCert.cer'
WITH PRIVATE KEY (
FILE = 'E:\certificates\ MyDatabaseCert.pvk' ,

ENCRYPTION BY PASSWORD = 'a strong password to encrypt the private key',
DECRYPTION BY PASSWORD = 'the strong password that protects the certificate');

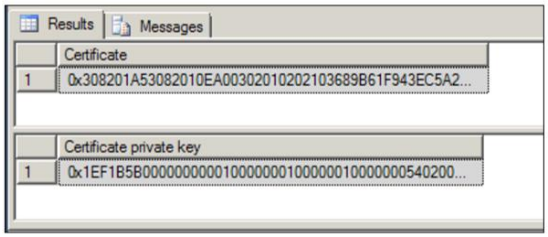
Puis pour l'importer ailleurs, nous pouvons recréer le certificat :

USE MyDatabase_Copy;
CREATE CERTIFICATE MyDatabaseCert
FROM FILE = 'E:\certificates\ MyDatabaseCert.cer'
WITH PRIVATE KEY (FILE = 'E:\certificates\ MyDatabaseCert.pvk',
DECRYPTION BY PASSWORD = 'a strong password to encrypt the private key');

Pour copier un certificate vers une autre base de données sur le même serveur, nous pouvons utiliser les fonctions **CERTENCODED( )** et **CERTPRIVATEKEY( )** pour passer des copies binaires à la commande **CREATE CERTIFICATE** :

USE MyDatabase;
SELECT CERTENCODED(CERT_ID('MyDatabaseCert')) as [Certificate];
SELECT CERTPRIVATEKEY(CERT_ID('MyDatabaseCert'),
'a password to encrypt the result',
'the password to decrypt the private key')
as [Certificate private key];

Le résultat de ces deux fonctions est montré dans la prise d'écran suivante :



Puis nous copions-collons les valeurs en paramètre pour la fonction **CREATE CERTIFICATE** dans une autre base de données :

USE MyDatabase2;
CREATE CERTIFICATE MyDatabase2Cert
FROM BINARY = <la valeur binaire de CERTENCODED>
WITH PRIVATE KEY (
BINARY = < a valeur binaire de CERTPRIVATEKEY>,
DECRYPTION BY PASSWORD 'the password to decrypt the private key');

Chiffrement des données via clés symétriques

Le but premier des clés est le cryptage des données telles que les valeurs des colonnes. Le moyen recommandé de le faire sur SQL Server est en utilisant les clés symétriques vu que le (dé)cryptage via clés asymétriques est significativement plus lent, au point où l'impact sur les performances est visible même pour des tables de petites tailles.

Démarche :

1- Pour crypter les données via une clé symétrique. Nous utilisons la fonction **EncryptByKey( )** :

```
USE MyDatabase ;

CREATE TABLE dbo.Customer (
CustomerId int NOT NULL IDENTITY(1,1) PRIMARY KEY,
Firstname varchar(50) NOT NULL,
Lastname varchar(50) NOT NULL,
CreditCardInfo varbinary(2000) NOT NULL
)

 CREATE CERTIFICATE KeyProtectionCert
WITH SUBJECT = 'to protect symmetric encryption keys';

CREATE SYMMETRIC KEY CreditCardSKey
WITH ALGORITHM = AES_256,
KEY_SOURCE = '4frT-7FGHFDfTh98#6erZ3dq#«',
IDENTITY_VALUE = 'l·Fg{(ZEfd@23fz4fqeRHY&4efVql'
ENCRYPTION BY CERTIFICATE KeyProtectionCert;

OPEN SYMMETRIC KEY CreditCardSKey
DECRYPTION BY CERTIFICATE KeyProtectionCert;

INSERT INTO dbo.Customer (Firstname, LastName, CreditCardInfo)
VALUES ('Mohamed', 'Tantaoui', EncryptByKey(Key_Guid('CreditCardSKey'),
'1111222233334444;12/13,456', 1, 'MohamedTantaouiSALT') );

 CLOSE SYMMETRIC KEY CreditCardSKey;
```

2- Pour lire les données et retrouver sa forme originale non cryptée, nous utilisons la fonction **DecryptByKey( )** :
OPEN SYMMETRIC KEY CreditCardSKey DECRYPTION BY CERTIFICATE KeyProtectionCert;

SELECT Firstname, Lastname,
CAST(DecryptByKey(CreditCardInfo, 1, Firstname + Lastname) as varchar(50))
FROM dbo.Customer;

CLOSE SYMMETRIC KEY CreditCardSKey;

Plus de détails :

Ici l'objectif est de stocker de la donnée cryptée au sein d'une colonne. Le résultat de **EncryptByKey ( )** est une valeur binaire d'une taille maximale de 8000 octets. Nous aurons donc à créer des colonnes **VARBINARY** pour stocker la data cryptée. Le cryptage contrairement au hashage permet d'inverser la valeur et retrouver le texte original. Quelqu'un ayant accès au binaire crypté sans pouvoir le décrypter pour autant peut toujours l'utiliser à des fins malicieuses. Supposons que moi, Ayman Yachaoui possède l'accès à la table **Customer**, je veux placer un ordre d'achat sans pour autant avoir à payer pour. Je pourrais exécuter une requête comme suit :

INSERT INTO dbo.Customer (Firstname, LastName, CreditCardInfo)
SELECT 'Ayman', 'Yachaoui', CreditCardInfo
WHERE Firstname = 'Mohamed' AND Lastname = 'Tantaoui';

Avec ce code, j'insère mon nom dans la table **Customer** mais avec les informations bancaires de Mohamed Tantaoui. Ainsi, en effectuant un achat, Mohamed paiera pour moi.

Ainsi, pour éviter ce risque, nous devons toujours utiliser un paramètre authentificateur au sein de **EncryptByKey( )**. Un **Authentificateur** est une valeur, un **salt** qui sera ajouté à la clé pour le cryptage et qui est spécifique à chaque ligne.

En cryptographie, un **salt** est une valeur aléatoire ajoutée à la clé pour augmenter la complexité du texte crypté. Dans notre exemple c'est *MohamedTantaouiSALT.* Le parameter 1 précédent indique qu'un authentificateur doit être utilisé. 0 autrement.

Chiffrement des données via clés symétriques ou des certificats

Nous avons précédemment indiqué que pour des considérations de performances, il est préférable de crypter les données via des clés symétriques et crypter celles-ci à leur tour par des clés asymétriques ou un certificat. Il est néanmoins possible de passer outre cette recommandation et crypter les données directement par une paire de clé privée/publique

Démarche :

Pour réaliser le cryptage asymétrique, nous avons tous simplement à créer une clé asymétrique ou un certificat et utiliser les fonctions **EncryptByAsymKey( )** ou **EncryptByCert( )**. Nous devons néanmoins nous assurer que la **DMK** existe pour la base de données courante.

```
CREATE ASYMMETRIC KEY DataEncryptionAsymKey
WITH ALGORITHM = RSA_2048;


DECLARE @plaintext NVARCHAR(1000) = 'I have nothing interesting to say, but I don''t want anybody to know that';
DECLARE @ciphertext VARBINARY(8000) ;

SELECT @ciphertext = ENCRYPTBYASYMKEY(ASYMKEY_
ID('DataEncryptionAsymKey'), @plaintext) ;

SELECT @ciphertext;

SELECT CAST(DECRYPTBYASYMKEY(ASYMKEY_ID('DataEncryptionAsymKey'),@
ciphertext) AS NVARCHAR(1000)) as plaintext_again;
```

Plus de détails :

Contrairement au clés symétriques, les clés asymétriques et les certificats ne font pas appel à la fonction **OPEN** au préalable de leur utilisation. Nous n’avons que les appeler avec les arguments qu’ils nécessitent, premièrement la **KeyID** que nous pouvons retrouver en utilisant les fonctions **AsymKey_ID( )** ou **Cert_Id( )**.

Création et stockage de valeurs de hashage

Ce dont nous avons discuté dans les chapitres précédent appartient à ce qu'on appelle le type de cryptage réversible. Parfois nous n'avons pas besoin de cette fonctionnalité. Nous nous suffisons de comparer les valeurs cryptées entre elles. C'est le cas notamment des login/mot de passes, ou bien utiliser les hashage pour générer des authentificateurs uniques pour le chiffrage symétrique. Le cryptage irréversible s'appelle le **hashage**.

Démarche :

1- Discutons l'exemple de création d'une table de logins. La taille du hash dépend de l'algorithme, nous allons utiliser **SHA2-512** qui retourne 64 octets que nous allons stocker dans une colonne de type **BINARY(64)**.
CREATE TABLE dbo.LoginPassword (
Login NVARCHAR(50) COLLATE Latin1_General_BIN2 NOT NULL PRIMARY KEY,
Password BINARY(64) NOT NULL)

2- Ensuite, nous insérons les details de **Login** et **Password** et générons un hash en utilisant la fonction **HASHBYTES( )**. Le premier paramètre à passer est le nom de l'algorithme :
INSERT INTO dbo.LoginPassword
VALUES ('Fred', HASHBYTES('SHA2_512', N'123456'));

3- Nous donnons un exemple de test pratique pour vérifier si un utilisateur est bien connu et le mot de passe fourni est bien correct.
IF EXISTS (
SELECT * FROM dbo.LoginPassword
WHERE Login = 'user' AND Password = HASHBYTES('SHA2_512', N'123456')
) PRINT 'ok';

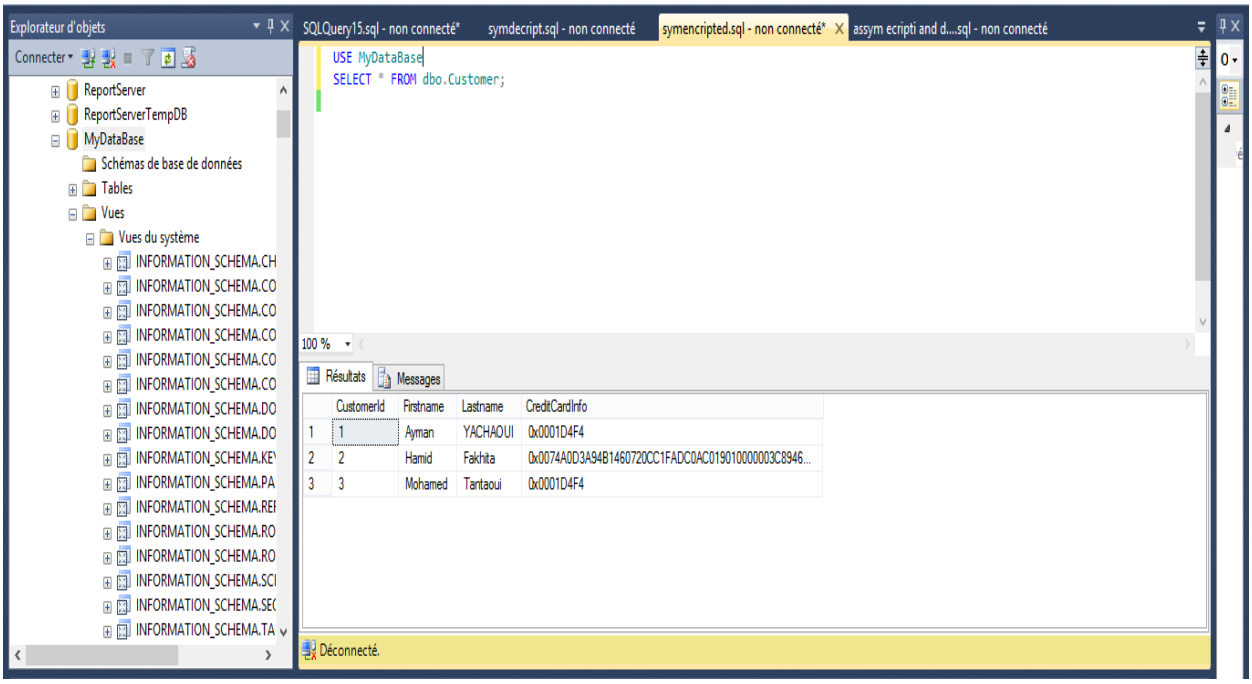
Plus de détails :

Dans cet exemple, nous utilisons la colonne **login** avec une forte collation binaire. Ceci force la sensibilité à la casse.

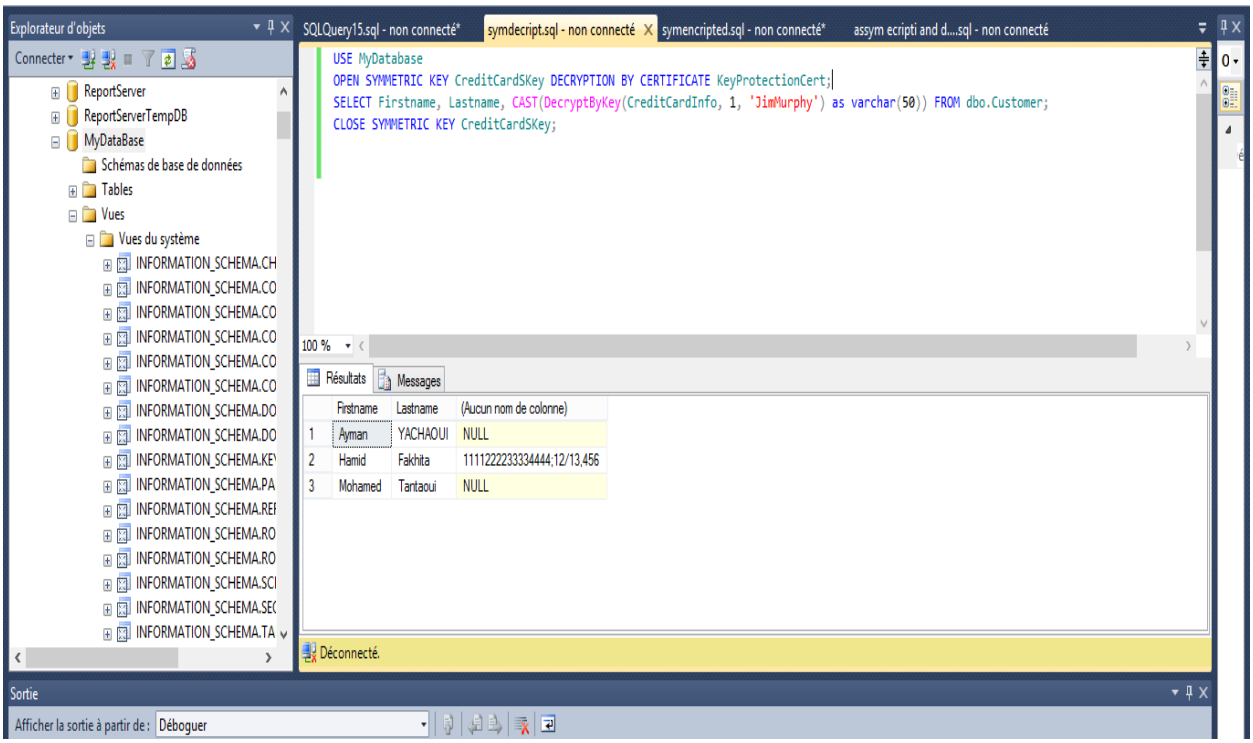
Evidemment, cet exemple n'est pas parfait, nous devrions en théorie crypter la session SSL pour prévenir le sniffage de packets comme discuté lors du premier chapitre.

Sorties d'écrans sélectionnées :

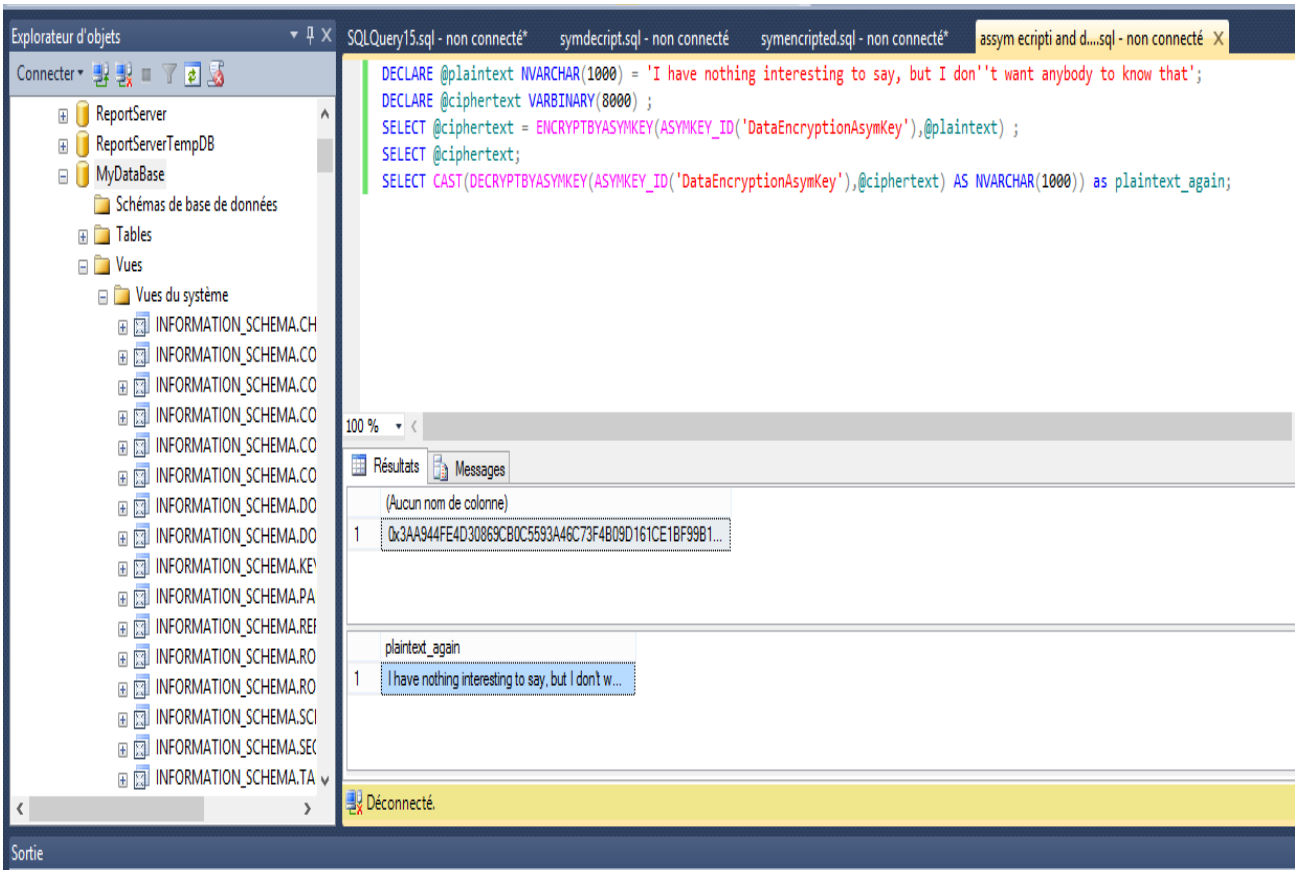
Cryptage symétrique :



Décryptage symétrique :



Cryptage asymétrique / certificats :



Création des valeurs de hashage :