Until this point, I’ve discussed the wide assortment of cryptographic functionality built into SQL Server. The tools I’ve talked about include symmetric and asymmetric encryption functions, encryption key management, hashing, EKM, and TDE. These tools provide an impressive level of cryptographic functionality that can help make SQL Server databases more secure than ever.

Even as I’ve discussed the power of these built-in SQL Server encryption functions, however, I’ve also discussed their limitations. The symmetric encryption and hash functions can encrypt only 8,000 bytes of data, for instance. In this chapter, I’ll talk about how you can use the SQL Server Common Language Runtime (SQL CLR) to overcome some of these limitations by accessing functions in the .NET Framework’s System.Security.Cryptography namespace from T-SQL.

**Encrypting By Passphrase**

As I explained in Chapter 3, SQL Server’s built-in EncryptByPassPhrase function accepts a passphrase and a block of data. This function uses the passphrase to generate an encryption key and then uses the key to encrypt the data with the Triple DES algorithm. There are two limitations with this method of encryption—you can’t specify a different algorithm and you can’t encrypt more than 8,000 bytes at one time.

Fortunately, SQL CLR allows you to work around these limitations, as I’ll demonstrate in the code samples in this section. These C# code samples implement two new functions, EncryptAesByPassPhrase and DecryptAesByPassPhrase. These two functions mirror the functionality of SQL Server’s built-in EncryptByPassPhrase and DecryptByPassPhrase functions. The signatures of the new SQL CLR functions are shown in the following:

**EncryptAesByPassPhrase** (PassPhrase, Plaintext, AddAuthenticator, Authenticator)
**DecryptAesByPassPhrase** (PassPhrase, Ciphertext, AddAuthenticator, Authenticator)

The functions share three parameters in common: *PassPhrase* is the password or passphrase used to internally generate an AES encryption key. *AddAuthenticator* is a bit value, which should be 1, if you want to use an authenticator to encrypt the text, or it can be 0 or NULL if you don’t want to use an authenticator. The *Authenticator* parameter is an nvarchar string that can be used to further obfuscate your ciphertext (assuming you set AddAuthenticator to 1). The SHA-1 hash value of the *Authenticator* value is appended to your *PassPhrase* prior to encryption key generation.
EncryptAesByPassPhrase Function

The EncryptAesByPassPhrase function accepts a varbinary(max) value for its Plaintext parameter. This plaintext is encrypted with the AES algorithm with a 256-bit encryption key. The result is an encrypted varbinary(max). Listing 8-1 is the C# source code listing for the EncryptByPassPhrase function.

Note The result of EncryptAesByPassPhrase has a 16-byte random initialization vector (IV)/salt value prepended to the encrypted ciphertext. The ciphertext is encrypted in CBC mode, which I described in Chapter 3.

Listing 8-1. EncryptByPassPhrase Source Code

```csharp
    IsDeterministic = false,
    DataAccess = DataAccessKind.None)]
[return: SqlFacet(MaxSize = -1)]
public static SqlBytes EncryptAesByPassPhrase(
    SqlString PassPhrase,
    [SqlFacet(MaxSize = -1)] SqlBytes Plaintext,
    SqlBoolean AddAuthenticator,
    SqlString Authenticator)
{
    try
    {
        // Automatically return NULL if passphrase or plaintext is NULL
        if (PassPhrase.IsNull || Plaintext.IsNull)
            return SqlBytes.Null;

        // Generate hash for authenticator
        SHA1Managed Sha1 = new SHA1Managed();
        string AuthHash = ""; // If authenticator not used, use empty string
        // Convert the authenticator hash to Base64 to avoid conversion problems
        if (AddAuthenticator.IsTrue && !Authenticator.IsNull)
            AuthHash = Convert.ToBase64String(Sha1.ComputeHash(Encoding.Unicode.GetBytes(Authenticator.Value)));

        // Append authenticator to passphrase
        string AuthPass = PassPhrase.Value + AuthHash;
    }
    //...