10

Cryptography in .NET 4.0

In Depth

Consider a situation where there is a peer-to-peer .NET 4.0 application used on the Intranet. Users of this application can communicate with one another by exchanging data through files. Now, one user named John intends to send an important file to another user named Harry on the same network. Although, John can send the file in its original form (in plain text), there is a risk of other users on the network who may intercept during the data exchange. Such an interception is not only unwarranted but also potentially harmful for the data. Therefore, to secure the data that is to be exchanged, it is advisable to use cryptography.

Cryptography, in literal sense, means hidden writing (or secret writing). It derives its name from the Greek words *krypto*, which means hidden, and *grafo*, which means to write. Cryptography is considered a branch of computer science that encapsulates mathematical algorithms and functions to allow you to secure data by modifying its original form such that it is incomprehensible to all users except the intended recipient. Cryptography is highly useful when you want to exchange data over a non-secure network and it serves three main purposes:

- **Confidentiality**—Implies that the data cannot be viewed or understood by any user other than the intended recipient. This is achieved by encrypting the data, i.e. modifying the original form of data such that only the intended recipient can read and understand the data by decrypting the encrypted data (also called ciphertext). In .NET Framework, you can maintain the confidentiality of the data by using secret-key encryption or public key-encryption.
- **DataIntegrity**—Implies that the data that is received by the intended recipient is not modified or tampered by other users on the network. In .NET Framework, this is achieved by using hash algorithms. A hash algorithm takes a piece of data and generates a fixed-length sequence of numeric/alphanumeric characters known as hash value (or simply hash). The hash value for one piece of data is uniquely distinct from hash values of other pieces of data. Therefore, if there is any change in the hash value of the data that reaches the intended recipient, then it can be concluded that the data may have been tampered with during the transmission.
- **Authenticity**—Implies that the received data is transmitted by the sender only and not by any other user on the network. In .NET Framework, you can use digital signatures and hash algorithms to verify the identity of the sender. A digital signature is a block of code, which ensures that any data that is received originates from a specific user. It is used to simulate the security properties of a signature in digital form. Two algorithms are involved here, one for signing, in which the secret key of the user is used and other for verifying signatures, in which the public key of the user is used. The benefit of using digital signatures is that they provide authentication and integrity to the messages sent. Authentication of data ensures that it is received from a trusted source and integrity of data ensures that the received data has not been tampered with during transmission. In case of any tampering with the data, you can also reject the data.

In the following sections,you learn about the secret-key encryption, public-key encryption, and hash algorithms. However, prior to encryption and hash algorithms, let's learn about the System.Security.Cryptography namespace.

The System.Security.Cryptography Namespace

.NET Framework4.0 includes the `System.Security.Cryptography` namespace that contains numerous classes for various cryptographic services, such as encoding and decoding of data, encryption and decryption of data, generation of hashes, and authentication of sender. Table 10.1 lists some of the noteworthy classes contained in the `System.Security.Cryptography` namespace:

Table 10.1: Noteworthy Classes in the System.Security.Cryptography Namespace

Class	Description
`Aes`	Represents the abstract base class from which every implementation of the Advanced Encryption Standard (AES) inherits.

Table 10.1: Noteworthy Classes in the System.Security.Cryptography Namespace	
Class	**Description**
`AesCryptoServiceProvider`	Provides the facility to perform symmetric encryption and decryption by using the Cryptographic Application Programming Interface (CAPI) of the AES algorithm.
`AesManaged`	Provides a managed implementation of the AES symmetric algorithm.
`AsnEncodedData`	Represents Abstract Syntax Notation One (ASN.1)-encoded data.
`AsnEncodedDataCollection`	Represents a collection of objects of the `AsnEncodedData` class.
`AsnEncodedDataEnumerator`	Provides the ability to navigate through a collection of AsnEncodedData objects.
`AsymmetricAlgorithm`	Represents the abstract class that is the base class from which implementations of all asymmetric algorithms inherit.
`CngAlgorithm`	Contains the name of an encryption algorithm.
`CngAlgorithmGroup`	Contains the name of an encryption algorithmgroup.
`CngKey`	Provides basic functionality for working with keys of Cryptography Next Generation (CNG) objects.
`CngKeyBlobFormat`	Contains the name of a key BLOB format for CNG objects.
`CngKeyCreationParameters`	Contains various advanced properties for creating a key for CNG objects.
`CngPropertyCollection`	Provides a strongly typed collection of CNG properties.
`CngProvider`	Contains the name of a key storage provider (KSP) for the CNG objects.
`CngUIPolicy`	Contains optional parameters for configuring the UI that appears when a protected key is accessed.
`CryptoAPITransform`	Allows you to perform a cryptographic transformation of data.
`CryptoConfig`	Allows you to access information about the cryptography configuration.
`CryptographicAttributeObject`	Contains a type and the collection of values associated with that type.
`CryptographicAttributeObjectCollection`	Represents a collection of `CryptographicAttributeObject` objects
`CryptographicAttributeObjectEnumerator`	Provides the facility to enumerate through a `CryptographicAttributeObjectCollection` object.
`CryptographicException`	Represents an exception that occurs during a cryptographic operation.
`CryptographicUnexpectedOperationException`	Represents an exception that occurs when an unexpected operation occurs during a cryptographic operation.
`CryptoStream`	Provides a stream that connects data streams to cryptographic transformations.
`CspKeyContainerInfo`	Provides additional information (for example, the key number) about a cryptographic key pair.
`CspParameters`	Provides certain parameters that you can pass to the cryptographic service provider (CSP). The CSP performs various cryptographic computations.
`DeriveBytes`	Represents the abstract base class for all classes that have byte sequences of a given length.
`DES`	Represents the base class for all the implementations of the Data Encryption Standard (DES) algorithm.
`DESCryptoServiceProvider`	Provides a wrapper for accessing the CSP implementation of the DES algorithm.

Table 10.1: Noteworthy Classes in the System.Security.Cryptography Namespace	
Class	**Description**
DSA	Represents the abstract base class for all the implementations of the Digital Signature Algorithm (DSA).
DSACryptoServiceProvider	Provides a wrapper for accessing the cryptographic service provider (CSP) implementation of the DSA algorithm.
DSASignatureDeformatter	Provides the facility to verify a Digital Signature Algorithm (DSA) PKCS#1 v1.5 signature.
DSASignatureFormatter	Provides the facility to create a Digital Signature Algorithm (DSA) signature.
FromBase64Transform	Allows you to convert base-64 encoded data to a CryptoStream object.
HashAlgorithm	Represents the base class for all the implementations of cryptographic hash algorithms.
HMAC	Represents the abstract base class for all the implementations of Hash-based Message Authentication Code (HMAC).
HMACMD5	Represents a hash algorithm that is created using the Message Digest Algorithm 5 (MD5) hash function and is used as HMAC.
HMACRIPEMD160	Represents a hash algorithm that is created using the RACE Integrity Primitives Evaluation Message Digest 160 (RIPEMD160 hash function and is used as HMAC.
HMACSHA1	Represents a hash algorithm that is created using the Secure Hash Algorithm 1 (SHA1) hash function and is used as HMAC.
HMACSHA256	Represents a hash algorithm that is created using the Secure Hash Algorithm 256 (SHA256) hash function and is used as HMAC.
HMACSHA384	Represents a hash algorithm that is created using the Secure Hash Algorithm 384 (SHA384) hash function and is used as HMAC.
HMACSHA512	Represents a hash algorithm that is created using the Secure Hash Algorithm 512 (SHA512) hash function and is used as HMAC.
KeyedHashAlgorithm	Represents the abstract base class for all the implementations of keyed hash algorithms.
KeySizes	Provides a set of key sizes for symmetric cryptographic algorithms.
MACTripleDES	Provides the facility to compute a Message Authentication Code (MAC) using the TripleDES hash algorithm for the input CryptoStream object.
ManifestSignatureInformation	Provides information for the manifest signature.
ManifestSignatureInformationCollection	Represents a read-only collection of objects of the ManifestSignatureInformation class.
MD5	Represents the abstract base class for all the implementations of the MD5 hash algorithm.
MD5Cng	Represents a CNG implementation of the MD5 128-bit hash algorithm.
MD5CryptoServiceProvider	Allows you to compute the MD5 hash value for the input data using the CSP implementation.
Oid	Represents an identifier for a cryptographic object.
OidCollection	Represents a collection of `Oid` objects.
OidEnumerator	Allows you to enumerate through an `OidCollection` object.
PasswordDeriveBytes	Provides the facility to use an extension of the PBKDF1 algorithm to get a key

Table 10.1: Noteworthy Classes in the System.Security.Cryptography Namespace	
Class	**Description**
	from a password.
`ProtectedData`	Provides the facility to make the data protected and unprotected.
`ProtectedMemory`	Provides the facility to make the memory protected and unprotected.
`RandomNumberGenerator`	Represents the abstract base class for all the implementations of generators of cryptographic random numbers.
`RC2`	Represents the base class for all the implementations of the RC2 algorithm.
`RC2CryptoServiceProvider`	Provides a wrapper for accessing the CSP implementation of the RC2 algorithm.
`Rfc2898DeriveBytes`	Provides the PBKDF2 functionality for password-based key derivation. This class uses a pseudo-random number generator that is based on HMACSHA1.
`Rijndael`	Represents the base class for all the implementations of the Rijndael symmetric encryption algorithm.
`RijndaelManaged`	Allows access to the managed implementation of the Rijndael algorithm.
`RijndaelManagedTransform`	Provides the facility to perform a cryptographic transformation of data by using the Rijndael algorithm.
`RIPEMD160`	Represents the abstract base class for all the implementations of the MD160 hash algorithm.
`RIPEMD160Managed`	Provides the facility to compute a RIPEMD160 hash of an input data using the managed library.
`RNGCryptoServiceProvider`	Implements a cryptographic Random Number Generator (RNG) by using the CSP implementation.
`RSA`	Represents the base class for all the implementations of the Random Sequential Adsorption (RSA) algorithm.
`RSACryptoServiceProvider`	Provides the facility to perform asymmetric encryption and decryption using the CSP implementation of the RSA algorithm.
`SHA1`	Provides the facility to compute the SHA1 hash for an input data.
`SHA1Cng`	Provides the CNG implementation of the SHA1 hash algorithm.
`SHA1CryptoServiceProvider`	Provides the facility to compute the SHA1 hash value for an input data by using the CSP implementation.
`SHA1Managed`	Provides the facility to compute the SHA1 hash for an input data using the managed library.
`SHA256`	Provides the facility to compute the SHA256 hash for an input data.
`SHA256Cng`	Provides the CNG implementation of the SHA256 hash algorithm.
`SHA256CryptoServiceProvider`	Provides a wrapper for accessing the CSP implementation of the SHA256 hash algorithm.
`SHA256Managed`	Provides the facility to compute the SHA256 hash for an input data using the managed library.
`SHA384`	Provides the facility tocompute the SHA384 hash for an input data.
`SHA384Cng`	Provides the CNG implementation of the SHA384 hash algorithm.
`SHA384CryptoServiceProvider`	Provides a wrapper for accessing the CSP implementation of the SHA384 hash algorithm.
`SHA384Managed`	Provides the facility to compute the SHA384 hash for an input data using the managed library.

Table 10.1: Noteworthy Classes in the System.Security.Cryptography Namespace	
Class	**Description**
SHA512	Provides the facility tocompute the SHA512 hash for an input data.
SHA512Cng	Provides the CNG implementation of the SHA512 hash algorithm.
SHA512CryptoServiceProvider	Provides a wrapper for accessing the CSP implementation of the SHA512 hash algorithm.
SHA512Managed	Provides the facility to compute the SHA512 hash for an input data using the managed library.
SignatureDescription	Contains information about the various properties of a given digital signature.
StrongNameSignatureInformation	Contains information about the strong name signature of a manifest.
SymmetricAlgorithm	Represents the abstract base class for all the implementations of symmetric algorithms.
ToBase64Transform	Provides the facility to convert a CryptoStream object to base-64 encoded data.
TripleDES	Represents the base class for all the implementations of the Triple Data Encryption Standard algorithms.
TripleDESCryptoServiceProvider	Provides a wrapper for accessing the CSP implementation of the TripleDES algorithm.

NOTE

The Cryptography Service Provider (CSP) is a library that contains the Cryptographic Application Programming Interface (CAPI), which provides the cryptographic algorithms and functions.

Now, let's learn about the encryption and decryption processes in detail.

Encryption and Decryption

As stated earlier, encryption refers to the process of modifying a piece of data in plain text into another piece of data, known as encrypted data or ciphertext, such that no one other than the intended recipient can decipher the original data. When the recipient regenerates the data (in plain text) from the encrypted data, it is referred to as decryption. Encryption is done at the sender's end and decryption is done at the receiver's end. Note that it is essential that a lossless channel/network is used for successfully regenerating data after decryption. Figure 10.1 depicts a pictorial representation of encryption and decryption:

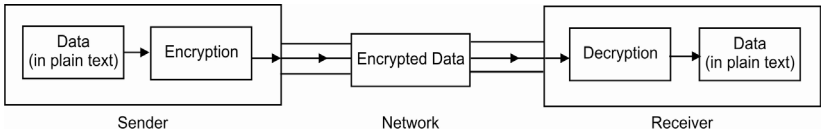

Figure 10.1: Displaying the Process of Encryption and Decryption

For encrypting and decrypting data, you need two things, a cryptography algorithm and a key. A cryptography algorithm is a sequence of instructions that encrypts and decrypts the data. Cryptographic algorithms use a particular value, known as key, to encrypt or decrypt data. The sender uses a given key to encrypt a piece of data, which is then transmitted to the intended recipient. When the intended recipient receives the encrypted data, the recipient decrypts the data using a given key. The key to encrypt and decrypt the data is decided prior to encrypting the data. This implies that if a user (including the recipient) does not have the predetermined key, then the user cannot decrypt the data. Therefore, it is essential for the intended recipient to have the right key; otherwise, it is almost impossible to decrypt the received encrypted data.

Depending on the key that is used for encryption and decryption, there are two types of encryption—secret-key encryption and public-key encryption. Now, let's first learn about the secret-key encryption.

Secret-Key Encryption

In secret-key encryption, a single key, also called the secret key, is used for both encryption and decryption of data. This secret key is shared between the sender and the intended recipient but is hidden from the remaining users on the network. The sender uses the secret key to encrypt the data and sends the encrypted data to the intended recipient over the network. When the intended recipient receives the encrypted data, the recipient uses the same secret key to decrypt the data. The recipient needs to use the same secret key that was used to encrypt the data; any other key will not serve the purpose. Note that the sender and the intended recipient need to know the secret key that is to be used for encryption and decryption. The sender and the intended recipient can share the secret key by generating the key at the sender's end, encrypting it using public-key encryption, and sending it to the intended recipient. Note that the length of the secret key determines the strength of the encryption, that is, encryption that uses a longer key is more difficult to break than the one that uses a smaller key.

Secret-key encryption is also known as symmetric encryption as encryption and decryption uses the same key. The cryptography algorithms used for secret-key (or symmetric) encryptions are known as symmetric algorithms. These algorithms are quite fast and,therefore, are suitable for encrypting and decrypting large amounts of data. Most of the symmetric algorithms are block ciphers as these algorithms encrypt a single block of data at a time. Block ciphers take a block of fixed number of bytes of the data and generate an output block of encrypted data. The fixed number of bytes that block ciphers take may differ from one block cipher to another. For example, one block cipher may take an input block of 8 bytes and another may take an input block of 16 bytes. If the entire data is larger than this fixed number of bytes, then a block cipher encrypts one block at a time. However, if the data is smaller than the fixed number of bytes, then the data is made larger to fit the fixed number of bytes.

The block ciphers provided by .NET Framework4.0 use an initialization vector (IV) along with the secret key to encrypt and decrypt data. An initialization vector initializes the first block of input data that is to be encrypted. For the remaining blocks of input data, a bitwise exclusive-OR is performed between the current block of input data and the previous block of encrypted data.This implies that except for the first block of encrypted data, a block of the encrypted data depends on its previous block of encrypted data. This process is known as cipher block chaining (CBC).

In .NET Framework4.0, there are various symmetric cryptography algorithms to implement secret-key encryption.All the symmetric algorithms inherit the SymmetricAlgorithm class defined in the System.Security.Cryptography namespace. The SymmetricAlgorithm class is an abstract base class that represents a symmetric algorithm. There are various properties and methods that the SymmetricAlgorithm class offers.

The syntax of the SymmetricAlgorithm class is as follows:

```
[ComVisibleAttribute(true)]
public abstract class SymmetricAlgorithm : IDisposable
```

Table 10.2 lists some of the noteworthy properties of the SymmetricAlgorithm class:

Table 10.2: Noteworthy Properties of the SymmetricAlgorithm Class

Property	Description
`BlockSize`	Sets or retrieves the block size (in bits) used for the cryptographic operation
`FeedbackSize`	Sets or retrieves the feedback size (in bits) used for the cryptographic operation
`IV`	Sets or retrieves the Initialization Vector (IV) used in the symmetric algorithm
`Key`	Sets or retrieves the secret key for the symmetric algorithm
`KeySize`	Sets or retrieves the size (in bits) of the secret key used in the symmetric algorithm
`LegalBlockSizes`	Retrieves the block sizes (in bits) available for a symmetric algorithm
`LegalKeySizes`	Retrieves the key sizes (in bits) available for a symmetric algorithm
`Mode`	Sets or retrieves the operation mode (for example, CBC) of the symmetric algorithm

Table 10.2: Noteworthy Properties of the SymmetricAlgorithm Class	
Property	**Description**
`Padding`	Sets or retrieves the type of padding (if the data is shorter than the fixed number of bytes) used for the symmetric algorithm

Table 10.3 lists the noteworthy methods of the SymmetricAlgorithm class:

Table 10.3: Noteworthy Methods of the SymmetricAlgorithm Class	
Method	**Description**
`Clear()`	Releases the resources used by the SymmetricAlgorithm class
`Create()`	Allows the creation of a cryptographic object for a symmetric algorithm
`CreateDecryptor()`	Creates an object for symmetric decryption
`CreateEncryptor()`	Creates an object for symmetric encryption
`GenerateIV()`	Generates a random IV for a symmetric algorithm
`GenerateKey()`	Generates a random secret key for a symmetric algorithm
`ValidKeySize()`	Allows you to determine if the given key size is acceptable by the current symmetric algorithm

The classes that inherit the SymmetricAlgorithm class correspond to the different symmetric algorithms. Figure 10.2 shows a pictorial representation of the inheritance hierarchy of the different symmetric algorithm classes:

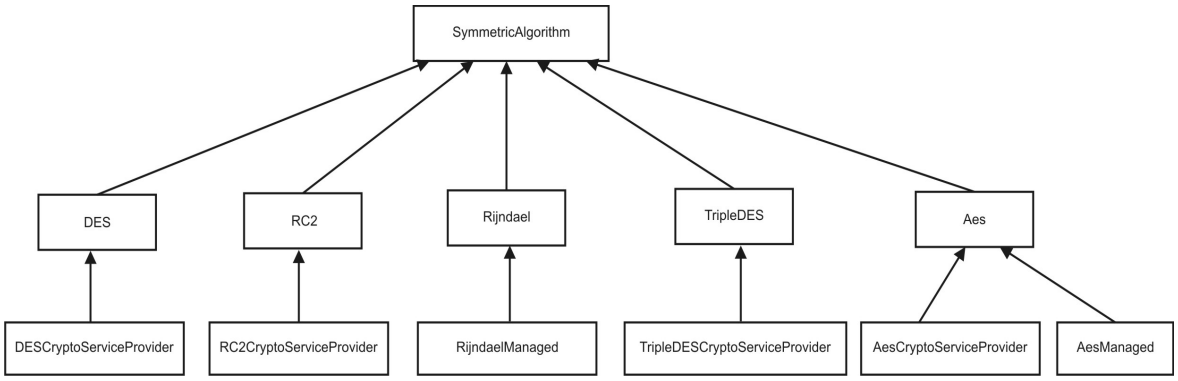

Figure 10.2: Displaying Inheritance Hierarchy of the Symmetric Algorithm Classes Used for Secret-Key Encryption

In Figure 10.2, you can see that the inheritance hierarchy has three tiers. At the root or top tier, the SymmetricAlgorithm class exists as the base class. In the middle tier, various abstract classes, such as DES, RC2, Rijndael, TripleDES, and Aes, implement the different symmetric algorithms.In the bottom tier, there are classes that provide the cryptographic service provider (CSP) or managed implementation of the respective algorithm.

The various symmetric algorithms and their description that are supported by .NET Framework4.0 are following:

- **Data Encryption Standard (DES) Algorithm**—Refers to an algorithm that is a block cipher. The DES algorithm encrypts and decrypts data one block at a time. Each block is 8-bytes or 64-bit long. The DES algorithm uses a 64-bit secret key to encrypt and decrypt the blocks of data. Every eighth bit of the key is ignored by the algorithm, which implies that the key in effect is only 56-bit long. This 56-bit effective length of the DES secret key is smaller as compared to other symmetric algorithms, which makes it relatively less secure.

In .NET Framework, the DES algorithm is implemented through the DESCryptoServiceProvider Class. The`DESCryptoServiceProvider` class encapsulates CSP version of the DES algorithm to allow you to encrypt and decrypt using the DES algorithm. The inheritance hierarchy of the `DESCryptoServiceProvider` class is as follows:

```
System.Object
  System.Security.Cryptography.SymmetricAlgorithm
        System.Security.Cryptography.DES
                System.Security.Cryptography.DESCryptoServiceProvider
```

- **Ron's Code2 or Rivest Cipher2 (RC2) Algorithm**—Refers to an algorithm that encrypts and decrypts data in 64-bit blocks. The secret key has a variable length that can range from 8 bits to 64 bits. Due to the smaller key size, RC2 also does not prove to be a very secure symmetric algorithm. However, this algorithm is twice as faster as the DES algorithm and therefore, can be considered as a replacement for the DES algorithm.

  The RC2 algorithm is implemented in the RC2CryptoServiceProvider class that inherits from the RC2 class. The RC2CryptoServiceProvider class encapsulates the CSP implementation of the RC2 algorithm for performing encryption. The inheritance hierarchy of the RC2CryptoServiceProvider class is as follows:

```
System.Object
  System.Security.Cryptography.SymmetricAlgorithm
        System.Security.Cryptography.RC2
                System.Security.Cryptography.RC2CryptoServiceProvider
```

  The RC2CryptoServiceProvider class has a property named EffectiveKeySize that it inherits from the RC2 class. The EffectiveKeySize property allows you to set or retrieve the effective size or length (in bits) of the secret key that is used by the algorithm.

- **TripleDES Algorithm**—Refers to an algorithm that performs three iterations of the DES algorithm to encrypt and decrypt 64-bit blocks of data. Every iteration may use three different secret keys where each key has a length of 64 bits; the effective length of the individual keys is 56 bits. Due to the three iterations, the TripleDES algorithm is considered as more secure than the DES algorithm; however, the TripleDES algorithm is slower as compared to the DES algorithm.

  In .NET Framework, the TripleDESCryptoServiceProvider class implements the CSP version of the TripleDES algorithm. The TripleDESCryptoServiceProvider class inherits the TripleDES algorithm, as given in the following inheritance hierarchy:

```
System.Object
  System.Security.Cryptography.SymmetricAlgorithm
        System.Security.Cryptography.TripleDES
                System.Security.Cryptography.TripleDESCryptoServiceProvider
```

- **Rijndael Algorithm**—Refers to an algorithm that is one of the advanced and complex symmetric algorithms available today. It is considered as an Advanced Encryption Standard (AES) and is a block cipher that encrypts and decrypts data in 128-, 192-, and 256-bit blocks. The secret key can be 128, 192, or 256 bits long, which is quite large as compared to other symmetric algorithms, such as DES. Due to its long key length, it provides almost unbreakable encrypted data and therefore, it is one of the most secure symmetric algorithms till date. It is also one of the fastest symmetric algorithms.

  In .NET Framework, the RijndaelManaged class contains the managed implementation of the Rijndael algorithm. The inheritance hierarchy of the `RijndaelManaged` class is as follows:

```
System.Object
  System.Security.Cryptography.SymmetricAlgorithm
        System.Security.Cryptography.Rijndael
                System.Security.Cryptography.RijndaelManaged
```

- **Advanced Encryption Standard (AES) Algorithm**—Refers to an algorithm that is actually a subset of the Rijndael algorithm. The secret key used in the AES algorithm is 128-, 192-, or 256-bit long and the data is encrypted and decrypted in blocks of 128 bits. With the AES algorithm, you cannot use the feedback mode.

  The Aes class in .NET Framework is the abstract base class that extends the properties and methods for using the AES algorithm. This class is inherited by two other classes, AesCryptoServiceProvider and AesManagedthat represent the CAPI and the managed implementations of the AES algorithm, respectively. The inheritance hierarchy of both these classes is as follows:

```
System.Object
   System.Security.Cryptography.SymmetricAlgorithm
         System.Security.Cryptography.Aes
                System.Security.Cryptography.AesCryptoServiceProvider
                       System.Security.Cryptography.AesManaged
```

Now that you have learned about the secret-key encryption, let's learn about the public-key encryption.

Public-Key Encryption

In public-key encryption, two keys are used to encrypt and decrypt data. The key used to encrypt the data is known as the public key and the key used to decrypt the data is known as the private key. You need to first generate the public and private keys to use public-key encryption. The public key is then made available to anybody who wants to send data. The sender can then use the public key to encrypt the data and send the encrypted data to you. When you receive the encrypted data, you need to use the private key, which remains hidden from the sender, to decrypt the data. The public and private keys are related to each other in such a way that data encrypted with a public key can be decrypted by using only the correct private key. Note that since you transmit the public key to the sender over the network, any user can intercept the transmission, access the public key, and use it to send encrypted data. Therefore, it is essential to first check the identity of the sender before decrypting the data. You also need to check if the data is encrypted with the same public key that you sent.

Public-key encryption is also known as asymmetric encryption as it uses one key to encrypt and another key to decrypt. The cryptography algorithms used to implement asymmetric encryption are known as asymmetric algorithms. This algorithm generates the private and public keys that are mathematically linked to each other. Public-key or asymmetric algorithms generate keys that are longer as compared to those generated in symmetric or secret-key algorithms. Due to the larger key lengths, asymmetric algorithms offer better security than symmetric algorithms. However, asymmetric algorithms are slow in comparison to the symmetric algorithms and therefore, are suitable only for exchanging small amounts of data. Asymmetric algorithms are primarily used for exchanging secret keys (in secret-key encryption) and creating digital signatures.

In .NET Framework, all the asymmetric algorithms are inherited from the AsymmetricAlgorithm class, which is an abstract class representing an asymmetric algorithm. There are various properties and methods of the AsymmetricAlgorithm class. Table 10.4 lists the noteworthy properties of the AsymmetricAlgorithm class:

Table 10.4: Noteworthy Properties of the AsymmetricAlgorithm Class

Property	Description
`KeyExchangeAlgorithm`	Retrieves the name of the key exchange algorithm
`KeySize`	Sets or retrieves the size of the key used in the asymmetric algorithm in bits
`LegalKeySizes`	Retrieves the key sizes that are available for the asymmetric algorithm
`SignatureAlgorithm`	Retrieves the name of the signature algorithm

Table 10.5 lists the noteworthy methods of the AsymmetricAlgorithm class:

Table 10.5: Noteworthy Methods of the AsymmetricAlgorithm Class

Method	Description
`Clear()`	Frees the resources used by the asymmetric algorithm
`Create()`	Allows you to create a cryptographic object for the asymmetric algorithm
`FromXmlString()`	Allows you to re-create an XML representation of the current asymmetric algorithm
`ToXmlString()`	Creates and returns an XML representation of the current asymmetric algorithm

Several .NET classes allow you to work with the asymmetric algorithms. Figure 10.3 shows the inheritance hierarchy of these classes:

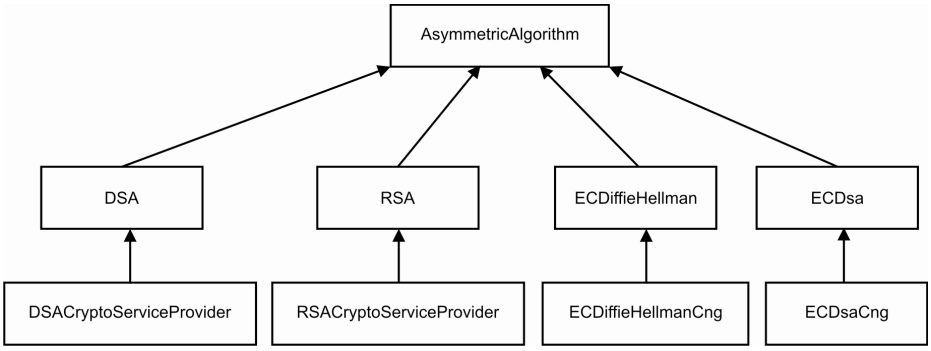

Figure 10.3: Inheritance Hierarchy of the Asymmetric Algorithm Classes Used for Public-Key Encryption

As shown in Figure 10.3, the inheritance hierarchy of the classes is divided into three tiers. The top tier consists of the AsymmetricAlgorithm class, which is the base class for all the asymmetric algorithms. The middle tier has abstract classes (for example, DSA, RSA, ECDiffieHellman, and ECDsa) that provide the properties and methods for a particular asymmetric algorithm. The classes in the bottom tier are the CSP or Cryptography Next Generation (CNG) implementation of the asymmetric algorithms.

NOTE

CNG is a Framework that encapsulates several managed APIs offering extensible cryptographic features and services. It is included in .NET Framework to serve as an alternative to the CryptoAPI.

In .NET Framework4.0, the descriptions of various asymmetric algorithms are as following:

- **Digital Signature Algorithm (DSA)—**Refers to an algorithm that was developed by National Institute of Standard and Technology (NIST) in 1991 for working with digital signatures. It is the United States Federal Government Standard for digital signatures. The DSA algorithm is quite fast and can use keys that have lengths ranging from 512 bits to 1024 bits. The value of length can be incremented in the sets of 64 bits. This algorithm is not used to encrypt or decrypt data, but only to digitally sign the data. The algorithm uses a hash algorithm to create the digital signature of the sender, which is then used by the receiver to authenticate the sender.

  In .NET Framework, the `DSACryptoServiceProvider` class contains the CSP implementation of the DSA algorithm. The inheritance hierarchy of the `DSACryptoServiceProvider` class is as follows:

```
System.Object
  System.Security.Cryptography.AsymmetricAlgorithm
        System.Security.Cryptography.DSA
              System.Security.Cryptography.DSACryptoServiceProvider
```

The `DSACryptoServiceProvider` class has several properties and methods that allow you to use the DSA algorithm in your applications. Table 10.6 lists some of the noteworthy properties of the DSACryptoServiceProvider class:

Table 10.6: Noteworthy Properties of the DSACryptoServiceProvider Class

Property	Description
`CspKeyContainerInfo`	Retrieves an object of the `CspKeyContainerInfo` class. A `CspKeyContainerInfo` object contains different types of information about the private and public key pair.
`PersistKeyInCsp`	Sets or retrieves a value that indicates whether a key should persist in the cryptographic service provider (CSP).
`PublicOnly`	Retrieves a value that indicates whether the `DSACryptoServiceProvider` object contains only a public key or both public and private keys.
`UseMachineKeyStore`	Sets or retrieves a value that indicates whether the key should persist in the key store of the computer or the user profile store.

Table 10.7 lists some of the noteworthy methods of the DSACryptoServiceProvider class:

Table 10.7: Noteworthy Methods of the DSACryptoServiceProvider Class

Method	Description
CreateSignature()	Creates a DSA signature for a particular data
ExportCspBlob()	Exports a blob containing information about the key that is associated with a `DSACryptoServiceProvider` object
ExportParameters()	Exports information about the keys to a DSAParameters object
ImportCspBlob()	Imports a blob that contains DSA key information
ImportParameters()	Imports a given DSAParameters object
SignData()	Allows you to compute the hash value of a given data and signs the data
SignHash()	Allows you to create a digital signature for a given hash value by using the private key to encrypt the hash value
VerifyData()	Returns a value that indicates whether the specified signature matches the signature computed for the given data
VerifyHash()	Returns a value that indicates whether the specified signature matches the signature computed for the given hash value
`VerifySignature()`	Returns a value that indicates whether the given DSA signature matches the signature computed for the specified data

- **Random Sequential Adsorption (RSA)—**Refers to an algorithm that was introduced in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman and therefore, named after them. You can use the RSA algorithm for digitally signing and encrypting data. This algorithm can use keys with a minimum length of 384 bits. With a key length of 1024 bits or more, RSA is considered as more secure than DSA.

In .NET Framework, the `RSACryptoServiceProvider` class provides the CSP implementation of the RSA algorithm. The inheritance hierarchy of the `RSACryptoServiceProvider` class is as follows:

```
System.Object
 System.Security.Cryptography.AsymmetricAlgorithm
        System.Security.Cryptography.RSA
                System.Security.Cryptography.RSACryptoServiceProvider
```

The `RSACryptoServiceProvider` class has many properties and methods that you can use to work with the RSA algorithm. Table 10.8 lists some of the noteworthy properties of the RSACryptoServiceProvider class:

Table 10.8: Noteworthy Properties of the RSACryptoServiceProvider Class

Property	Description
CspKeyContainerInfo	Retrieves an object of the `CspKeyContainerInfo` class. A `CspKeyContainerInfo` object contains information about the private and public key pair.
PersistKeyInCsp	Sets or retrieves a value that indicates whether a key should persist in the cryptographic service provider (CSP).
PublicOnly	Retrieves a value that indicates whether the `RSACryptoServiceProvider` object contains only a public key or both the public and private keys.
UseMachineKeyStore	Sets or retrieves a value that indicates whether the key should persist in the key store of the computer or the user profile store.

Table 10.9 lists some of the noteworthy methods of the RSACryptoServiceProvider class:

Table 10.9: Noteworthy Methods of the RSACryptoServiceProviderClass	
Method	**Description**
`Decrypt()`	Allows the decryption of data with the RSA algorithm
`Encrypt()`	Allows encryption of data with the RSA algorithm
`ExportCspBlob()`	Exports a blob containing information about the key that is associated with a `RSACryptoServiceProvider` object
`ExportParameters()`	Exports information about the keys to an RSAParameters object
`ImportCspBlob()`	Imports a blob that contains RSA key information
`ImportParameters()`	Imports a given RSAParameters object
`SignData()`	Allows to the computation of the hash value of a given data and signs the data
`SignHash()`	Allows the creation of a digital signature for a given hash value by using the private key to encrypt the hash value
`VerifyData()`	Returns a value that indicates whether the specified signature matches the signature computed for the given data
`VerifyHash()`	Returns a value that indicates whether the specified signature matches the signature computed for the given hash value

- **Elliptic Curve Diffie Hellman Algorithm (ECDH)—**Refers to an algorithm that is primarily used for the generation of private and public keys rather than encryption and decryption of data. The ECDH algorithm is provided by .NET Framework4.0 through the Cryptography Next Generation (CNG) classes. The ECDiffieHellmanCng class is the class that contains the implementation of the ECDH algorithm.The inheritance hierarchy of the ECDiffieHellmanCng class is as follows:

```
System.Object
  System.Security.Cryptography.AsymmetricAlgorithm
        System.Security.Cryptography.ECDiffieHellman
              System.Security.Cryptography.ECDiffieHellmanCng
```

The ECDiffieHellmanCng class has many properties and methods that you can use to work with the ECDH algorithm. Table 10.10 lists some of the noteworthy properties of the ECDiffieHellmanCng class:

Table 10.10: Noteworthy Properties of the ECDiffieHellmanCng Class	
Property	**Description**
`HashAlgorithm`	Sets or retrieves the hash algorithm used to generate the keys. The default hash algorithm is SHA256.
`HmacKey`	Sets or retrieves the Hash-based Message Authentication Code (HMAC) key in use.
`Key`	Refers to a CngKey object that is used by the ECDH algorithm.
`KeyDerivationFunction`	Sets or retrieves the key derivation function used by the algorithm to convert the secret agreement to key information.
`Label`	Sets or retrieves the label used for deriving the key. By default, it is null.
`PublicKey`	Retrieves the public key used by one ECDiffieHellmanCngobject to share the key with another ECDiffieHellmanCng object.
`SecretAppend`	Sets or retrieves a value that is appended to the secret agreement.
`SecretPrepend`	Sets or retrieves a value that is included at the very beginning of the secret agreement.
`Seed`	Sets or retrieves the seed value for generating the key. By default, it is a null reference.
`UseSecretAgreementAsHmacKey`	Retrieves a value that indicates whether the secret agreement is used as a HMAC key.

Table 10.11 lists the noteworthy methods of the ECDiffieHellmanCng class:

Table 10.11: Noteworthy Methods of the ECDiffieHellmanCngClass	
Method	**Description**
`DeriveKeyMaterial()`	Generates the key material from a given secret agreement
`DeriveSecretAgreementHandle()`	Returns a handle for a secret agreement

- **Elliptic Curve Digital Signature Algorithm (ECDSA)**—Refers to an algorithm that has its use in digital signatures. The sender can digitally sign data with its private key and the receiver can validate the identity of the sender by using the sender's public key. The ECDSA algorithm is provided by .NET Framework4.0 through the ECDsaCng class that has the CNG implementation of the ECDSA algorithm. The inheritance hierarchy of the ECDsaCng class is as follows:

```
System.Object
   System.Security.Cryptography.AsymmetricAlgorithm
         System.Security.Cryptography.ECDsa
                  System.Security.Cryptography.ECDsaCng
```

The ECDsaCng class has many properties and methods that you can use to work with the ECDSA algorithm. Table 10.12 lists some of the noteworthy properties of the ECDsaCng class:

Table 10.12: Noteworthy Properties of the ECDsaCngClass	
Property	**Description**
`HashAlgorithm`	Sets or retrieves the hash algorithm used for digitally signing and validating the data
`Key`	Sets or retrieves the key used for digitally signing and checking the data

Table 10.13 lists the noteworthy methods of the ECDsaCng class:

Table 10.13: Noteworthy Methods of the ECDsaCngClass	
Method	**Description**
`SignData()`	Creates a digital signature
`SignHash()`	Creates a digital signature for a given hash value
`VerifyData()`	Verifies a given digital signature
`VerifyHash()`	Verifies a given digital signature against a hash value

Hash Algorithms

Hash algorithms are hash functions that take a sequence of bytes of any length and convert it into smaller fixed-length sequence of bytes known as hash values. The hash values, also called message digest or digital fingerprint, represent the alphanumerical/numerical form of the data. The hash values are randomly generated and are unique for a particular piece of data (or message). That is, the hash values of two different pieces of data are also different. Even if two pieces of data or messages differ in a single letter, their hash values can be significantly different. Therefore, it can be considered that if two hash values are same, then the respective data or messages are also same. Due to this uniqueness of hash values, cryptographic hash algorithms are used for authenticating the sender and ensuring data integrity.

In .NET Framework, the class that represents all the cryptographic hash algorithms is the HashAlgorithm class. This class is an abstract class that is inherited by any class that implements a hash algorithm. The HashAlgorithm class has several properties and methods to work with the hash algorithms.

Table 10.14 lists the noteworthy properties of the HashAlgorithm class:

Table 10.14: Noteworthy Properties of the HashAlgorithmClass	
Property	**Description**
`CanReuseTransform`	Retrieves a value that indicates whether the current transform is reusable
`CanTransformMultipleBlocks`	Retrieves a value that indicates whether transformation can be performed on multiple blocks
`Hash`	Retrieves the hash value
`HashSize`	Retrieves the size (in bits) of the hash value
`InputBlockSize`	Retrieves the size of the input block
`OutputBlockSize`	Retrieves the size of the output block

Table 10.15 lists the noteworthy methods of the HashAlgorithm class:

Table 10.15: Noteworthy Methods of the HashAlgorithmClass	
Method	**Description**
`Clear()`	Frees all the resources used by the current HashAlgorithm object
`ComputeHash()`	Computes the hash value for a given input data
`Create()`	Creates a HashAlgorithm object
`Initialize()`	Initializes an implementation of the HashAlgorithm class
`TransformBlock()`	Computes the hash value for a particular portion of the input byte array and saves it in a given portion of the output byte array
`TransformFinalBlock()`	Computes the hash value for a given portion of a given byte array

The various algorithms that are supported in .NET Framework4.0 are:

- Message-Digest Algorithm 5 (MD5)
- RACE Integrity Primitives Evaluation Message Digest-160(RIPEMD-160) Algorithm
- Secure Hash Algorithms (SHA)

MD5 Algorithm

The MD5 algorithm was introduced by Ron Rivest in the early 1990s to enforce data integrity and identity validation for data. It takes a message (or data) of any length and generates a hash value called message digest of 128 bits..NET Framework allows you to work with the MD5 algorithm through the MD5 base class, which is an abstract class that offers the properties and methods for the actual implementation of the algorithm. The MD5CryptoServiceProvider class provides the CSP implementation of the MD5 algorithm. The inheritance hierarchy of the `MD5CryptoServiceProvider` class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.MD5
                System.Security.Cryptography.MD5CryptoServiceProvider
```

In .NET Framework4.0, another class, named MD5Cng, provides the CNG implementation of the algorithm. The inheritance hierarchy of the MD5Cng class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.MD5
                System.Security.Cryptography.MD5Cng
```

RIPEMD-160 Algorithm

The RIPEMD-160 algorithm is an enhanced version of the MD5 algorithm. The RIPEMD-160 generates a 160-bit message digest as compared to the 128-bit message digest by MD5 algorithm..NET Framework provides the

RIPEMD160 abstract class, which is the base class for all the classes that implement the RIPEMD-160 algorithm. The RIPEMD160Managed class is an in-built class that implements the algorithm using the managed library. The inheritance hierarchy of the RIPE160Managed class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.RIPEMD160
              System.Security.Cryptography.RIPEMD160Managed
```

SHA Algorithms

The SHA algorithms refer to a set of hash algorithms developed by the National Institute of Standards and Technology (NIST). The SHA algorithms are based on the MD5 algorithm and are comparatively more secure due to their longer message digests. The various SHA algorithms supported by .NET Framework are:

- **SHA1**—Generates a 160-bit message digest and is one of the most common SHA algorithms. The SHA1 class provided by .NET Framework is the abstract class that facilitates the implementation of the SHA1 algorithm. The managed implementation of the SHA1 algorithm is provided by the SHA1Managed class. The inheritance hierarchy of the SHA1Managed class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA1
              System.Security.Cryptography.SHA1Managed
```

The `SHA1CryptoServiceProvider` class has the CSP implementation of the algorithm by inheriting the SHA1 class. The inheritance hierarchy of the `SHA1CryptoServiceProvider` class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA1
              System.Security.Cryptography.SHA1CryptoServiceProvider
```

In .NET Framework4.0, there is the SHA1Cng class that provides the CNG implementation of the SHA1 algorithm. The inheritance hierarchy of the SHA1Cng class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA1
              System.Security.Cryptography.SHA1Cng
```

- **SHA256**—Generates a 256-bit message digest and is supported by .NET Framework through the SHA256 abstract base class. The classes that implement the SHA256 algorithm inherit the SHA256 class. The managed implementation of the SHA256 algorithm is provided by the `SHA256Managed` class. The inheritance hierarchy of the `SHA256Managed` class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA256
              System.Security.Cryptography.SHA256Managed
```

In .NET Framework4.0, the SHA256CryptoServiceProvider class provides the CSP implementation of the SHA256 algorithm. The inheritance hierarchy of the SHA256CryptoServiceProvider is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA256
              System.Security.Cryptography.SHA256CryptoServiceProvider
```

The CNG implementation of the algorithm is also provided in .NET Framework through the SHA256Cng class. The inheritance hierarchy of the SHA256Cng class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA256
              System.Security.Cryptography.SHA256Cng
```

- **SHA384**—Generates a 384-bit message digest and is supported by .NET Framework through the SHA384 abstract base class. The classes that implement the SHA384 algorithm inherit the SHA256 class. The managed implementation of the SHA384 algorithm is provided by the `SHA384Managed` class. The inheritance hierarchy of the `SHA384Managed` class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA384
                System.Security.Cryptography.SHA384Managed
```

In .NET Framework 4.0, the SHA384CryptoServiceProvider class provides the CSP implementation of the SHA384 algorithm. The inheritance hierarchy of the SHA384CryptoServiceProvider is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA384
                System.Security.Cryptography.SHA384CryptoServiceProvider
```

The CNG implementation of the algorithm is also provided in .NET Framework through the SHA384Cng class. The inheritance hierarchy of the SHA384Cng class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA384
                System.Security.Cryptography.SHA384Cng
```

- **SHA512—**Generates a 512-bit message digest and is supported by .NET Framework through the SHA512 abstract base class. The classes that implement the SHA512 algorithm inherit the SHA512 class. The managed implementation of the SHA512 algorithm is provided by the `SHA512Managed` class. The inheritance hierarchy of the `SHA512Managed` class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA512
                System.Security.Cryptography.SHA512Managed
```

In .NET Framework 4.0, the SHA512CryptoServiceProvider class provides the CSP implementation of the SHA512 algorithm. The inheritance hierarchy of the SHA512CryptoServiceProvider class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA512
                System.Security.Cryptography.SHA512CryptoServiceProvider
```

The CNG implementation of the algorithm is also provided in .NET Framework through the SHA512Cng class. The inheritance hierarchy of the SHA512Cng class is as follows:

```
System.Object
  System.Security.Cryptography.HashAlgorithm
        System.Security.Cryptography.SHA512
                System.Security.Cryptography.SHA512Cng
```

NOTE

The SHA256, SHA384, and SHA512 algorithms are collectively known as SHA2 algorithms.

Immediate Solutions

Using Secret-Key Encryption

You learned in the *In Depth* section how a single secret key is used to encrypt and decrypt the data. You also learned about the various secret-key or symmetric algorithms such as DES, RC2, TripleDES, Rijndael, and AES. In this section, you see how to use secret-key encryption in your .NET applications to protect the data used by the applications.

Let's create a new Windows Forms application, named SecretKeyFile (also on the CD-ROM) by performing the following steps:

1. In the Visual Studio 2010 IDE, select File→New→Project to open the New Project dialog box. In the dialog box, select the Visual C#→Windows option in the Project types pane and the Windows Forms Application option in the Templates pane. Specify SecretKeyFile as the name of the application, select an appropriate location, and click the OK button. The SecretKeyFile application is created.
2. In the designer view of Form1, add three buttons, four labels, two text boxes, a status strip, and an open file dialog on the form as shown in Figure 10.4.

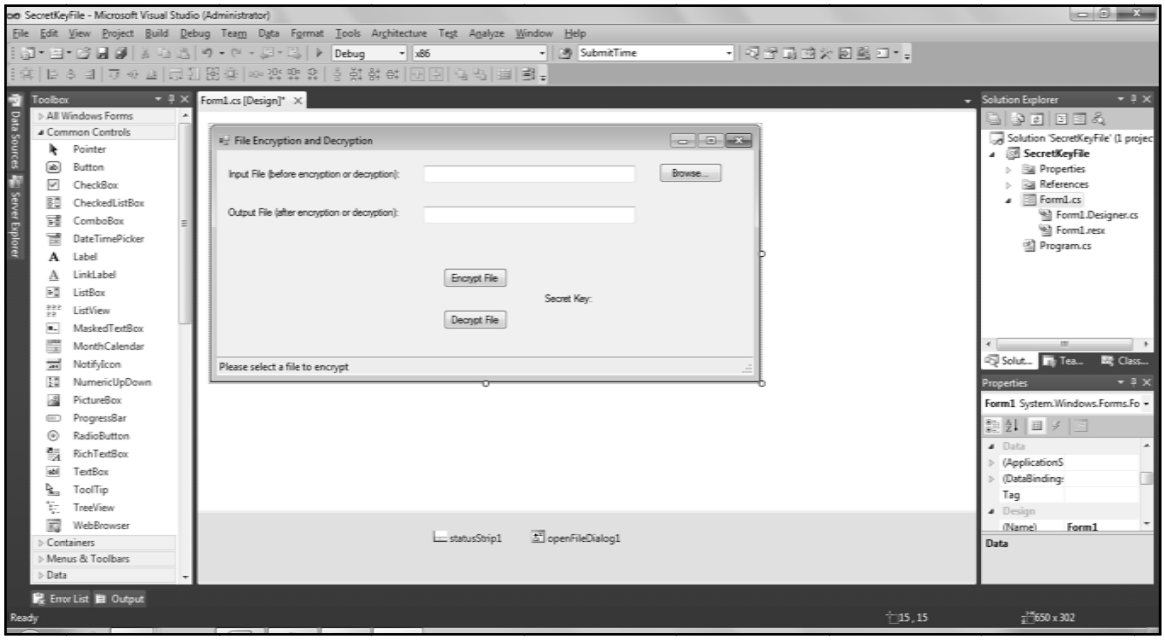

Figure 10.4: Displaying the SecretKeyFile Application at Design Time

In the SecretKeyFile application, DES algorithm is used to demonstrate how to generate secret key, encrypt data with the key, and then decrypt data using the same key.

3. Open the code-behind file of Form1 (Form1.cs) and add the code, given in Listing 10.1, to reference the necessary namespace:

Listing 10.1: Adding the Namespace References

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
```

```
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Security;
using System.Security.Cryptography;
using System.Runtime.InteropServices;

namespace SecretKeyFile
{

 public partial class Form1 : Form
 {
         public Form1()
         {
                 InitializeComponent();
         }
         //  Call this function to remove the key from memory after use
          [System.Runtime.InteropServices.DllImport("KERNEL32.DLL", EntryPoint =
         "RtlZeroMemory")]
         public static extern bool ZeroMemory(IntPtr Destination, int Length);
         private void Form1_Load(object sender, EventArgs e)
         {
                 button3.Enabled = false;
         }

         private void textBox1_TextChanged(object sender, EventArgs e)
         {
                 if (button3.Enabled == false)
                 toolStripStatusLabel1.Text = "Please specify an input file to encrypt.";
                 else
                 toolStripStatusLabel1.Text = "Please specify an input file to decrypt.";
         }

         private void textBox2_TextChanged(object sender, EventArgs e)
         {
                 if (button3.Enabled == false)
                 toolStripStatusLabel1.Text = "Output file path specified. Press the
                 Encrypt File button for encrypting the file.";
                 else
                 toolStripStatusLabel1.Text  =  "Output  file  path  specified.  Press  the
                 Decrypt File button for decrypting the file.";
         }

         private void button1_Click(object sender, EventArgs e)
         {
                 openFileDialog1.Filter = "txt files (*.txt)|*.txt";
                 if (openFileDialog1.ShowDialog() == DialogResult.OK)
                 {
                         textBox1.Text = openFileDialog1.FileName.ToString();
                         toolStripStatusLabel1.Text = "Input file selected.Specify the
                         path for the output file.";
                 }
         }
}

}
```

In Listing 10.1, four using statements to reference the System.IO, System.Security, System.Security.Cryptography, and System.Runtime.InteropServices namespaces are added. The textBox1 and button1 allow you to specify an input file for encryption or decryption. In textBox2, you need to specify the output file for encryption or decryption.

4. Add the code, given in Listing 10.2, in the Form1 class to generate the secret key and encrypt a given file:

Listing 10.2: Generating the Secret Key and Encrypting a File

```
// Method to generate a 64-bit secret key for the DES algorithm
static string GenerateSecretKey()
{

 // Create an object of the symmetric algorithm. A default secret key and IV are
 generated.
 DESCryptoServiceProvider desCrypto =
 (DESCryptoServiceProvider)DESCryptoServiceProvider.Create();

 // Return the default key
 return ASCIIEncoding.ASCII.GetString(desCrypto.Key);
}
private void button2_Click(object sender, EventArgs e)
{
 string MySecretKey;
 // Get the secret key for encryption
 MySecretKey = GenerateSecretKey();
 label4.Text = MySecretKey;
 // For additional security, pin the key
 GCHandle gch = GCHandle.Alloc(MySecretKey, GCHandleType.Pinned);
 // Encrypt the input file with the default secret key
 EncryptFile(@textBox1.Text, @textBox2.Text, MySecretKey);
 button3.Enabled = true;
 toolStripStatusLabel1.Text = "Specify the input file to be decrypted.";
}
static void EncryptFile(string sInputFilename, string sOutputFilename, string sKey)
{
 //Open the input file
 FileStream FileStreamInputContainer = new FileStream(sInputFilename, FileMode.Open,
 FileAccess.Read);
 //Start encrypting the file
 FileStream fsEncrypted = new FileStream(sOutputFilename, FileMode.Create,
 FileAccess.Write);
 DESCryptoServiceProvider DES = new DESCryptoServiceProvider();

 //A 64-bit key and IV are required for the DES algorithm
 //Set secret key for the DES algorithm

 DES.Key = ASCIIEncoding.ASCII.GetBytes(sKey);
 //Set the IV for the DES algorithm
 DES.IV = ASCIIEncoding.ASCII.GetBytes(sKey);

 ICryptoTransform desencrypt = DES.CreateEncryptor();
 CryptoStream cryptostream = new CryptoStream(fsEncrypted, desencrypt,
 CryptoStreamMode.Write);
 byte[] bytearrayinput = new byte[FileStreamInputContainer.Length];
 FileStreamInputContainer.Read(bytearrayinput, 0, bytearrayinput.Length);
 cryptostream.Write(bytearrayinput, 0, bytearrayinput.Length);
```

```
    cryptostream.Close();
    FileStreamInputContainer.Close();
    fsEncrypted.Close();

   }
```

In Listing 10.2, the GenerateSecretKey() method has the code to generate the secret key and IV for the DES algorithm. This method uses the Create() method of a DESCryptoServiceProvider object and returns the key as a string. In the `button2_Click()` method, the `GenerateSecretKey()` method is called to generate the secret key and IV. The `button2_Click()` method also contains a call to the EncryptFile() method that encrypts the given input file by using a`DESCryptoServiceProvider` object. The Key and IV properties of the DESCryptoServiceProvider object is set to the secret key that was generated earlier. The encrypted data is saved in the given output file by using a CryptoStream object.

5. Add the code (given in Listing 10.3) in the Form1.cs file to decrypt the encrypted data:

Listing 10.3: Adding the Code to Decrypt the Encrypted File

```
   private void button3_Click(object sender, EventArgs e)
   {
    string MySecretKey;
    // Get the secret key for decrypting the file
    MySecretKey = label4.Text;
    DecryptFile(@textBox1.Text, @textBox2.Text, MySecretKey);
   }

   static void DecryptFile(string sInputFilename, string sOutputFilename, string sKey)
   {
    DESCryptoServiceProvider DES = new DESCryptoServiceProvider();
    DES.Key = ASCIIEncoding.ASCII.GetBytes(sKey);
    DES.IV = ASCIIEncoding.ASCII.GetBytes(sKey);
    //Create a file stream to read the decrypted file
    FileStream fsread = new FileStream(sInputFilename,FileMode.Open,FileAccess.Read);
    //Create a DES decryptor object for the current DES object
    ICryptoTransform desdecrypt = DES.CreateDecryptor();
    //Create a crypto stream to read and do a
    //DES decryption transform on incoming bytes
    CryptoStream cryptostreamDecr = new
    CryptoStream(fsread,desdecrypt,CryptoStreamMode.Read);
    //Store the decrypted data in the given output file
    StreamWriter fsDecrypted = new StreamWriter(sOutputFilename);
    fsDecrypted.Write(new StreamReader(cryptostreamDecr).ReadToEnd());
    fsDecrypted.Flush();
    fsDecrypted.Close();

   }
```

In Listing 10.3, the button3_Click() method has a string variable to hold the secret key for decrypting the encrypted file. The method also has a call to another method named DecryptFile.The DecryptFile() performs the decryption of the encrypted file. It uses the same secret key that was used for encryption by accessing the value of label4 where it was stored after the generation of the key. The DecryptFile() uses a DESCryptoServiceProvider object to decrypt the file and store the decrypted data in the given output file.

6. Create a text file named MyText with the content shown in Figure 10.5.

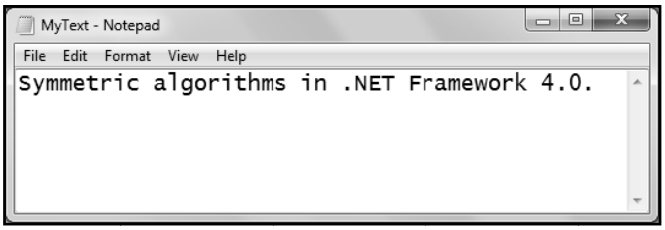

Figure 10.5: Displaying the Content of the file that Has to be Encrypted

7. Add the MyText.txt file in the current project file of the SecretKeyFile application and run the SecretKeyFile application by pressing F5. The output of the application appears, as shown in Figure 10.6.

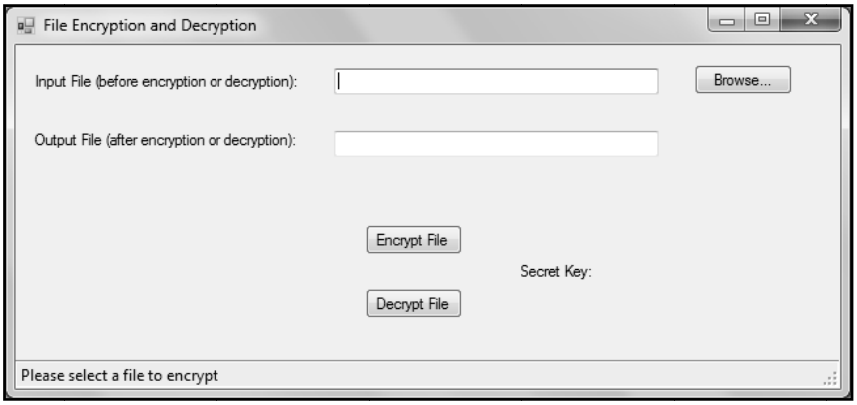

Figure 10.6: Showing the Output of the SecretKeyFile Application

8. Click the Browse button to display the Open dialog box for locating and selecting the MyText.txt file, as shown in Figure 10.7.

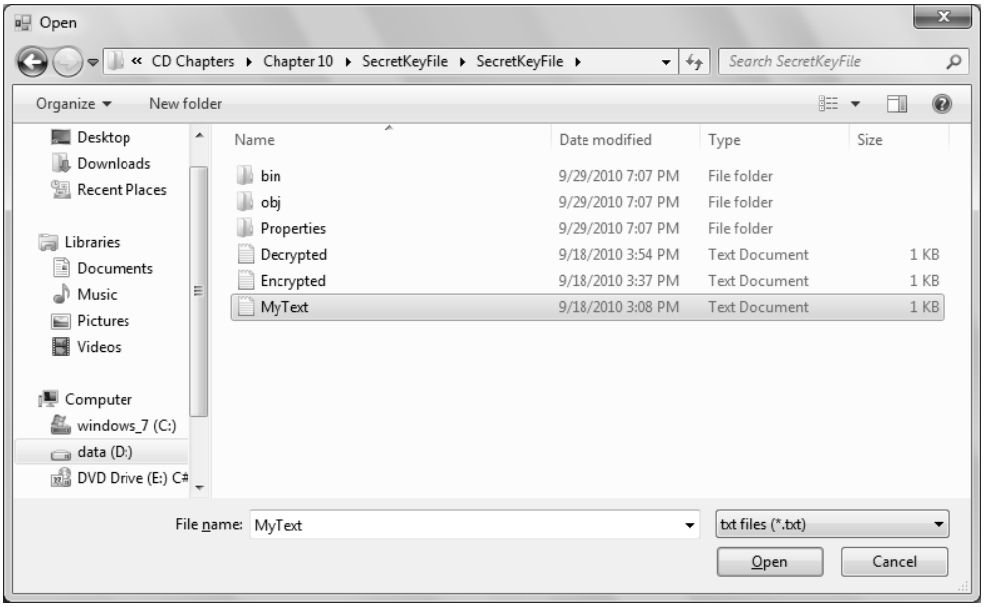

Figure 10.7: Selecting the MyText.txt file for Encryption

9. Click the Open button after selecting the MyText.txt file,. The path to the MyText.txt file is automatically added to the text box next to Input File. Specify the path for the output file (after the MyText.txt file is encrypted) in the text box next to Output File, as shown in Figure 10.8.

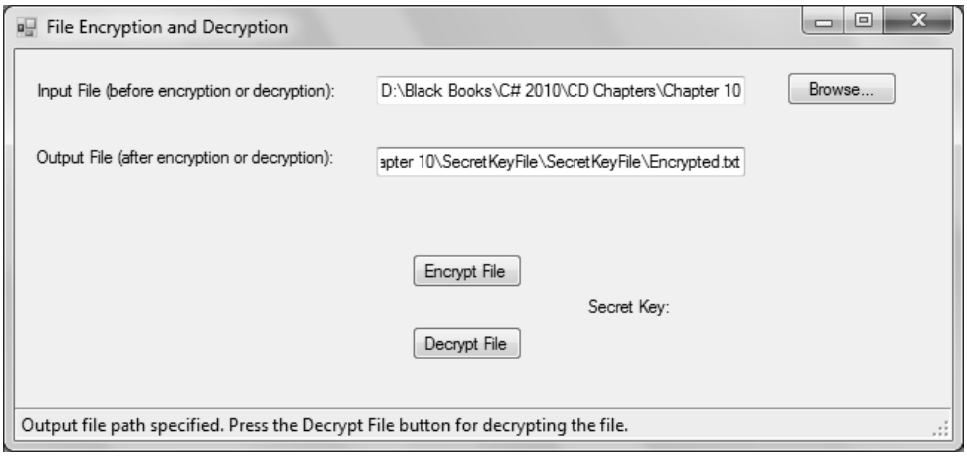

Figure 10.8: Specifying the Input and Output Files for Encryption

10. Click the Encrypt File button, which generates the secret key and uses it to encrypt the MyText.txt file. The secret key appears next to the Secret Key label as shown in Figure 10.9.

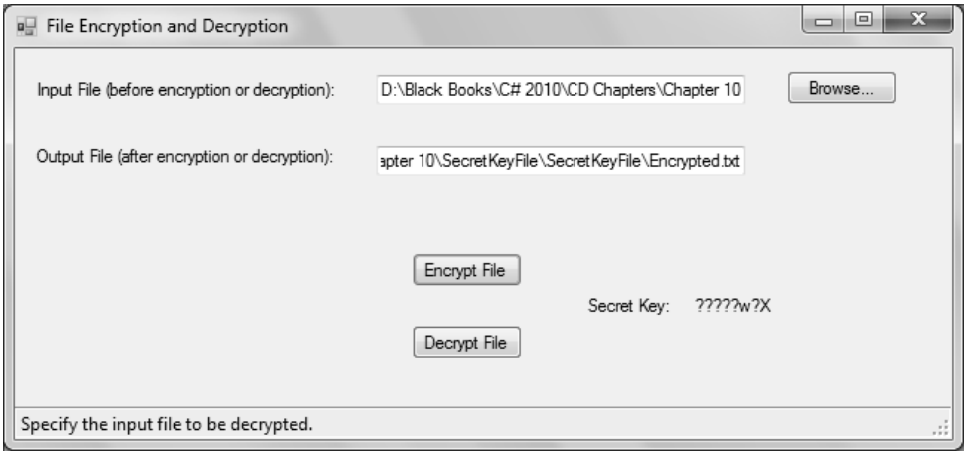

Figure 10.9: Encrypting MyText.txt File by Using the Secret Key

If you now open the Encrypted.txt file, you can see its contents, as shown in Figure 10.10.

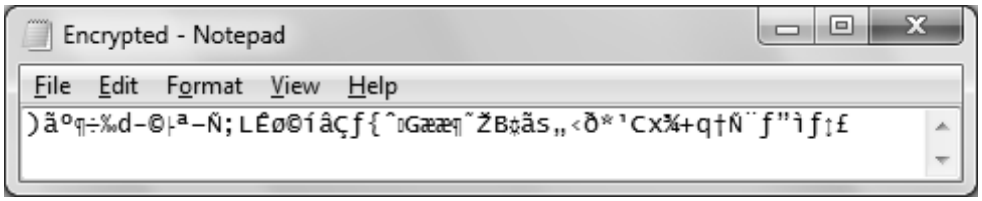

Figure 10.10: Displaying the Encrypted Data of the MyText.txt File

Let's now, decrypt the encrypted file to regenerate the plain text content of the MyText.txt file.

11. Specify the Encrypted.txt as the input file and another file, Decrypted.txt, as the output file, as shown in Figure 10.11.

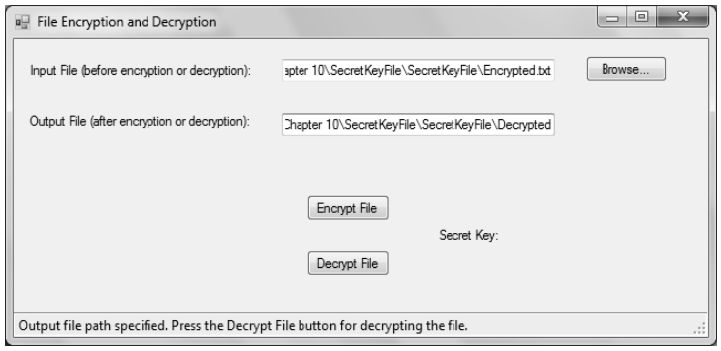

Figure 10.11: Specifying the Input and Output Files for Decryption

12. Click the Decrypt File button, which decrypts the Encrypted.txt file using the same secret key used for encryption. The contents of the Decrypted.txt file are shown in Figure 10.12.

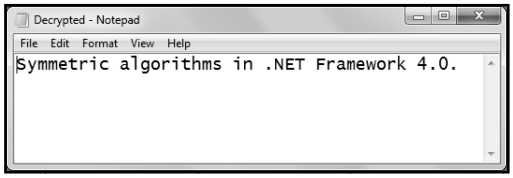

Figure 10.12: Displaying the Content of the Decrypted File

Using Public-Key Encryption

As you know that public-key encryption involves the creation of a private/public key pair. The public key is used to encrypt the data and the private key is used to decrypt the data. Let's create a Windows Forms application named `PublicKeyEncryption`(available on the CD-ROM) by performing the following steps:

1. In the Visual Studio 2010 IDE, select File→New→Project to open the New Project dialog box. In the dialog box, select the Visual C#→Windows option in the Project types pane and the Windows Forms Application option in the Templates pane. Specify PublicKeyEncryption as the name of the application, select an appropriate location, and click the OK button. The PublicKeyEncryption application is created.
2. In the designer view of Form1, add five labels, five text boxes, and three buttons in the PublicKeyEncryption application, as shown in Figure 10.13.

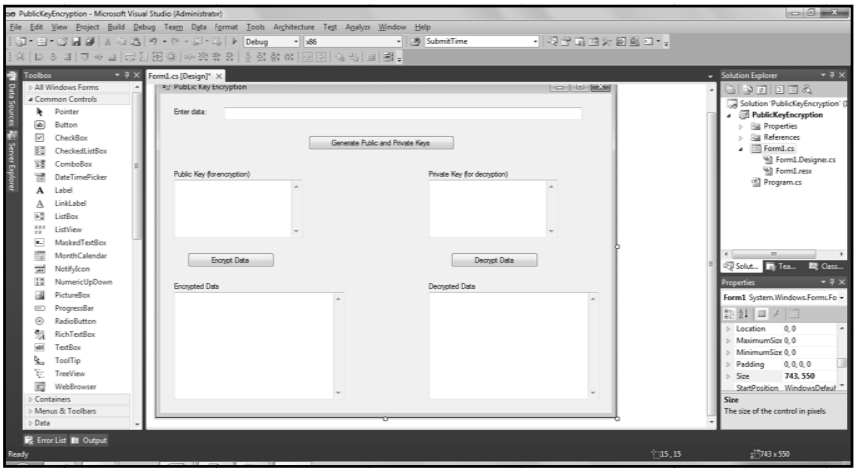

Figure 10.13: Displaying the PublicKeyEncryption Application at the Design Time

Although there are several asymmetric algorithms available, let's use the RSA algorithm to encrypt and decrypt data.

3. Open the Form1.cs file and add the code, given in Listing 10.4:

Listing 10.4: Showing the Code for the Form1.cs File of the PublicKeyEncryption Application

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;

namespace PublicKeyEncryption
{
 public partial class Form1 : Form
 {
         public Form1()
         {
                 InitializeComponent();
                 button2.Enabled = false;
                 button3.Enabled = false;
         }

         string publicKey, privateKey;
         byte[] EncryptedStrAsByt;
 }

}
```

In Listing 10.4, a using directive for the System.Security.Cryptography namespace is added. The button2 and button3 are disabled initially. Three variables named publicKey, privateKey, and EncryptedStrAsByt are declared to hold the public key, private key, and the encrypted data in bytes respectively.

4. Add the code given in Listing 10.5 to the Form1 class in the Form1.cs file for generating the private and public keys.

Listing 10.5: Generating the Public and Private Keys

```
private void button1_Click(object sender, EventArgs e)
{
 CspParameters cspParam = new CspParameters();
 cspParam.Flags = CspProviderFlags.UseMachineKeyStore;
 RSACryptoServiceProvider RSA = new RSACryptoServiceProvider(cspParam);
 publicKey = RSA.ToXmlString(false); // gets the public key
 privateKey = RSA.ToXmlString(true); // gets the private key
 textBox2.Text = publicKey.ToString();
 textBox3.Text = privateKey.ToString();
 button2.Enabled = true;
}
```

In Listing 10.5, the `button1_Click()` method has an `RSACryptoServiceProvider` object that is used to generate the keys. The public key and the private key are then displayed in the textBox2 and textBox3 text boxes respectively.

5. Add the code, given in Listing 10.6, in the Form1.cs file to encrypt the data:

Listing 10.6: Showing the Code to Encrypt the Data

```
private void button2_Click(object sender, EventArgs e)
{
 string str = textBox1.Text;
```

```
    RSACryptoServiceProvider RSA2 = new RSACryptoServiceProvider();
    // Load the public key to Encrypt Data
    RSA2.FromXmlString(publicKey);
    //Start Encrypting Data
    EncryptedStrAsByt = RSA2.Encrypt(Encoding.UTF8.GetBytes(str), false);
    object EncryptedStrAsString = Encoding.UTF8.GetString(EncryptedStrAsByt);
    //Display Encrypted Data
    textBox4.Text = EncryptedStrAsString.ToString();
    button3.Enabled = true;
   }
```

In Listing 10.6, an RSACryptoServiceProvider object is used to encrypt the data entered in textBox1. The Encrypt() method of the RSACryptoServiceProvider object takes the data as one of its arguments and displays it in textBox4.

6. Add the code, given in Listing 10.7, in the Form1.cs file to decrypt the encrypted data:

Listing 10.7: Showing the Code to Decrypt the Encrypted Data

```
   private void button3_Click(object sender, EventArgs e){
    //Start Decrypting Data
    RSACryptoServiceProvider RSA3 = new RSACryptoServiceProvider();
    RSA3.FromXmlString(privateKey);
    byte[] DecryptedStrAsByt = RSA3.Decrypt(EncryptedStrAsByt, false);
    object DecryptedStrAsString = Encoding.UTF8.GetString(DecryptedStrAsByt);
    //Display Decrypted Data
    textBox5.Text = DecryptedStrAsString.ToString();
   }
```

In Listing 10.7, the Decrypt() method of the RSACryptoServiceProvider object decrypts the encrypted data.

7. Run the application and type some text in text box next to Enter Data, as shown in Figure 10.14.

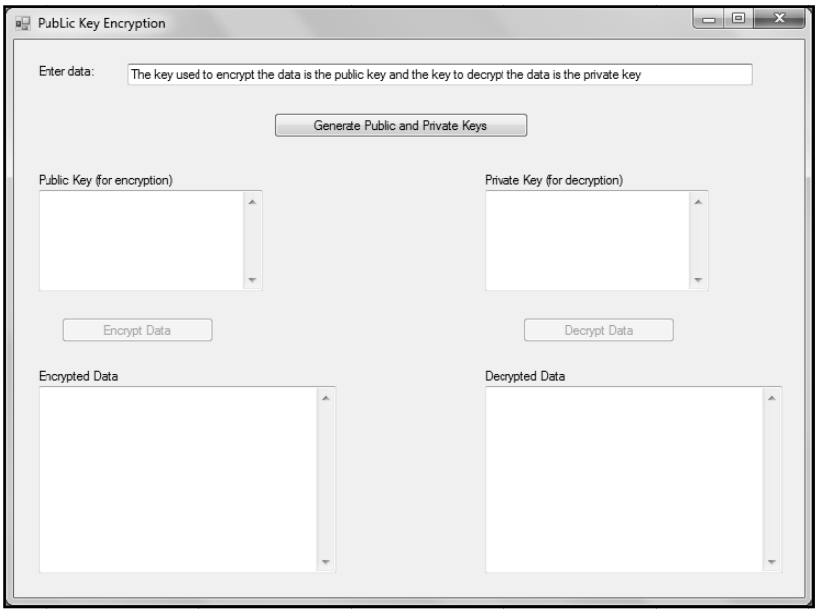

Figure 10.14: Entering the Data

8. Click the Generate Public and Private Keys button to generate the keys, which are displayed in the text boxes below Public Key (for encryption) and Private Key (for decryption), as shown in Figure 10.15.

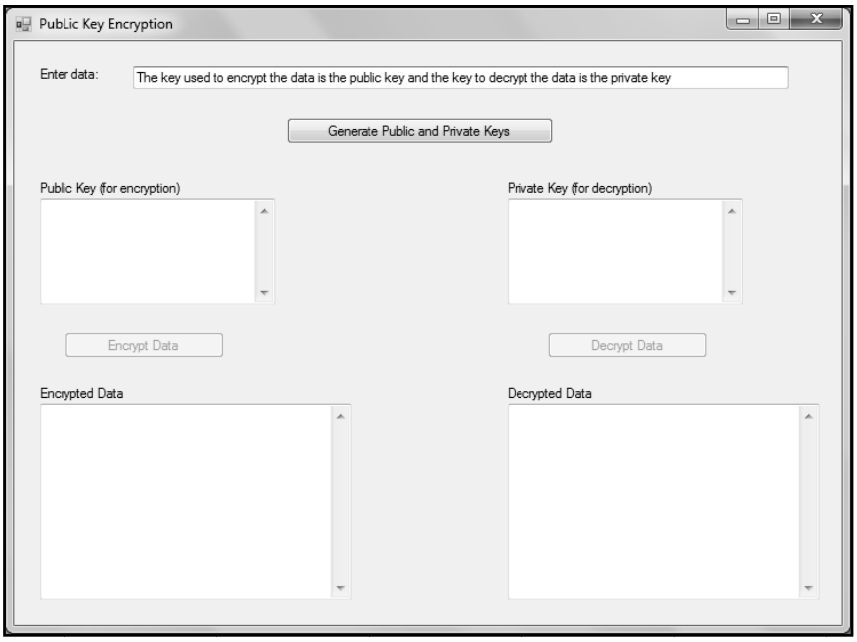

Figure 10.15: Generating Public and Private Keys

9. Click the Encrypt Data button to see how the data is encrypted. The encrypted data appears in the text box below Encrypted Data, as shown in Figure 10.16.

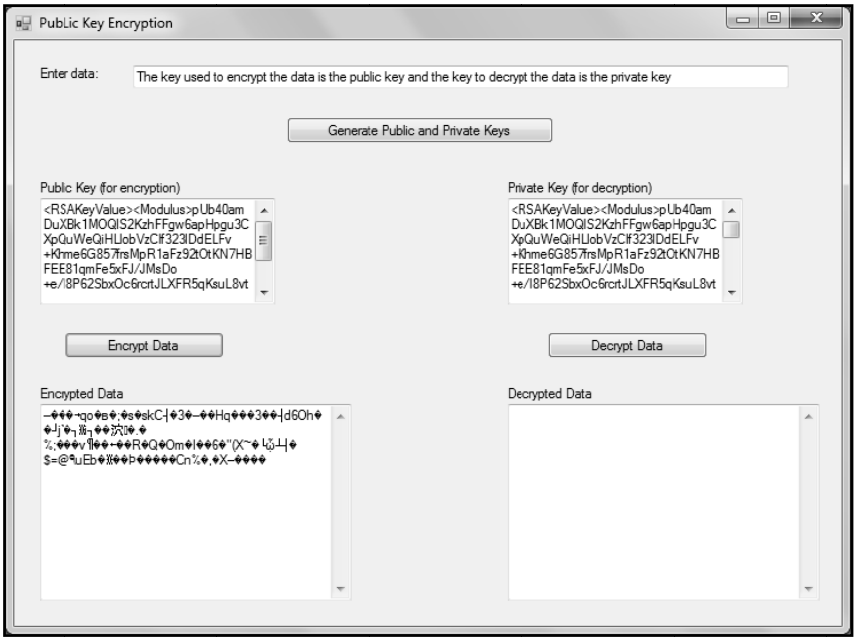

Figure 10.16: Encrypting the Data

10. Click the Decrypt Data button to decrypt the encrypted data. You can see the decrypted data in Figure 10.17.

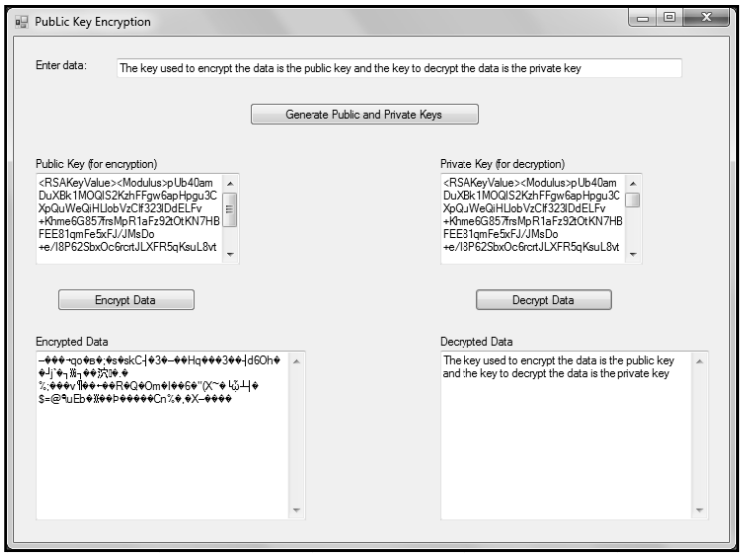

Figure 10.17: Decrypting the Data

Using Hash Algorithms

In the In-Depth section, you learned how hash algorithms allow conversion of data to a compressed, alphanumerical form called message digest, which you can later encrypt and send to another user. Let's create a Windows Forms application named HashDemo (on the CD-ROM) to demonstrate hashing of data. For this, perform the following steps:

1. In the Visual Studio 2010 IDE, select File→New→Project to open the New Project dialog box. In the dialog box, select the Visual C#→Windows option in the Project types pane and the Windows Forms Application option in the Templates pane. Specify HashDemo as the name of the application, select an appropriate location, and click the OK button. The HashDemo application is created.
2. In the designer view of Form1, add the controls as shown in Figure 10.18.

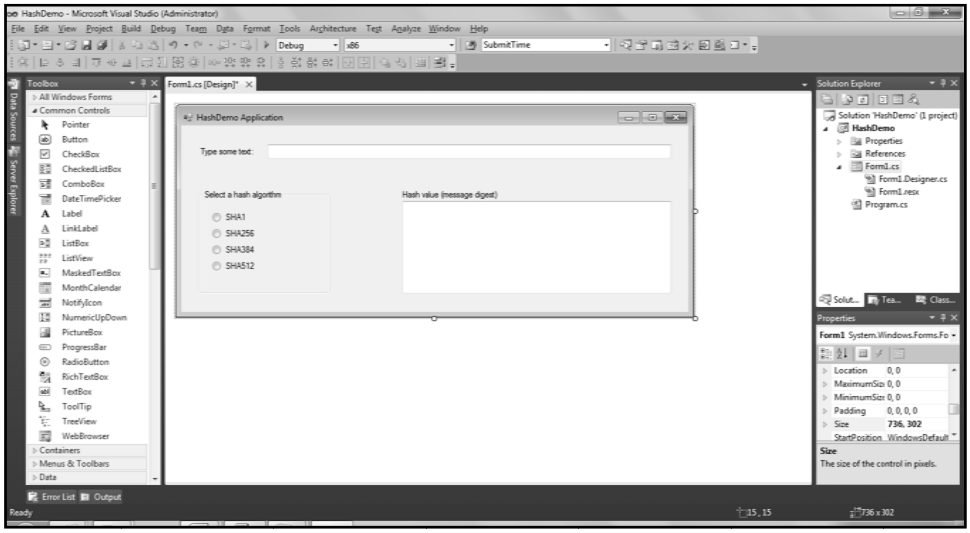

Figure 10.18: Displaying the HashDemo Application at Design Time

Now, to write the code to generate the hash values using the SHA1, SHA256, SHA384, and SHA512 hash algorithms.

3. Open the code-behind file of Form1(Form1.cs file) and add the code, given in Listing 10.8:

Listing 10.8: Showing the Code for using the SHA1, SHA256, SHA384, and SHA512 Algorithms

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;
namespace HashDemo
{
  public partial class Form1 : Form
  {
        public Form1()
        {
                InitializeComponent();
        }
        byte[] hashValue, MessageBytes;
        string StringtoConvert;
        UnicodeEncoding MyUniCodeEncoding;
        private void radioButton1_CheckedChanged(object sender, EventArgs e)
        {
                textBox2.Text = "";
                StringtoConvert = textBox1.Text;
                MyUniCodeEncoding = new UnicodeEncoding();
                MessageBytes = MyUniCodeEncoding.GetBytes(StringtoConvert);
                SHA1Managed SHhash = new SHA1Managed();
                hashValue = SHhash.ComputeHash(MessageBytes);
                foreach (byte b in hashValue)
                {
                        textBox2.Text = textBox2.Text + string.Format("{0:X2}", b);
                }
        }
        private void radioButton2_CheckedChanged(object sender, EventArgs e)
        {
                textBox2.Text = "";
                StringtoConvert = textBox1.Text;
                MyUniCodeEncoding = new UnicodeEncoding();
                MessageBytes = MyUniCodeEncoding.GetBytes(StringtoConvert);
                SHA256Managed SHhash = new SHA256Managed();
                hashValue = SHhash.ComputeHash(MessageBytes);
                foreach (byte b in hashValue)
                {
                        textBox2.Text = textBox2.Text + string.Format("{0:X2}", b);
                }
        }
        private void radioButton3_CheckedChanged(object sender, EventArgs e)
        {
                textBox2.Text = "";
                StringtoConvert = textBox1.Text;
                MyUniCodeEncoding = new UnicodeEncoding();
                MessageBytes = MyUniCodeEncoding.GetBytes(StringtoConvert);
                SHA384Managed SHhash = new SHA384Managed();
                hashValue = SHhash.ComputeHash(MessageBytes);
                foreach (byte b in hashValue)
                {
                        textBox2.Text = textBox2.Text + string.Format("{0:X2}", b);
                }
        }
        private void radioButton4_CheckedChanged(object sender, EventArgs e)
```

```
            {
                    textBox2.Text = "";
                    StringtoConvert = textBox1.Text;
                    MyUniCodeEncoding = new UnicodeEncoding();
                    MessageBytes = MyUniCodeEncoding.GetBytes(StringtoConvert);
                    SHA512Managed SHhash = new SHA512Managed();
                    hashValue = SHhash.ComputeHash(MessageBytes);
                    foreach (byte b in hashValue)
                    {
                            textBox2.Text = textBox2.Text + string.Format("{0:X2}", b);
                    }
            }
    }
}
```

In Listing 10.8, two variables named hashValue and MessageBytes are declared as arrays of bytes. The StringtoConvert variable holds the text that is entered in textBox1, which is the text that is to be hashed. In the `radioButton1_CheckedChanged()` method,the `SHA1Managed` class is used to hash the text entered in textBox1. Similarly, the SHA256Managed, SHA384Managed, and SHA512Managed classes are used in the `radioButton2_CheckedChanged()`,`radioButton3_CheckedChanged()`,and `radioButton4_CheckedChanged()` methods, respectively.

4. Run the HashDemo application and type C# 2010 Black Book in the text box next to Type some text, as shown in Figure 10.19.

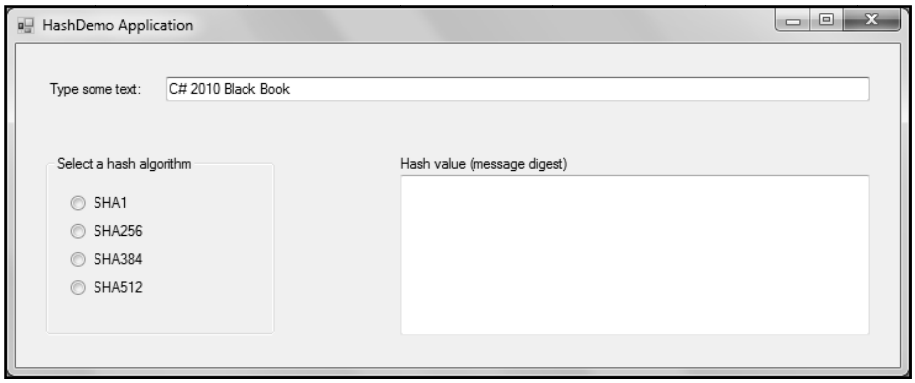

Figure 10.19: Entering the Text That Has to be Hashed

5. Select the SHA1 radio button to see the hash value of the entered text when SHA1 algorithm is used. The hash value for the entered text appears in the text box below Hash value (message digest), as shown in Figure 10.20.

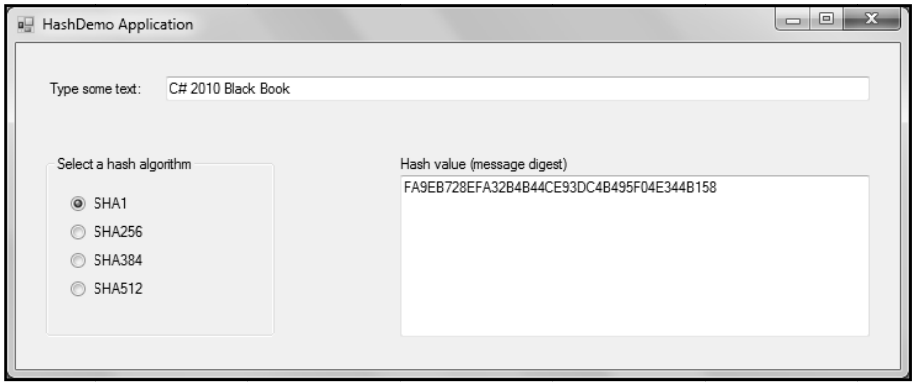

Figure 10.20: Generating Hash Value Using the SHA1 Algorithm for the Entered Text

6. Select the SHA256 radio button for using the SHA256 algorithm to hash the entered text. The hash value for the entered text appears in the text box below Hash value (message digest), as shown in Figure 10.21.

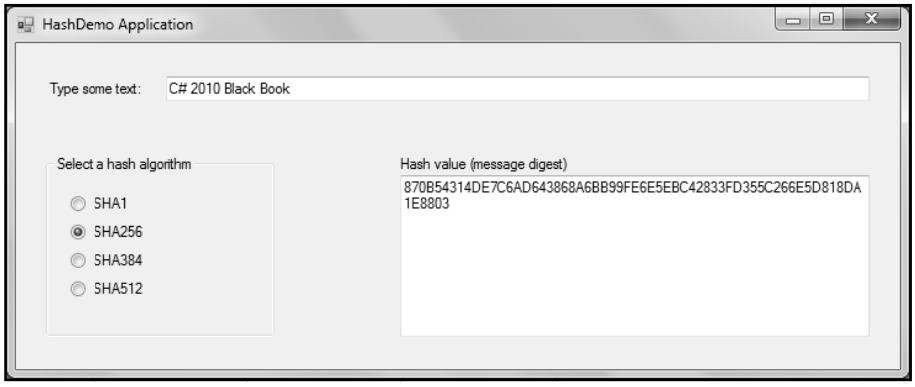

Figure 10.21: Generating Hash Value Using the SHA256 Algorithm for the Entered Text

7. Select the SHA384 radio button for using the SHA384 algorithm to hash the entered text. The hash value for the entered text appears in the text box below Hash value (message digest), as shown in Figure 10.22.

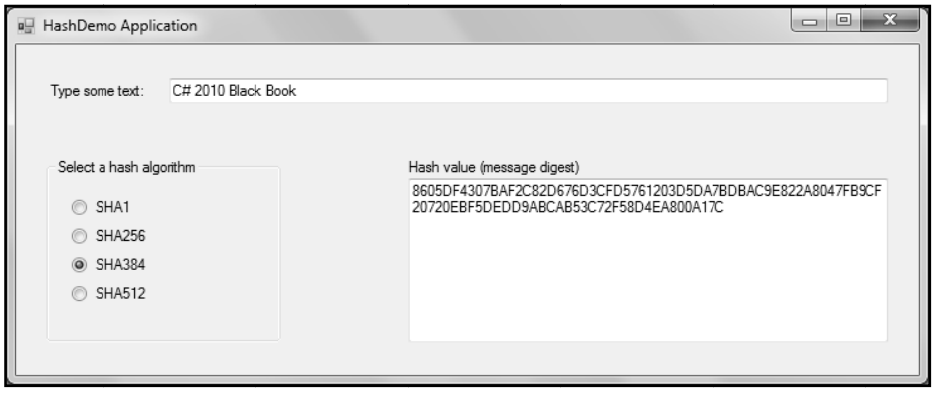

Figure 10.22: Generating Hash Value Using the SHA384 Algorithm for the Entered Text

8. Select the SHA512 radio button for using the SHA512 algorithm to hash the entered text. The hash value for the entered text appears in the text box below Hash value (message digest), as shown in Figure 10.23.

Figure 10.23: Generating Hash Value Using the SHA512 Algorithm for the Entered Text

Summary

In this chapter you learned about implementing cryptography in .NET Framework 4.0. You learned about the System.Security.Cryptographynamespace, which consists of classes that help in cryptographic services. You also learned about encryption and decryption in .NET Framework 4.0 applications by using secret-key encryption and public-key encryption. Next in the chapter, you learned about hash algorithms that are used for authenticating the sender and ensuring data integrity. The practical implementation of cryptography in .NET Framework 4.0 is discussed in the *Immediate Solutions* section of the chapter.