Module 13: Encrypting and Decrypting Data

Contents:

Module overview

It is a common requirement for applications to be able to secure information, whether it is a case of encrypting files saved to disk or web requests sent over an untrusted connection to other remote systems. The Microsoft® .NET Framework provides a variety of classes that enable you to secure your data by using encryption and hashing.

In this module, you will learn how to implement symmetric and asymmetric encryption and how to use hashes to generate mathematical representations of your data. You will also learn how to create and manage X509 certificates and how to use them in the asymmetric encryption process.

Objectives

After completing this module, you will be able to:

- Encrypt data by using symmetric encryption.
- Encrypt data by using asymmetric encryption.

Lesson 1: Implementing Symmetric Encryption

Symmetric encryption is the process of performing a cryptographic transformation of data by using a mathematical algorithm. Symmetric encryption is an established technique and is used by many applications to provide a robust way of protecting confidential data.

In this lesson, you will learn about several .NET Framework classes that enable applications to secure data by means of encryption and hashing.

Lesson objectives

After completing this lesson, you will be able to:

- Describe symmetric encryption.
- Encrypt and decrypt data by using symmetric encryption.
- Create digital fingerprints of data by using hashes.

What Is Symmetric Encryption?

- Symmetric encryption is the cryptographic transformation of data by using a mathematical algorithm
- The same key is used to encrypt and decrypt the data
- The **System.Security.Cryptography** namespace includes:
  - **DESCryptoServiceProvider** class
  - **AesManaged** class
  - **RC2CryptoServiceProvider** class
  - **RijndaelManaged** class
  - **TripleDESCryptoServiceProvider** class

The name symmetric is derived from the fact that the same secret key is used to encrypt and decrypt the data. Therefore, when you use symmetric encryption, you must keep the secret key secure.

To help improve the effectiveness of symmetric encryption, many symmetric encryption algorithms also use an initialization vector (IV) in addition to a secret key. The IV is an arbitrary block of bytes that helps to randomize the first encrypted block of data. The IV makes it much more difficult for a malicious user to decrypt your data.

Advantages and Disadvantages of Symmetric Encryption

The following table describes some of the advantages and disadvantages of symmetric encryption.

Advantage	Disadvantage
There is no limit on the amount of data you can encrypt.	The same key is used to encrypt and decrypt the data. If the key is compromised, anyone can encrypt and decrypt the data.
Symmetric algorithms are fast and consume far fewer system resources than asymmetric algorithms.	If you choose to use a different secret key for different data, you could end up with many different secret keys that you need to manage.

Symmetric algorithms are perfect for quickly encrypting large amounts of data.

Symmetric Encryption Classes in the .NET Framework

The .NET Framework contains a number of classes in the **System.Security.Cryptography** namespace, which provide managed implementations of common symmetric encryption algorithms, such as Advanced Encryption Standard (AES), Data Encryption Standard (DES), and TripleDES. Each .NET Framework symmetric encryption class is derived from the abstract **SymmetricAlgorithm** base class.

The following table describes the key characteristics of the.NET Framework encryption classes.

Algorithm	.NET Framework Class	Encryption Technique	Block Size	Key Size
DES	**DESCryptoServiceProvider**	Bit shifting and bit substitution	64 bits	64 bits
AES	**AesManaged**	Substitution-Permutation Network (SPN)	128 bits	128, 192, or 256 bits
Rivest Cipher 2 (RC2)	**RC2CryptoServiceProvider**	Feistel network	64 bit	40-128 bits (increments of 8 bits)
Rijndael	**RijndaelManaged**	SPN	128-256 bits (increments of 32 bits)	128, 192, or 256 bits
TripleDES	**TripleDESCryptoServiceProvider**	Bit shifting and bit substitution	64 bit	128-192 bits

Each of the .NET Framework encryption classes are known as block ciphers, which means that the algorithm will chunk data into fixed-length blocks and then perform a cryptographic transformation on each block.

> **Note:** You can measure the strength of an encryption algorithm by the key size. The higher the number of bits, the more difficult it is for a malicious user

trying a large number of possible secret keys to decrypt your data.

Additional Reading: For more information about **symmetric encryption in the .NET Framework,** refer to the SymmetricAlgorithm Class **page** at **https://aka.ms/moc-20483c-m13-pg1**.

Encrypting Data by Using Symmetric Encryption

To encrypt and decrypt data symmetrically, perform the following steps:

1. Create an **Rfc2898DeriveBytes** object
2. Create an **AesManaged** object
3. Generate a secret key and an IV
4. Create a stream to buffer the transformed data
5. Create a symmetric encryptor or decryptor object
6. Create a **CryptoStream** object
7. Write the transformed data to the buffer stream
8. Close the streams

You can encrypt data by using any of the symmetric encryption classes in the **System.Security.Cryptography** namespace. However, these classes only provide managed implementations of a particular encryption algorithm; for example, the **AesManaged** class provides a managed implementation of the AES algorithm. Aside from encrypting and decrypting data by using an algorithm, the encryption process typically involves the following tasks:

- Derive a secret key and an IV from a password or salt. A salt is a random collection of bits used in combination with a password to generate a secret key and an IV. A salt makes it much more difficult for a malicious user to randomly discover the secret key.
- Read and write encrypted data to and from a stream.

To help simplify the encryption logic in your applications, the .NET Framework includes a number of other cryptography classes that you can use.

The Rfc2898DeriveBytes and CryptoStream Classes

The **Rfc2898DeriveBytes** class provides an implementation of the password-based key derivation function (PBKDF2), which complies with the Public-Key Cryptography Standards (PKCS). You can use the PBKDF2 functionality to derive your secret keys and your IVs from a password and a salt.

> **Additional Reading:** For more information about **the** Rfc2898DeriveBytes class**,** refer to the Rfc2898DeriveBytes Class **page** at **https://aka.ms/moc-20483c-m13-pg2**.

The **CryptoStream** class is derived from the abstract **Stream** base class in the **System.IO** namespace, and it provides streaming functionality specific to reading and writing cryptographic transformations.

> **Additional Reading:** For more information about **the** CryptoStream class**,** refer to the CryptoStream Class **page** at **https://aka.ms/moc-20483c-m13-pg3**.

Symmetrically Encrypting and Decrypting Data

The following steps describe how to encrypt and decrypt data by using the **AesManaged** class:

1. Create an **Rfc2898DeriveBytes** object, which you will use to derive the secret key and the IV. The following code example shows how to create an instance of the **Rfc2898DeriveBytes** class, passing values for the password and salt into the constructor.

```
var password = "Pa$$w0rd";
var salt = "S@lt";
var rgb = new Rfc2898DeriveBytes(password,
Encoding.Unicode.GetBytes(salt));
```

2. Create an instance of the encryption class that you want to use to encrypt the data. The following code example shows how to create an **AesManaged** object.

```
var algorithm = new AesManaged();
```

3. Generate the secret key and the IV from the **Rfc2898DeriveBytes** object. The following code example shows how to generate the secret key and the IV by using the algorithm's **KeySize** and **BlockSize** properties.

```
var rgbKey = rgb.GetBytes(algorithm.KeySize / 8);
var rgbIV = rgb.GetBytes(algorithm.BlockSize / 8);
```

> **Note:** You typically use the algorithm's **KeySize** and **BlockSize** properties when generating the secret key and the IV, so that the secret key and the IV that you generate are compatible with the algorithm.

4. Create a stream object that you will use to buffer the encrypted or unencrypted bytes. The following code example shows how to create an instance of the **MemoryStream** class.

```
var bufferStream = new MemoryStream();
```

5. Create either a symmetric encryptor or decryptor depending on whether you want to encrypt or decrypt data. The following code example shows how to invoke the **CreateEncryptor** method to create an encryptor and how to invoke the **CreateDecryptor** method to create a decryptor. Both methods accept the secret key and the IV as parameters.

```
// Create an encryptor object.
var algorithm = algorithm.CreateEncryptor(rgbKey, rgbIV);
...
// Create a decryptor object.
var algorithm = algorithm.CreateDecryptor(rgbKey, rgbIV);
```

6. Create a **CryptoStream** object, which you will use to write the cryptographic bytes to the buffer stream. The following code example shows how to create an instance of the **CryptoStream** class, passing the **bufferStream** object, the **algorithm** object, and the stream mode as parameters.

```
var cryptoStream = new CryptoStream(
   bufferStream,
   algorithm,
   CryptoStreamMode.Write)
```

7. Invoke the **Write** and **FlushFinalBlock** methods on the **CryptoStream** object, to perform the cryptographic transform. The following code example shows how to invoke the **Write** and **FlushFinalBlock** methods of the **CryptoStream** object.

```
var bytesToTransform = FourthCoffeeDataService.GetBytes();
cryptoStream.Write(bytesToTransform, 0,
bytesToTransform.Length);
cryptoStream.FlushFinalBlock();
```

8. Invoke the **Close** method on the **CryptoStream** and the **MemoryStream** objects, so that the transformed data is flushed to the buffer stream. The following code example shows how to invoke the **Close** methods on both the **CryptoStream** and the **MemoryStream** objects.

```
cryptoStream.Close();
bufferStream.Close();
```

Hashing Data

- A hash is a numerical representation of a piece of data
- A hash can be computed by using the following code

```
public byte[] ComputeHash(byte[] dataToHash, byte[] secretKey)
{
  using (var hashAlgorithm = new HMACSHA1(secretKey))
  {
    using (var bufferStream = new MemoryStream(dataToHash))
    {
      return hashAlgorithm.ComputeHash(bufferStream);
    }
  }
}
```

Hashing is the process of generating a numerical representation of your data. Typically, hash algorithms compute hashes by mapping the binary representation of your data to the binary values of a fixed-length hash. If you use a proven hash algorithm, it is considered unlikely that you could compute the same hash from two different pieces of data. Therefore, hashes are considered a reliable way to generate a unique digital fingerprint that can help to ensure the integrity of data.

Consider the example of the FourthCoffee.Beverage service, which sends messages to the FourthCoffee.Inventory service. When the FourthCoffee.Inventory service receives a message, how do the two services know that the message was not sabotaged during the transmission? You could use hashes, as the following steps describe:

1. Compute a hash of the message before the FourthCoffee.Beverage service sends the message.

2. Compute a hash of the message when the FourthCoffee.Inventory service receives the message.

3. Compare the two hashes. If the two hashes are identical, the data has not been tampered with. If the data has been modified, the two hashes will not match.

The .NET Framework provides a number of classes in the **System.Security.Cryptography** namespace, which encapsulate common hash algorithms.

Hash Algorithms in the .NET Framework

The following table describes some of the hash classes that the .NET Framework provides.

.NET Framework Class	Description
SHA512Managed	The **SHA512Managed** class is an implementation of the Secure Hash Algorithm (SHA) and is able to compute a 512-bit hash. The .NET Framework also includes classes that implement the SHA1, SHA256, and SHA384 algorithms.
HMACSHA512	The **HMACSHA512** class uses a combination of the SHA512 hash algorithm and the Hash-Based Message Authentication Code (HMAC) to compute a 512-bit hash.
MACTripleDES	The **MACTripleDES** class uses a combination of the TripleDES encryption algorithm and a Message Authentication Code (MAC) to compute a 64-bit hash.
MD5CryptoServiceProvider	The **MD5CryptoServiceProvider** class is an implementation of the Message Digest (MD) algorithm, which uses block chaining to compute a 128-bit hash.
RIPEMD160Managed	The **RIPEMD160Managed** class is derived from the MD algorithm and is able to compute a 160-bit hash.

Computing a Hash by Using the HMACSHA512 Class

To compute a hash by using the **HMACSHA512** class, perform the following steps:

1. Generate a secret key that the hash algorithm will use to hash the data. The sender would need access to the key to generate the hash, and the receiver would need access to the key to verify the hash.

2. Create an instance of the hash algorithm.

3. Invoke the **ComputeHash** method, passing in a stream that contains the data you want to hash. The **ComputeHash** method returns a byte array that represents the hash of your data.

The following code example shows how to compute a hash by using the **HMACSHA512** class.

Hashing Data by Using the HMACSHA512 class

```
public byte[] ComputeHash(byte[] dataToHash, byte[] secretKey)
{
   using (var hashAlgorithm = new HMACSHA1(secretKey))
   {
      using (var bufferStream = new MemoryStream(dataToHash))
      {
         return hashAlgorithm.ComputeHash(bufferStream);
      }
   }
}
```

> **Additional Reading:** For more information about **hashing in the .NET Framework,** refer to the Hash Values section on the Cryptographic Services **page** at **http://go.microsoft.com/fwlink/?LinkID=267880**.

Demonstration: Encrypting and Decrypting Data

In this demonstration, you will use symmetric encryption to encrypt and decrypt a message.

Demonstration steps

You will find the steps in the **Demonstration: Encrypting and Decrypting Data** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD13_DEMO.md**.

Lesson 2: Implementing Asymmetric Encryption

Asymmetric encryption is the process of performing a cryptographic transformation of data by using an asymmetric encryption algorithm and a combination of public and private keys.

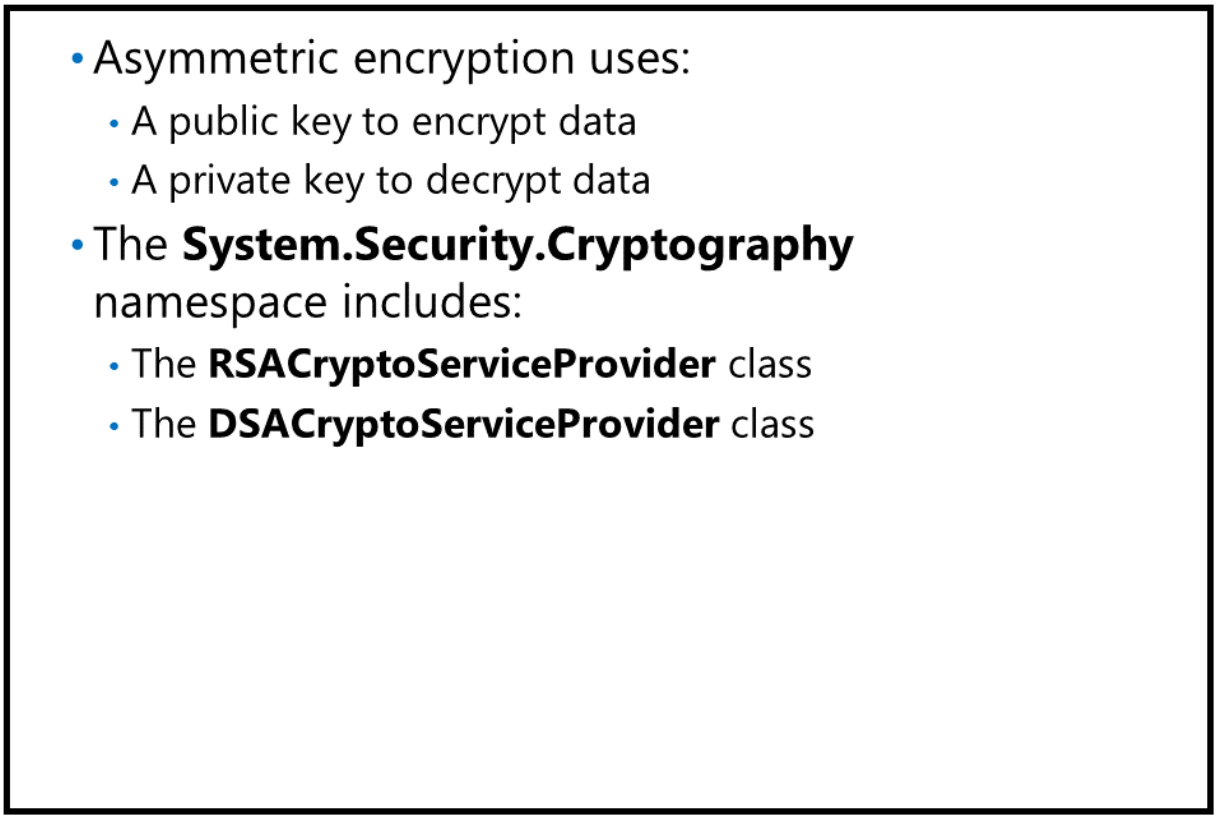
In this lesson, you will learn about the classes and tools that you can use to implement asymmetric encryption in your applications.

Lesson objectives

After completing this lesson, you will be able to:

- Describe asymmetric encryption.
- Encrypt and decrypt data by using asymmetric encryption.
- Create and manage X509 certificates.
- Manage encryption keys in your applications.

What Is Asymmetric Encryption?



Unlike symmetric encryption, where one secret key is used to perform both the encryption and the decryption, asymmetric encryption uses a public key to perform the encryption and a private key to perform the decryption.

> **Note:** The public and private keys are mathematically linked, in that the private key is used to derive the public key. However, you cannot derive a private key from a public key. Also, you can only decrypt data by using the private key that is linked to the public key that was used to encrypt the data.

In a system that uses asymmetric encryption, the public key is made available to any application that requires the ability to encrypt data. However, the private key is kept safe and is only distributed to applications that require the ability to decrypt the data. For example, HTTPS uses asymmetric encryption to encrypt and decrypt the browser's session key when establishing a secure connection between the browser and the server.

> **Note:** You can also use asymmetric algorithms to sign data. Signing is the process of generating a digital signature so that you can ensure the integrity of the data. When signing data, you use the private key to perform the signing and then use the public key to verify the data.

Advantages and Disadvantages of Asymmetric Encryption

The following table describes some of the advantages and disadvantages of asymmetric encryption.

Advantage	Disadvantage
Asymmetric encryption relies on two keys, so it is easier to distribute the keys and to enforce who can encrypt and decrypt the data.	With asymmetric encryption, there is a limit on the amount of data that you can encrypt. The limit is different for each algorithm and is typically proportional with the key size of the algorithm. For example, an **RSACryptoServiceProvider** object with a key length of 1,024 bits can only encrypt a message that is smaller than 128 bytes.
Asymmetric algorithms use larger keys than symmetric algorithms, and they are therefore less susceptible to being cracked by using brute force attacks.	Asymmetric algorithms are very slow in comparison to symmetric algorithms.

Asymmetric encryption is a powerful encryption technique, but it is not designed for encrypting large amounts of data. If you want to encrypt large amounts of data with asymmetric encryption, you should consider using a combination of asymmetric and symmetric encryption.

Best Practice: To encrypt data by using asymmetric and symmetric encryption, perform the following steps:

1. Encrypt the data by using a symmetric algorithm, such as the **AesManaged** class.
2. Encrypt the symmetric secret key by using an asymmetric algorithm.
3. Create a stream and write bytes for the following:

- The length of the IV
- The length of the encrypted secret key
- The IV
- The encrypted secret key
- The encrypted data

To decrypt, simply step through the stream extracting the data, decrypt the symmetric encryption key, and then decrypt the data.

Asymmetric Encryption Classes in the .NET Framework

The .NET Framework contains a number of classes in the **System.Security.Cryptography** namespace, which enable you to implement asymmetric encryption and signing. Each .NET Framework asymmetric class is derived from the **AsymmetricAlgorithm** base class.

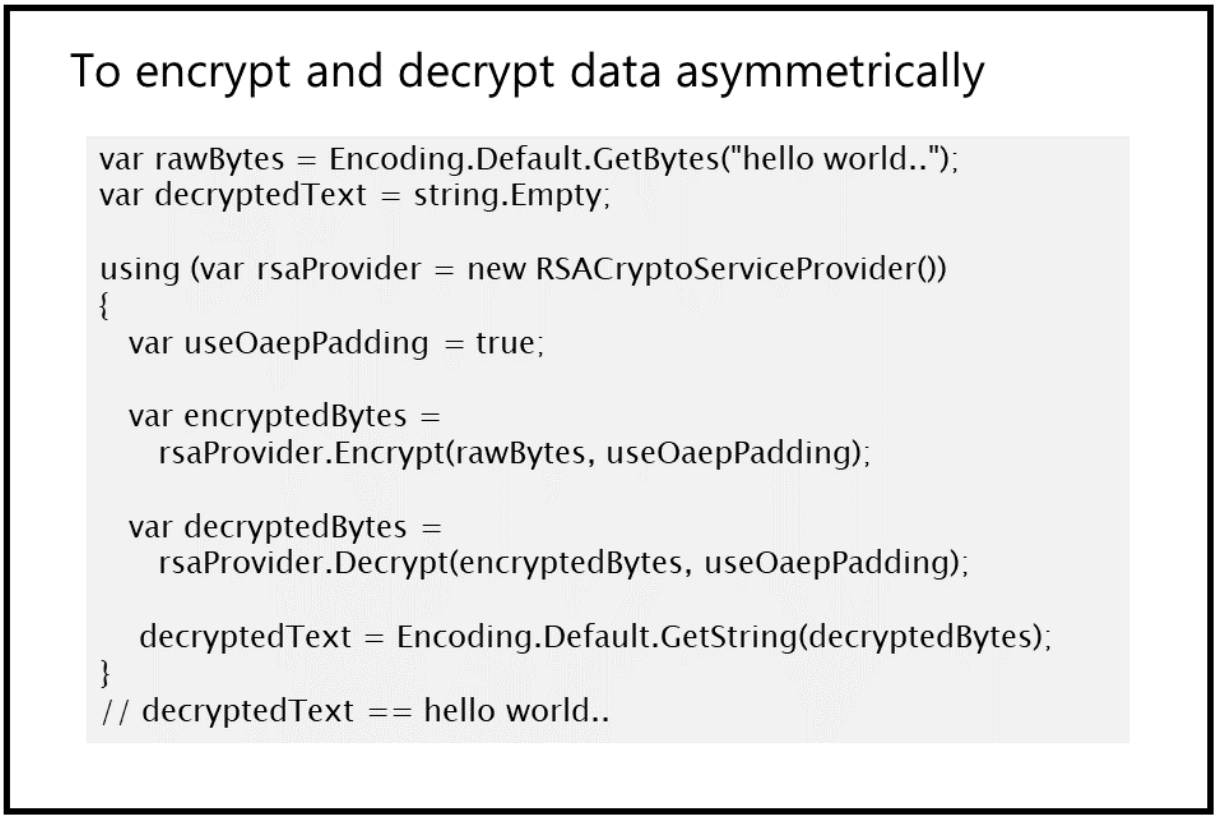
The following list describes some of these classes:

- **RSACryptoServiceProvider**. This class provides an implementation of the RSA algorithm, which is named after its creators, Ron Rivest, Adi Shamir, and Leonard Adleman. By default, the **RSACryptoServiceProvider** class supports key lengths ranging from 384 to 512 bits in 8-bit increments, but optionally, if you have the Microsoft Enhanced Cryptographic Provider installed, the **RSACryptoServiceProvider** class will support keys up to 16,384 bits in length. You can use the **RSACryptoServiceProvider** class to perform both encryption and signing.
- **DSACryptoServiceProvider**. This class provides an implementation of the Digital Signature Algorithm (DSA) algorithm and supports keys ranging from 512 to 1,024 bits in 64-bit increments. Although the **RSACryptoServiceProvider** class supports both encryption and signing, the **DSACryptoServiceProvider** class only supports signing.

.

Additional Reading: For more information about **asymmetric encryption in the .NET Framework,** refer to the Public-Key Encryption section on the Cryptographic Services **page** at **http://go.microsoft.com/fwlink/?LinkID=267881**.

Encrypting Data by Using Asymmetric Encryption



You can encrypt your data asymmetrically by using the **RSACryptoServiceProvider** class in the **System.Security.Cryptography** namespace.

Encrypting Data by Using the RSACryptoServiceProvider Class

The **RSACryptoServiceProvider** class provides a number of members that enable you to implement asymmetric encryption functionality in your applications, including the ability to import and export key information and encrypt and decrypt data.

You can create an instance of the **RSACryptoServiceProvider** class by using the default constructor. If you choose this approach, the **RSACryptoServiceProvider** class will generate a set of public and private keys.

The following code example shows how to create an instance of the **RSACryptoServiceProvider** class by using the default constructor.

Instantiating the RSACryptoServiceProvider Class

```
var rsaProvider = new RSACryptoServiceProvider();
```

After you have created an instance of the **RSACryptoServiceProvider** class, you can then use the **Encrypt** and **Decrypt** methods to protect your data.

The following code example shows how you can use the **Encrypt** and **Decrypt** methods to protect the contents of a string variable.

Encrypting and Decrypting Data by Using the RSACryptoServiceProvider Class

```
var plainText = "hello world…";
var rawBytes = Encoding.Default.GetBytes(plainText);
var decryptedText = string.Empty;
using (var rsaProvider = new RSACryptoServiceProvider())
{
    var useOaepPadding = true;
    var encryptedBytes = rsaProvider.Encrypt(rawBytes,
useOaepPadding);
    var decryptedBytes = rsaProvider.Decrypt(encryptedBytes,
useOaepPadding);
    decryptedText = Encoding.Default.GetString(decryptedBytes);
}
// The decryptedText variable will now contain " hello world…"
```

> **Note:** You use the *useOaepPadding* parameter to determine whether the **Encrypt** and **Decrypt** methods use Optimal Asymmetric Encryption Padding (OAEP). If you pass **true**, the methods use OAEP, and if you pass **false**, the methods use PKCS#1 v1.5 padding.

Typically, applications do not encrypt and decrypt data in the scope of the same **RSACryptoServiceProvider** object. One application may perform the encryption, and then another performs the decryption. If you attempt to use different **RSACryptoServiceProvider** objects to perform the encryption and decryption, without sharing the keys, the **Decrypt** method will throw a **CryptographicException** exception. The **RSACryptoServiceProvider** class exposes members that enable you to export and import the public and private keys.

The following code example shows how to instantiate different **RSACryptoServiceProvider** objects and use the **ExportCspBlob** and **ImportCspBlob** methods to share the public and private keys.

Importing and Exporting Keys

```
var keys = default(byte[]);
var exportPrivateKey = true;
using (var rsaProvider = new RSACryptoServiceProvider())
{
    keys = rsaProvider.ExportCspBlob(exportPrivateKey);
    // Code to perform encryption.
}
var decryptedText = string.Empty;
using (var rsaProvider = new RSACryptoServiceProvider())
{
    rsaProvider.ImportCspBlob(keys);
    // Code to perform decryption.
}
```

> **Note:** The *exportPrivateKey* parameter instructs the **ExportCspBlob** method to include the private key in the return value. If you pass **false** into the **ExportCspBlob** method, the return value will not contain the private key. If you try to decrypt data without a private key, the Common Language Runtime (CLR) will throw a **CryptographicException** exception.

Instead of maintaining and persisting keys in your application, you can use the public and private keys in an X509 certificate, stored in the certificate store on the computer that is running your application.

> **Additional Reading:** For more information about **the** RSACryptoServiceProvider **class,** refer to the **RSACryptoServiceProvider Classpage** at **https://aka.ms/moc-20483c-m13-pg4**.

Creating and Managing X509 Certificates

An X509 certificate is a digital document that contains information, such as the name of the organization that is supplying the data. X509 certificates are normally stored in certificate stores. In a typical Windows installation, there are user account, service account, and local computer machine certificate stores.

X509 certificates can also contain public and private keys, which you can use in the asymmetric encryption process. You can create and manage your X509 certificates by using the tools that Windows and the .NET Framework provide.

Creating a Certificate by Using MakeCert

MakeCert is a certificate creation tool that the .NET Framework provides. You can access the tool by using the Visual Studio command prompt. The MakeCert tool provides a number of command-line switches that enable you to configure the X509 certificate to meet the requirements of your application.

The following table describes some of the MakeCert command-line switches.

Switch	Description

Switch	Description
-n	This enables you to specify the name of the certificate.
-a	This enables you to specify the algorithm that the certificate uses.
-pe	This enables you to create a certificate that allows you to export the private key.
-r	This enables you to create a self-signed certificate.
-sr	This enables you to specify the name of the certificate store where the MakeCert tool will import the generated certificate.
-ss	This enables you to specify the name of the container in the certificate store where the MakeCert tool will import the generated certificate.
-sky	This enables you to specify the type of key that the certificate will contain.

The following code example shows how you can use the MakeCert command-line tool to generate a self-signed certificate, which contains both a public and a private key.

MakeCert Example

```
makecert -n "CN=FourthCoffee" -a sha1 -pe -r -sr LocalMachine -ss my
-sky exchange
```

> **Additional Reading:** For more information about **MakeCert,** refer to the **Makecert.exe (Certificate Creation Tool) page** at **https://aka.ms/moc-20483c-m13-pg5**

Managing X509 Certificates by Using the Microsoft Management Console Certificates Snap-in

The Microsoft Management Console (MMC) Certificates snap-in enables you to manage any X509 certificates installed in the context of your user account, service account, or local computer.

To open a certificate store by using the MMC snap-in, perform the following steps:

1. Log on as an administrator. If you log on without administrative privileges, you will only be able to view certificates in your user account certificate store.
2. In the Windows 8 **Start** window, use search to find mmc.exe, and then click **mmc.exe**.
3. In the MMC window, on the **File** menu, click **Add/Remove Snap-in**.
4. In the **Add or Remove Snap-ins** dialog box, in the **Available snap-ins** list, click **Certificates**, and then click **Add**.
5. In the **Certificates snap-in** dialog box, click either **My user account**, **Service account**, or **Computer account**, and then perform one of the following steps:
    a. If you chose **My user account**, click **Finish**.
    b. If you chose **Service account**, click **Next**, and then perform the following steps:
        i. In the **Select Computer** dialog box, click **Next**.
        ii. In the **Certificates snap-in** dialog box, in the **Service account** list, click the service account you want to manage, and then click **Finish**.
    c. If you chose **Computer account**, click **Next**, and then click **Finish**.
6. In the **Add or Remove Snap-ins** dialog box, repeat steps 4 and 5 if you want to add additional certificate stores to your session, and then click **OK**.

After you have opened one or more certificate stores, you can perform any of the following tasks:

- View the properties that are associated with any X509 certificate in any of the certificate stores, such as Personal or Trusted Root Certificate Authorities stores.
- Export an X509 certificate from a certificate store to the file system.
- Manage the private keys that are associated with an X509 certificate.
- Issue a request to renew an existing X509 certificate.

Additional Reading: For more information about managing certificates, refer to the **Working with Certificates page** at **http://go.microsoft.com/fwlink/?LinkID=267884**.

Managing Encryption Keys

The **System.Security.Cryptography.X509Certificates** namespace contains classes that enable access to the certificate store and certificate metadata

```
var store = new X509Store(
  StoreName.My,
  StoreLocation.LocalMachine);

store.Open(OpenFlags.ReadOnly);

foreach (var storeCertificate in store.Certificates)
{
  // Code to process each certificate.
}

store.Close();
```

The .NET Framework provides the **System.Security.Cryptography.X509Certificates** namespace, which contains a number of classes that enable you to use X509 certificates and their keys in your applications. These classes include the following:

- **X509Store**. This class enables you to access a certificate store and perform operations, such as finding an X509 certificate with a particular name.
- **X509Certificate2**. This class enables you create an in-memory representation of an X509 certificate that currently exists in a certificate store. When you have instantiated an **X509Certificate2** object, you can then use its members to access information, such as the X509 certificate's public and private keys.
- **PublicKey**. This class enables you to manipulate the various pieces of metadata that are associated with an X509 certificate's public key, such as the public key's value.

The following code example shows how to use the **X509Store** and **X509Certificate2** classes to enumerate the personal certificate store on the local machine.

Enumerating a Certificate Store

```
var store = new X509Store(StoreName.My, StoreLocation.LocalMachine);
var certificate = default(X509Certificate2);
var certificateName = "CN=FourthCoffee";
store.Open(OpenFlags.ReadOnly);
foreach (var storeCertificate in store.Certificates)
{
   if (storeCertificate.SubjectName.Name == certificateName)
   {
      certificate = storeCertificate;
      continue;
   }
}
store.Close();
```

After you have created an **X509Certificate2** object, you can use its members to determine whether the X509 certificate contains either a public or private key. The following list describes some of the members you can use:

- **HasPrivateKey**. This property enables you to determine whether the X509 certificate contains a private key.
- **FriendlyName**. This property enables you to get the friendly name that is associated with the X509 certificate.
- **GetPublicKeyString**. This method enables you to extract the public key that the X509 certificate contains as a string value.
- **PublicKey**. This property enables you to get the public key that the X509 certificate contains as a **PublicKey** object.

- **PrivateKey**. This property enables you to get the private key that the X509 certificate contains as an **AsymmetricAlgorithm** object.

You can use the **PublicKey** and **PrivateKey** properties of the **X509Certificate2** class to create an instance of the **RSACryptoServiceProvider** class.

The following code example shows how you can use the **PublicKey** and **PrivateKey** properties to create an instance of the **RSACryptoServiceProvider** class..

Instantiating the RSACryptoServiceProvider Class

```
var certificate = new X509Certificate2();
// Code to set the public and private keys.
// Create an RSA encryptor.
var rsaEncryptorProvider =
(RSACryptoServiceProvider)certificate.PublicKey.Key;
// Create an RSA decryptor.
var rsaDecryptorProvider =
(RSACryptoServiceProvider)certificate.PrivateKey;
```

> **Additional Reading:** For more information about managing encryption keys in your application, refer to the **System.Security.Cryptography.X509Certificates Namespace page** at **https://aka.ms/moc-20483c-m13-pg6**

Demonstration: Encrypting and Decrypting Grade Reports Lab

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Demonstration steps

You will find the steps in the **Demonstration: Encrypting and Decrypting Grade Reports Lab** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD13_DEMO.md**.

Lab: Encrypting and Decrypting the Grades Report

Scenario

You have been asked to update the Grades application to ensure that reports are secure when they are stored on a user's computer. You decide to use asymmetric encryption to protect the report as it is generated, before it is written to disk. Administrative staff will need to merge reports for each class into one document, so you decide to develop a separate application that generates a combined report and prints it.

Objectives

After completing this lab, you will be able to:

- Encrypt data by using asymmetric encryption.
- Decrypt data.

Lab setup

Estimated Time: 60 minutes

You will find the high-level steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD13_LAB_MANUAL.md**._

You will find the detailed steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD13_LAK.md**._

Exercise 1: Encrypting the Grades Report

Scenario

In this exercise, you will update the reporting functionality to encrypt the report as it is generated, but before it is saved to disk.

First, you will create an asymmetric certificate by using a prewritten batch file. The batch file uses the MakeCert tool that ships with the Windows Software Development Kit (SDK). You will create a self-signed certificate named Grades using the SHA-1 hash algorithm and store it in the LocalMachine certificate store. You will then write code in the Grades application to retrieve the certificate by looping through the certificates in the LocalMachine store and checking the name of the certificate against the name that is stored in the App.Config file. Next, you will use the classes that are provided in the **System.Security.Cryptography** and **System.Security.Cryptography.X509Certificates** namespaces to write the **EncryptWithX509** method in the **Grades.Utilities.WordWrapper** class. You will get the public key from the certificate that you created and use it to create an instance of the **RSAPKCS1KeyExchangeFormatter** class. You will use this to encrypt the data for the report and then return the encrypted buffered data to the calling method as a byte array. You will then write code in the **EncryptAndSaveToDisk** method to write the returned data to the file that the user specifies. Finally, you will build and test the application and verify that the reports are now encrypted.

> **Result**: After completing this exercise, you should have updated the Grades application to encrypt generated reports.

Exercise 2: Decrypting the Grades Report

Scenario

In this exercise, you will create a separate utility to enable users to print reports. Users will be able to select a folder that contains encrypted reports, and the application will then generate one combined report and send it to the default printer.

First, you will use the classes that are provided in the **System.Security.Cryptography** and

System.Security.Cryptography.X509Certificates namespaces to write the **DecryptWithX509** method in the **SchoolReports.WordWrapper** class. You will get the private key from the certificate and use it to create an instance of the **RSACryptoServiceProvider** class. You will use this to decrypt the data from the individual reports and then return the decrypted data to the calling method as a byte array. Finally, you will build and test the application and verify that a printed version of the composite report has been generated.

Result: After completing this exercise, you should have a composite unencrypted report that was generated from the encrypted reports.

Module review and takeaways

In this module, you learned how to implement symmetric and asymmetric encryption and how to use hashes to generate mathematical representations of your data.

Review Question(s)

Check Your Knowledge

Select the best answer

Fourth Coffee wants you to implement an encryption utility that can encrypt and decrypt large image files. Each image will be more than 200 megabytes (MB) in size. Fourth Coffee envisages that only a small internal team will use this tool, so controlling who can encrypt and decrypt the data is not a concern. Which of the following techniques will you choose?

- Symmetric encryption
- Asymmetric encryption
- Hashing

Check answer Show solution Reset

Check Your Knowledge

True or False Question

Is the following statement true or false? Asymmetric encryption uses a public key to encrypt data.

True False

Check answer Reset