Session: 11

Security

Objectives

- Define and describe how to secure applications
- Define and describe different types of malicious attacks
- Explain how to protect a Web application against malicious attacks
- Explain the process of securing application data

Malicious Attacks and their Preventions

- Web applications are deployed so that it can be accessed by different types of users.
- Some of these users might plan to carry out malicious attacks against these applications.
- Irrespective of the motive behind malicious attacks, while developing an application, you should ensure that a proper security mechanism is used to protect it from such attacks.
- Some of the common malicious attacks are:
  - Cross-site Scripting (XSS)
  - Cross-site Request Forgery (CSRF)
  - SQL Injection

Cross-site Scripting 1-5

- An XSS attack can be of the following two type:
  - **Persistent attack**: In this type of attack, the harmful code is stored in the database.
  - **Non-persistent attack**: In this type of attack, malicious code is not stored in the database; instead this code is rendered on the browser itself.
- To prevent XSS attacks, you should ensure that all HTML code that an application accepts from a user is encoded.
- HTML encoding is a process that converts potentially unsafe characters to their HTML-encoded equivalent.
- As a result, when you encode HTML inputs submitted to your application, the HTML engine does not interpret these characters as parts of the HTML markup and renders them as strings.

Cross-site Scripting 2-5

- Apart from this, you also require ensuring that all HTML outputs of the application are encoded to prevent XSS attacks.
- Consider a scenario, where a user has gained access to the database of your application and inserted a script in a table.
- Following code snippet shows the script inserted in a table:

Snippet

```
<script>alert('Visit the gethacked website')</script>
```

- This code displays an alert box on the browser when the application accesses the script and sends it to the browser as response.

Cross-site Scripting 3-5

- In such scenario, you need to encode the HTML output to prevent such attacks.
- Following code snippet shows encoding HTML output that converts the preceding markup:

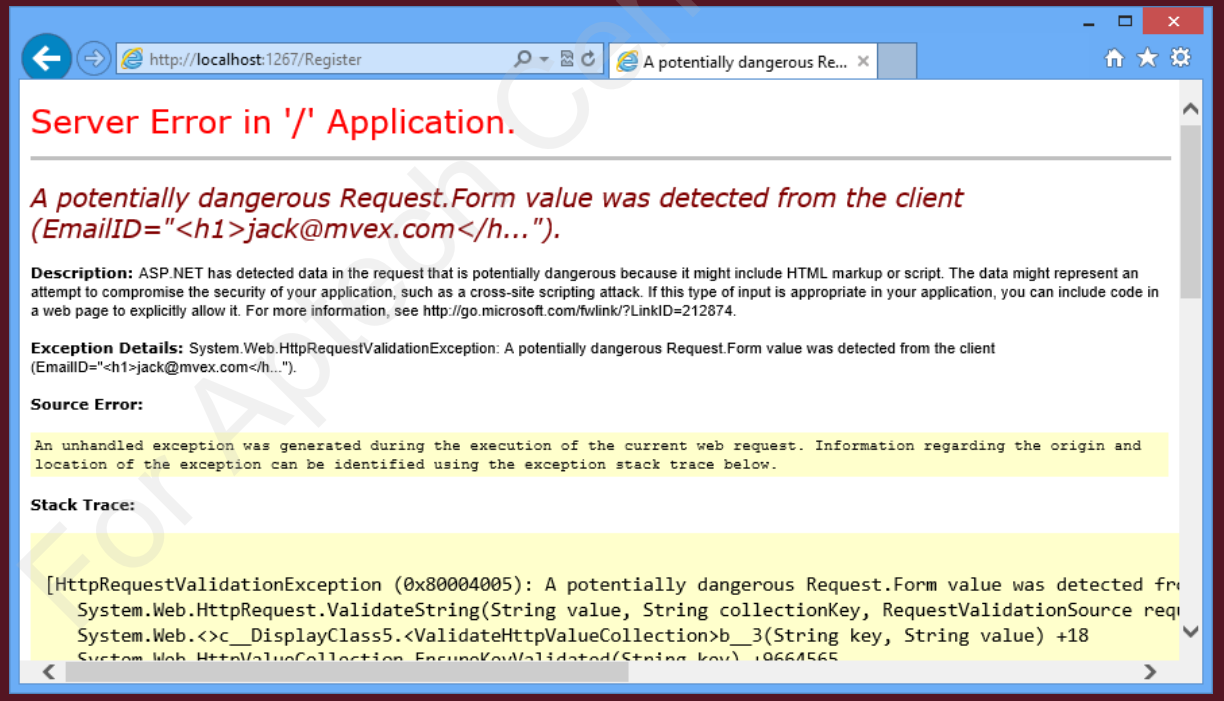
Snippet

```
&lt;script&gt;alert(&#44;Visit the gethacked
website&#44;)&lt;/script&gt;
```

- This code will be displayed by the browser instead of executing the JavaScript code.

Cross-site Scripting 4-5

- The ASP.NET MVC Framework provides a request validation feature that examines an HTTP request to check and prevent potentially malicious content.
- Following figure shows an error message that the browser displays once a form is submitted that contains HTML or JavaScript code to carry out an XSS attack:



Cross-site Scripting 5-5

- Another built-in feature that you can use in an ASP.NET MVC application to prevent XSS attack is, the Razor HTML encoding of the Razor view engine.
- When you use the @ symbol in a view to refer any data, the Razor HTML encoding feature automatically encodes the data and makes it safe to display as HTML on the browser.

Cross-site Request Forgery 1-5

- There are two approaches that you can use to block CSRF attacks, such as domain referrer and user-generated token.
- The domain referrer approach:
  - Checks whether or not an incoming request has a referrer header for your domain.
  - Enables you to ensure that the request has only been initiated from your application.
  - Prevents any requests coming from unknown sources.
- While developing an application, you can use the domain referrer approach by checking whether a user that posts data through a form is only from your application.
- You can use the Request.Url.Host property in an action method to retrieve the host name of your application.

Cross-site Request Forgery 2-5

- You can retrieve the host name of the referrer by using the Request.UrlReferrer.Host property.
- Following code snippet shows comparison of both the host names and throws an exception if they do not m

Snippet

```
 string referrerHostName = Request.UrlReferrer.Host;
string appHostName = Request.Url.Host;
if (appHostName != referrerHostName)
{
    throw new UnauthorizedAccessException();
}
```

- In this code:
  - The host name of the application and the referrer is retrieved and compared.
  - If both the host names do not match, then the UnauthorizedAccessException exception is thrown.

Cross-site Request Forgery 3-5

- You can also use the user-generated token approach to prevent a CSRF attack.
- This approach enables storing a user-generated token using a HTML hidden field in the user session.
- Thereafter, for each incoming request from a user, you can verify whether or not the submitted token is valid.
- ASP.NET MVC Framework provides a set of helpers that you can use to detect and block any CSRF attacks.
- This can be done by creating a user-generated token, which is passed between the view and the controller and verified on each request.
- One of such helper is the @Html.AntiForgeryToken() method that allows adding a hidden HTML field in a page that the controller will verify at each request.

Cross-site Request Forgery 4-5

- Following code snippet shows how to add the @Html.AntiForgeryToken() to a form in a view:

Snippet

```
<form action="/user/register" method="post">
@Html.AntiForgeryToken()
…
</form>
```

- This code will generate an encrypted value for a hidden input and sends back a cookie named, _RequestVerificationToken, with the same encrypted value.
- Following code snippet shows the generated encrypted value for a hidden input:

Snippet

```
<input type="hidden" value="012837udny31w90hjhf7u">
```

- In addition, you need to add the [ValidateAntiforgeryToken] filter to the action method that processes the posted form.

Cross-site Request Forgery 5-5

- Following code snippet shows using the [ValidateAntiforgeryToken] filter:

Snippet

```
[ValidateAntiForgeryToken]
public ActionResult RegisterUser(RegisterUserModel model)
{
/*Code to register and return an ActionResult object*/
}
```

- In this code:
  - The [ValidateAntiforgeryToken] filter checks for the _RequestVerificationToken cookie.
  - The _RequestVerificationToken form value when a form is posted.
  - If both the values are present, the filter matches.
  - A successful match results in the execution of the RegisterUser action. Else, the filter throws an exception.

SQL Injection 1-3

- SQL injection is a form of attack in which a user posts SQL code to an application.
- It creates an SQL statement that will execute with malicious intent.
- Using a SQL injection, a user can store and update malicious data, access sensitive data, or delete any existing data of the application.
- Following code snippet shows a dynamic SQL query that accesses data from the User table based on a userName form value:

Snippet

```
String UserName= context.Request.Form["userName"];
String Query = "select * from User where User_name ='"+ UserName +"'";
```

SQL Injection 2-3

- In the code:
  - The variable named, UserName stores the value of the userName field of a submitted form.
  - The Query variable constructs a dynamic SQL query by adding the UserName variable to retrieve those records from the User table whose User_Name field matches the value of the UserName variable.
  - When a user submits a user name, this code will execute and return all data from the User table that matches the user name.
- A user who identifies the vulnerabilities for SQL injection in the code, can carry out an attack by submitting a query in the User name text field of the form.

SQL Injection 3-3

- Following code snippet shows submitting a query in the User name text field of a form:

Snippet

```
drop table User;
```

- On submitting the form, following query will be generated dynamically:

Snippet

```
select * from User where User_name = ''; drop table User --'
```

- This code will execute in the following two phases:
  - In the first phase, the query will attempt to retrieve records from the User table.
  - Then, in the next phase, the query will attempt to drop the User table. The -- symbol after the drop statement in the query specifies that the remaining part of the query does not required to be executed.

Deferred Validation and Unvalidated Requests 1-4

- In an ASP.NET MVC application, you can enable deferred validation by configuring it in the Web.config file of the application.
- For this, you need to set the **requestValidationMode** attribute of the <httpRuntime> element to 4.5.
- Following code snippet allows setting the requestValidationMode attribute of the <httpRuntime> element:

Snippet

```
<httpruntime requestvalidationmode="4.5" />
```

- Once you configure deferred validation, you can validate data of a particular field in a form.
- You can do this when you expect a field that can contain HTML code or scripts that the request validation process checks for potential XSS attack.

Deferred Validation and Unvalidated Requests 2-4

- One of the approaches to instruct the request validation process not to validate the data of a particular field is by using the Unvalidated property of the HttpRequest class.
- Following code snippet shows using the Unvalidated property to access the content of the code field:

Snippet

```
var c = context.Request.Unvalidated.Form["code"];
```

- Apart from this, you can also instruct the request validation process not to perform validation at the controller level or at action method level.
- You can do this by setting the value of the ValidateInput attribute to false.

Deferred Validation and Unvalidated Requests 3-4

- Following code snippet shows disabling the validation of form data processed by the SubmitQuery() action method:

Snippet

```
[HttpPost]
[ValidateInput(false)]
public ActionResult SubmitQuery(QueryModel model)
```

- In this code, the value of the ValidateInput attribute is set to false to disable the validation of form data processed by the SubmitQuery() action method.
- You can also disable validation for a field for a strongly-typed view by adding the AllowHtml attribute to the corresponding property of the model.

Deferred Validation and Unvalidated Requests 4-4

- Following code snippet shows using the AllowHtml attribute:

Snippet

```
[AllowHtml]
[Required]
[Display(Name = "Code Snippet")]
public string CodeSnippet{ get; set; }
)
```

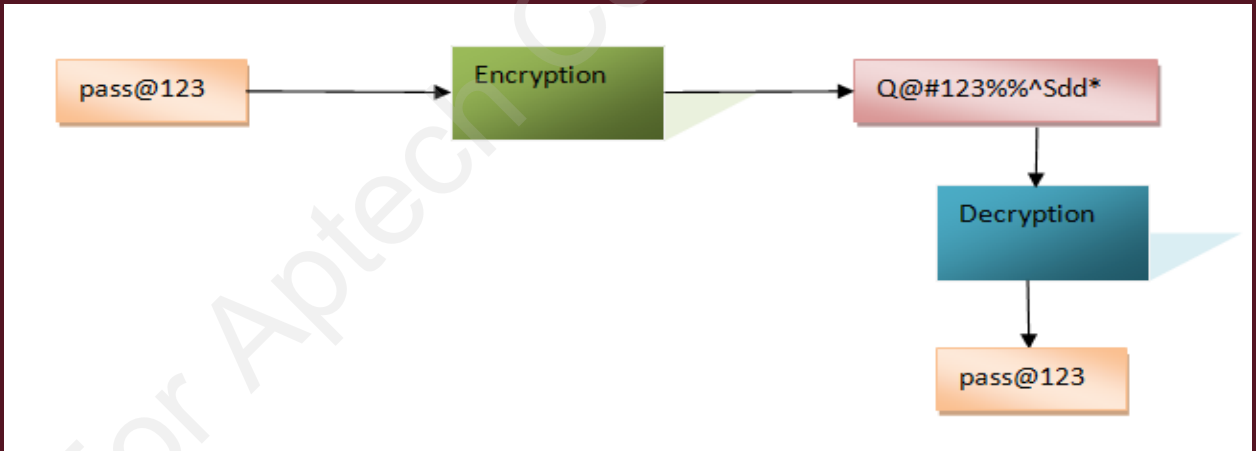
- In this code:
  - The AllowHtml attribute is applied to the CodeSnippet property.
  - As a result, when the user adds HTML markups and script in the CodeSnippet field and submits the form, the request validation process will not report an error, as it will do for other fields.

Data Security

- All organizations need to handle sensitive data.
- This data can be either present in storage or may be exchanged between different entities within and outside the organization over a network.
- To avoid misuse of such data, there should be some security mechanism that can ensure confidentiality and integrity of the data.
- One of the commonly used techniques to secure such sensitive data is known as encryption.

Encryption and Decryption

- Encryption:
  - Is a technique that ensures data confidentiality.
  - Converts data in plain text to cipher (secretly coded) text.
- Decryption is a process that converts the encrypted cipher text back to the original plain text.
- Following figure shows the process of encryption and decryption of a password as an example:



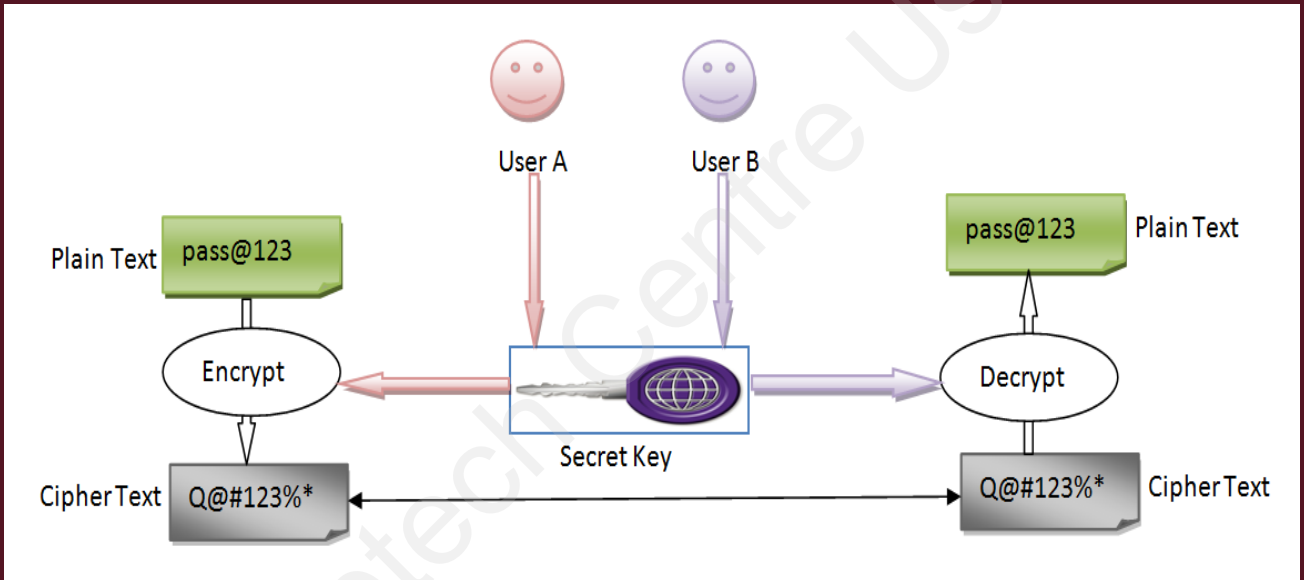
- In this figure, the plain text, pass@123 is encrypted to a cipher text. The cipher text is decrypted back to the original plain text.

Types of Encryption and Decryption 1-4

- There are two types of encryption and decryption, such as symmetric and asymmetric.
- Symmetric encryption, or secret key encryption, uses a single key, known as the secret key both to encrypt and decrypt data.
- Following steps outline an example usage of symmetric encryption:
  - User A uses a secret key to encrypt a plain text to cipher text.
  - User A shares the cipher text and the secret key with User B.
  - User B uses the secret key to decrypt the cipher text back to the original plain text.

Types of Encryption and Decryption 2-4

- Following figure shows the symmetric encryption and decryption process:

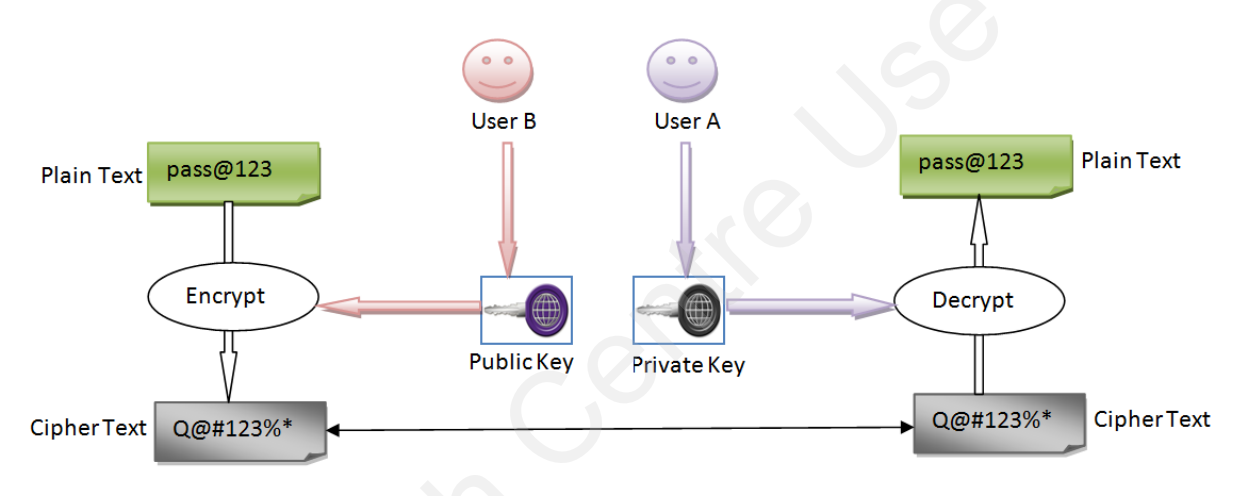


Types of Encryption and Decryption 3-4

- The asymmetric encryption uses a pair of public and private key to encrypt and decrypt data.
- Following steps outline an example usage of asymmetric encryption:
  - User A generates a public and private key pair.
  - User A shares the public key with User B.
  - User B uses the public key to encrypt a plain text to cipher text.
  - User A uses the private key to decrypt the cipher text back to the original plain text.

Types of Encryption and Decryption 4-4

- Following figure shows the symmetric encryption and decryption process:



Using Symmetric Encryption 1-13

- You need to use one of the symmetric encryption implementation classes of the .NET Framework to perform symmetric encryption.
- The first step to perform symmetric encryption is to create the symmetric key.
- When you use the default constructor of the symmetric encryption classes, such as RijndaelManaged and AesManaged, a key and IV is automatically generated.
- The generated key and the IV can be accessed as byte arrays using the Key and IV properties of the encryption class.

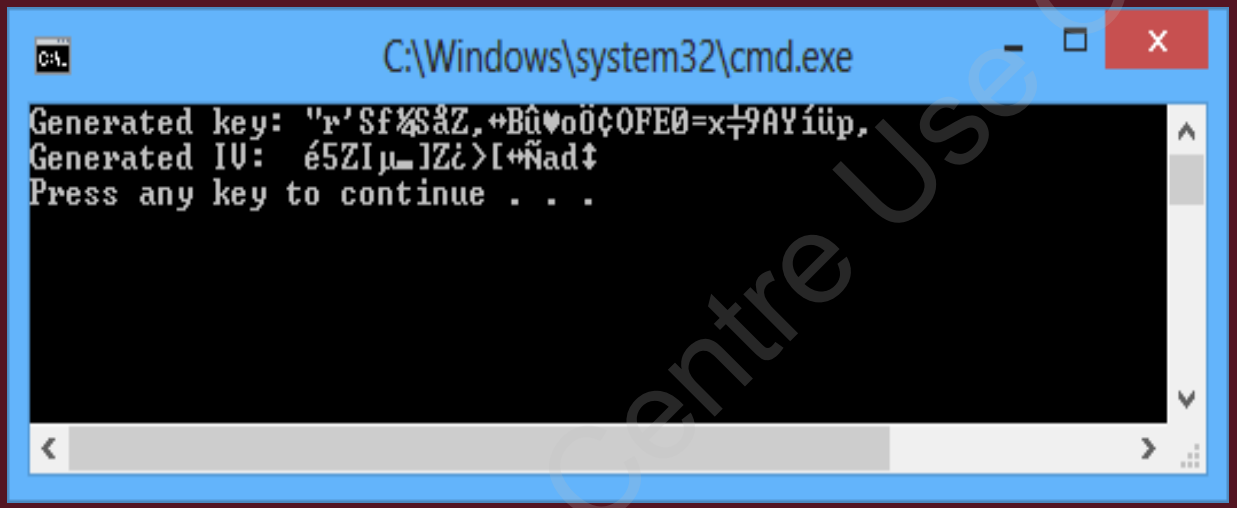
- Following code snippet shows creating a symmetric key and IV using the RijndaelManaged class:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
. . .
. . .
RijndaelManaged symAlgo = new RijndaelManaged();
Console.WriteLine("Generated key: {0}, \nGenerated IV: {1}",
Encoding.Default.GetString(symAlgo.Key),
Encoding.Default.GetString(symAlgo.IV));
```

- This code uses the default constructor of the RijndaelManaged class to generate a symmetric key and IV.
- The Key and IV properties are accessed and printed as strings using the default encoding to the console.

- Following figure shows the output of the preceding code:



- The symmetric encryption classes, such as RijndaelManaged also provide the GenerateKey() and GenerateIV() methods that you can use to generate keys and IVs.

- Following code snippet shows using the GenerateKey() and GenerateIV() methods to generate keys and IVs:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
. . .
. . .
RijndaelManagedsymAlgo = new RijndaelManaged();
       RijndaelManagedsymAlgo.GenerateKey();
       RijndaelManagedsymAlgo.GenerateIV();
       byte[] generatedKey = RijndaelManagedsymAlgo.Key;
       byte[] generatedIV = RijndaelManagedsymAlgo.IV;
       Console.WriteLine("Generated key through GenerateKey(): {0},
\nGenerated IV through GenerateIV():  {1}",
       Encoding.Default.GetString(generatedKey),
       Encoding.Default.GetString(generatedIV));
```

- This code creates a RijndaelManaged object and then, calls the GenerateKey() and GenerateIV() methods to generate a key and an IV. The Key and the IV properties are then, accessed and printed as strings using the default encoding to the console.

Using Symmetric Encryption 5-13

- The symmetric encryption classes of the ASP.NET MVC Framework provides the CreateEncryptor() method that returns an object of the ICryptoTransform interface.
- The ICryptoTransform object is responsible for transforming the data based on the algorithm of the encryption class.
- Once you have obtained an ICryptoTransform object, you can use the CryptoStream class to perform the encryption.
- The CryptoStream class acts as a wrapper of a stream-derived class, such as FileStream, MemoryStream, and NetworkStream.
- A CryptoStream object operates in one of the following two modes defined by the CryptoStreamMode enumeration:
  - First is Write mode that allows writing operation on the underlying stream, and you can use this mode to perform encryption.
  - Second is the Read mode that allows reading operation on the underlying stream.

Using Symmetric Encryption 6-13

- You can create a CryptoStream object by calling the constructor that accepts the following three parameters:
  - The underlying stream object
  - The ICryptoTransform object
  - The mode defined by the CryptoStreamMode enumeration
- After creating the CryptoStream object, you can call the Write() method to write the encrypted data to the underlying stream.
- Following code snippet encrypts data using the RijndaelManaged class and writes the encrypted data to a file:

Snippet

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class SymmetricEncryptionDemo
{
    static void EncryptData(String plainText, RijndaelManaged algo)
```

Using Symmetric Encryption 7-13

- Following code snippet encrypts data using the RijndaelManaged class and writes the encrypted data to a file:

Snippet

```
{
            byte[] plainDataArray =
ASCIIEncoding.ASCII.GetBytes(plainText);


            ICryptoTransform transform=algo.CreateEncryptor();
            using (var fileStream = new
FileStream("D:\\CipherText.txt", FileMode.OpenOrCreate,
FileAccess.Write))
            {
                using (var cryptoStream = new CryptoStream(fileStream,
transform, CryptoStreamMode.Write))
                {
                    cryptoStream.Write(plainDataArray, 0,
plainDataArray.GetLength(0));
                    Console.WriteLine("Encrypted data written to:
D:\\CipherText.txt");
                }
            }
        }
```

Using Symmetric Encryption 8-13

Snippet

```
static void Main()
        {
           RijndaelManaged symAlgo = new RijndaelManaged();
           Console.WriteLine("Enter data to encrypt.");
           string dataToEncrypt = Console.ReadLine();
           EncryptData(dataToEncrypt, symAlgo);
        }
}
```

- In this code:
  - The Main() method creates a RijndaelManaged object and passes it along with the data to encrypt to the EncryptData() method.
  - In the EncryptData() method, the call to the CreateEncryptor() method creates the ICryptoTransform object.
  - Then, a FileStream object is created to write the encrypted text to the CipherText.txt file.
  - Next, the CryptoStream object is created and its Write() method is called.

Using Symmetric Encryption 9-13

- On the other hand, to decrypt data:
  - You can use the same symmetric encryption class, key, and IV used for encrypting the data.
  - You call the CreateDecryptor() method to obtain a ICryptoTransform object that will perform the transformation.
  - You then, need to create the CryptoStream object in Read mode and initialize a StreamReader object with the CryptoStream object.
  - Finally, you need to call the ReadToEnd() method of the StreamReader that returns the decrypted text as a string.
- Following code snippet shows creating a program that performs both encryption and decryption:

Snippet

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class SymmetricEncryptionDemo
{
static void EncryptData(String plainText, RijndaelManaged algo)
    {
```

Using Symmetric Encryption 10-13

Snippet

```
byte[] plainDataArray = ASCIIEncoding.ASCII.GetBytes(plainText);
    ICryptoTransform transform = algo.CreateEncryptor();
    using (var fileStream = new FileStream("D:\\CipherText.txt",
FileMode.OpenOrCreate, FileAccess.Write))
    {
      using (var cryptoStream = new CryptoStream(fileStream, transform,
CryptoStreamMode.Write))
      {
      cryptoStream.Write(plainDataArray, 0, plainDataArray.GetLength(0));
      Console.WriteLine("Encrypted data written to: D:\\CipherText.txt");
          }
      }
   }
   static void DecryptData(RijndaelManaged algo)
   {
     ICryptoTransform transform = algo.CreateDecryptor();
     using (var fileStream = new FileStream("D:\\CipherText.txt",
FileMode.Open, FileAccess.Read))
using (CryptoStream = new CryptoStream(fileStream, transform,
CryptoStreamMode.Read))
          {
```

Using Symmetric Encryption 11-13

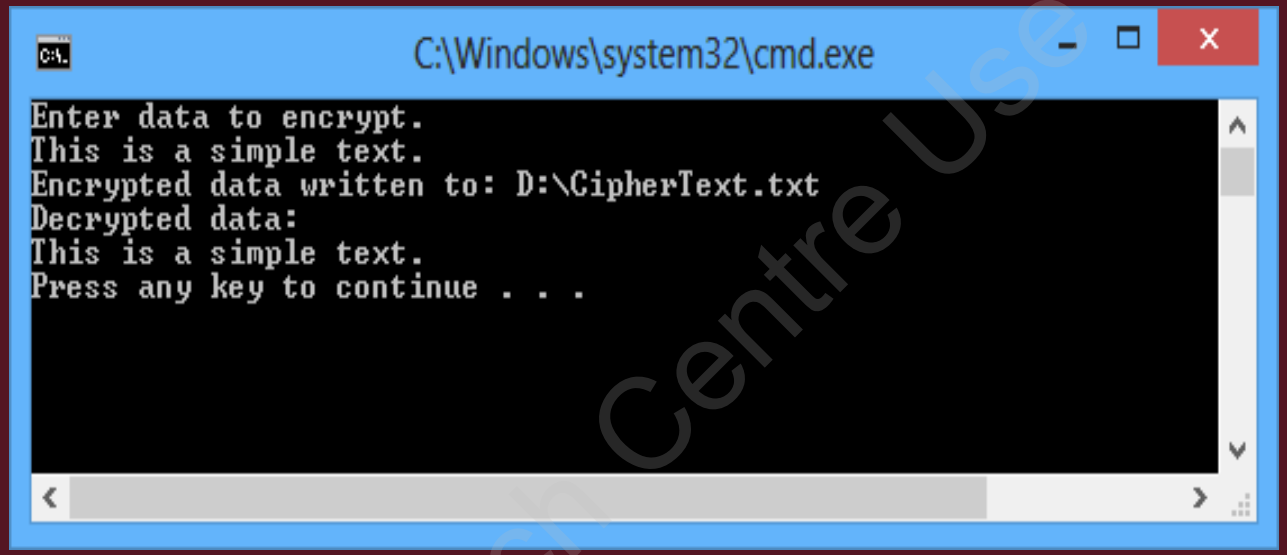
Snippet

```
 using (var streamReader = new StreamReader(cryptoStream))
        {
             string decryptedData = streamReader.ReadToEnd();
             Console.WriteLine("Decrypted data: \n{0}", decryptedData);
                }
            }
        }
    }
static void Main()
    {
        RijndaelManaged symAlgo = new RijndaelManaged();
        Console.WriteLine("Enter data to encrypt.");
        string dataToEncrypt = Console.ReadLine();
        EncryptData(dataToEncrypt, symAlgo);
        DecryptData(symAlgo);
    }
}
```

Using Symmetric Encryption 12-13

- In the code:
  - The Main() method creates a RijndaelManaged object and passes it along with the data to encrypt the EncryptData() method. The encrypted data is saved to the CipherText.txt file.
  - The Main() method calls the DecryptData() method passing the same RijndaelManaged object created for encryption.
  - The DecryptData() method creates the ICryptoTransform object and uses a FileStream object to read the encrypted data from the file.
  - The CryptoStream object is created in the Read mode initialized with the FileStream and ICryptoTransform objects.
  - A StreamReader object is created by passing the CryptoStream object to the constructor.
  - The ReadToEnd() method of the StreamReader object is called.

- Following figure shows the decrypted text returned by the ReadToEnd() method:



Using Asymmetric Encryption 1-12

- The System.Security.Cryptography namespace provides the RSACryptoServiceProvider class that you can use to perform asymmetric encryption.
- When you call the default constructor of the RSACryptoServiceProvider class, a new public-private key pair is automatically generated.
- After you create a new instance of this class, you can export the key information by using one of the following methods:
  - **ToXMLString():** Returns an XML representation of the key information.
  - **ExportParameters():** Returns an RSAParameters structure that holds the key information.
- Both of these methods accept a Boolean value.
- A false value indicates that the method should return only the public key information, while a true value indicates that the method should return information of both the public and private keys.

Using Asymmetric Encryption 2-12

- Following code snippet shows creating and initializing an RSACryptoServiceProvider object and then, exports the public key in XML format:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
. . .
RSACryptoServiceProviderrSAKeyGenerator = new
RSACryptoServiceProvider();
string publicKey = rSAKeyGenerator.ToXmlString(false);
```

- This code creates and initializes an RSACryptoServiceProvider object and then, export the public key in XML format.

Using Asymmetric Encryption 3-12

- Following code snippet shows creating and initializing an RSACryptoServiceProvider object:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
. . .
RSACryptoServiceProviderrSAKeyGenerator = new
RSACryptoServiceProvider();
RSAParametersrSAKeyInfo = rSAKeyGenerator.ExportParameters(true);
```

- This code creates and initializes an RSACryptoServiceProvider object and then, exports both the public and private keys as an RSAParameters structure.
- Now, to encrypt data, you need to create a new instance of the RSACryptoServiceProvider class and call the ImportParameters() method to initialize the instance with the public key information exported to an RSAParameters structure.

- Following code snippet shows initializing an RSACryptoServiceProvider object with the public key exported to an RSAParameters structure:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
. . .
. . .
RSACryptoServiceProviderrSAKeyGenerator = new
RSACryptoServiceProvider();
RSAParametersrSAKeyInfo = rSAKeyGenerator.ExportParameters(false);
RSACryptoServiceProviderrsaEncryptor= new RSACryptoServiceProvider();
rsaEncryptor.ImportParameters(rSAKeyInfo);
```

- If the public key information is exported to XML format, you need to call the FromXmlString() method to initialize the RSACryptoServiceProvider object with the public key.

Using Asymmetric Encryption 5-12

- Following code snippet shows initializing the RSACryptoServiceProvider object with the public key:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
. . .
RSACryptoServiceProviderrSAKeyGenerator = new
RSACryptoServiceProvider();
string publicKey = rSAKeyGenerator.ToXmlString(false);
RSACryptoServiceProviderrsaEncryptor= new RSACryptoServiceProvider();
rsaEncryptor.FromXmlString(publicKey);
```

- Once you have initialized the RSACryptoServiceProvider object with the public key, you can encrypt data by using the Encrypt() method of the RSACryptoServiceProvider class.
- The Encrypt() method accepts the two parameters byte array of the data to encrypt and a Boolean value that indicates whether or not to perform encryption.

Using Asymmetric Encryption 6-12

- The Encrypt() method after performing encryption returns a byte array of the encrypted text.
- Following code snippet shows returning a byte array of the encrypted text:

Snippet

```
byte[] plainbytes = new UnicodeEncoding().GetBytes("Plain text to
encrypt.");
byte[] cipherbytes = rsaEncryptor.Encrypt(plainbytes, true);
```

- To decrypt data, you need to initialize an RSACryptoServiceProvider object using the private key of the key pair whose public key was used for encryption.

Using Asymmetric Encryption 7-12

- Following code snippet shows how to initialize an RSACryptoServiceProvider object with the private key exported to an RSAParameters structure:

Snippet

```
RSACryptoServiceProviderrSAKeyGenerator = new
RSACryptoServiceProvider();
RSAParametersrSAKeyInfo = rSAKeyGenerator.ExportParameters(true);
RSACryptoServiceProviderrsaDecryptor= new RSACryptoServiceProvider();
rsaDecryptor.ImportParameters(rSAKeyInfo);
```

- This code initializes an RSACryptoServiceProvider object with the private key exported to an RSAParameters structure.

Using Asymmetric Encryption 8-12

- Following code snippet shows how to initialize an RSACryptoServiceProvider object with the private key exported to XML format:

Snippet

```
RSACryptoServiceProviderrSAKeyGenerator = new
RSACryptoServiceProvider();
string keyPair = rSAKeyGenerator.ToXmlString(true);
RSACryptoServiceProviderrsaDecryptor = new
RSACryptoServiceProvider();
rsaDecryptor.FromXmlString(keyPair);
```

- When the RSACryptoServiceProvider object is initialized with the private key, you can decrypt data by using the Decrypt() method of the RSACryptoServiceProvider class.
- The Decrypt() method accepts the two parameters, a byte array of the encrypted data and a Boolean value that indicates whether or not to perform encryption.

Using Asymmetric Encryption 9-12

Snippet

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class AsymmetricEncryptionDemo  {
static byte[] EncryptData(string plainText, RSAParameters
rsaParameters)
   {
byte[] plainTextArray = new UnicodeEncoding().GetBytes(plainText);
RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
RSA.ImportParameters(rsaParameters);
byte[] encryptedData = RSA.Encrypt(plainTextArray,true);
  return encryptedData;
}
static byte[] DecryptData(byte[] encryptedData, RSAParameters
rsaParameters)
 {
RSACryptoServiceProvider RSA = new
RSACryptoServiceProvider();
```

Using Asymmetric Encryption 10-12

- Following code snippet shows a program that performs asymmetric encryption and decryption:

Snippet

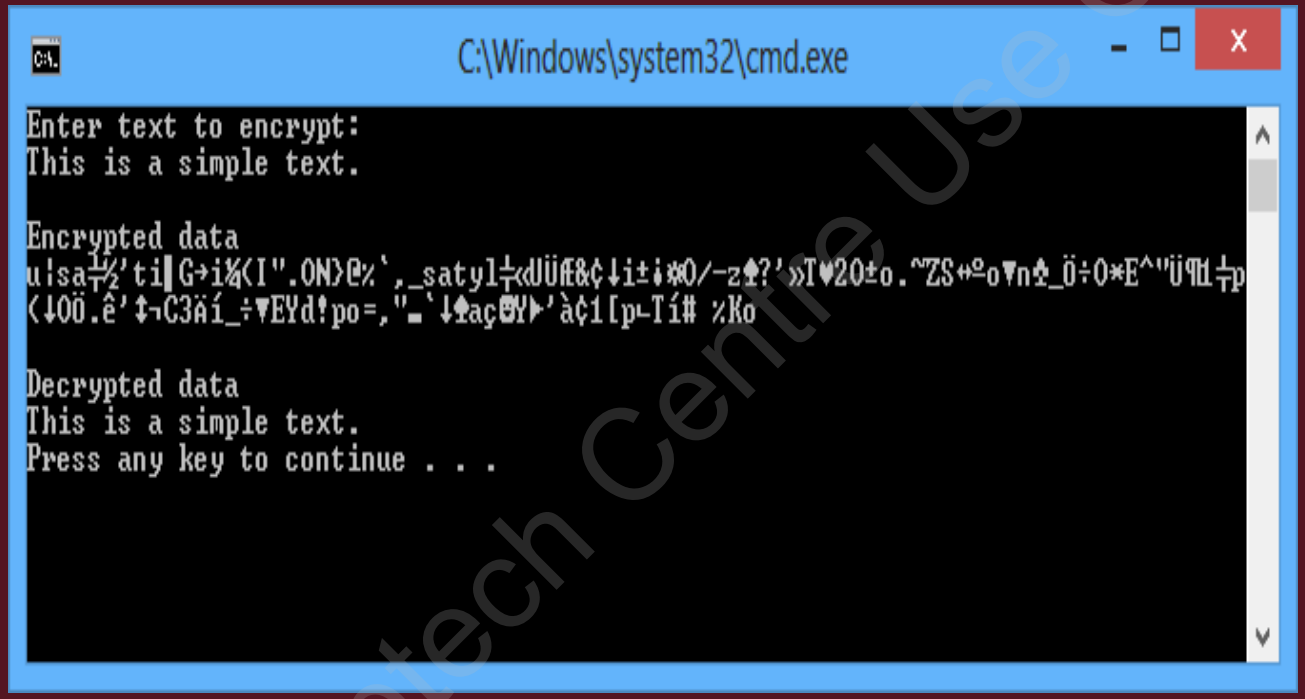
```
RSA.ImportParameters(rsaParameters);
byte[] decryptedData = RSA.Decrypt(encryptedData,true);
return decryptedData;
   }
static void Main(string[] args)
    {
Console.WriteLine("Enter text to encrypt:");
 String inputText = Console.ReadLine();
RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
RSAParameters RSAParam=RSA.ExportParameters(false);
byte[] encryptedData = EncryptData(inputText, RSAParam);
string encryptedString = Encoding.Default.GetString(encryptedData);
Console.WriteLine("\nEncrypted data \n{0}",encryptedString);
byte[] decryptedData = DecryptData(encryptedData,
RSA.ExportParameters(true));
String decryptedString = new
UnicodeEncoding().GetString(decryptedData);
Console.WriteLine("\nDecrypted data \n{0}", decryptedString);
        }
    }
```

Using Asymmetric Encryption 11-12

- In the code:
  - The Main() method creates a RSACryptoServiceProvider object and exports the public key as a RSAParameters structure.
  - The EncryptData() method is called passing the user entered plain text and the RSAParameters object. The EncryptData() method uses the exported public key to encrypt the data and returns the encrypted data as a byte array.
  - The Main() method then, exports both the public and private key of the RSACryptoServiceProvider object into a second RSAParameters object.
  - The DecryptData() method is called passing the encrypted byte array and the RSAParameters object.
  - Finally, the DecryptData() method performs the decryption and returns the original plain text as a string.

Using Asymmetric Encryption 12-12

- Following figure shows the output of the code:



Salting and Hashing 1-2

- Hashing is a technique that allows generating a unique hash value of data by using a hashing algorithm.
- Once you have generated a unique hash value for any data, this data cannot be converted back to the original form.
- The process of hashing in order to protect passwords in a database can be better understood by the following steps:
  - First, the application generates a hash value for a user password and stores it to the database.
  - Next, the application receives a login request comprising of a user name and password.
  - Then, the application generates a new hash value for the received password.
  - Finally, the application compares the newly generated hash value with the existing value stored in the database.

Salting and Hashing 2-2

- Salting is another security technique that you can apply on hashing.
- Salting allows creating and adding a random string to the input data before generating a hash value.
- The process of hashing in combination with salting to protect passwords stored in the database can be better understood by the following steps:
  - First, the application adds a random salt to the user password and generates a hash value and the application stores both the hash value and the salt to the database.
  - Then, the application receives a login request with a user name and password.
  - Next, the application retrieves the salt, applies it to the received password, and generates a new hash value.
  - Finally, the application compares both the new hash value and the existing one stored in the database.

Digital Signature 1-7

- You can use digital signature to authenticate the identity of a sender of some kind of data.
- This type of signature also ensures that the data has not been tampered while in transit.
- By using digital signature, a user sending a signed data cannot later deny his or her ownership of the data.
- The process of using digital signature requires the following operations to be performed:
  - First, a sender computes a hash value from the data being sent.
  - Then, the sender encrypts the hash value with the private key of an asymmetric key pair.
  - Next, the sender sends the data and the digital signature to the receiver.
  - Next, the receiver computes a hash from the received data.
  - Finally, the receiver decrypts the signature using the public key of the sender and then, compares the hash values for authenticity.

Digital Signature 2-7

- To understand how digital signatures work, you first need to generate a digital signature for some data.
- To generate the asymmetric keys whose private key can be used to sign data, you need to:
  - Use the RSACryptoServiceProvider class.
  - Then, you need to store the asymmetric keys using a secured mechanism.
  - Then, the CspParameters class can be used to create a key container and to add and remove keys to and from the container.
- To create a key container for an RSACryptoServiceProvider object:
  - You first need to use the default constructor of the CspParameters class to create a key container instance.
  - Then, you need to set the container name using the KeyContainerName property of the CspParameters class.

- Following code snippet shows using the KeyContainerName property of the CspParameters class:

Snippet

```
CspParameters param = new CspParameters();
 param.KeyContainerName = "SignatureContainer121";
```

- Now, you need to store the key pair of the RSACryptoServiceProvider object in the key container. For this:
  - You need to pass the CspParameters object to the constructor while creating the RSACryptoServiceProvider object.
  - Then, you need to set the PersistKeyInCsp property of the RSACryptoServiceProvider object to true.

- Following code snippet shows how to set the PersistKeyInCsp property:

Snippet

```
using (RSACryptoServiceProvider rsa = new
RSACryptoServiceProvider(param))
        {
                    rsa.PersistKeyInCsp = true;
                    . . .
        }
```

- In this code, an RSACryptoServiceProvider object is initialized with a CspParameters object and then, the key pair is stored in the CspParameters object.
- Then, you can generate the signature using the SignData() method.
- This method accepts the data to sign in a byte array and the hash algorithm is used to create the hash value.
- You can pass the SHA256 string value to use the SHA256 hash algorithm to the SignData() method that returns the signature in a byte array.

- Following code snippet shows how to use the SignData() method:

Snippet

```
byte[] byteData = Encoding.Unicode.GetBytes("Data to Sign.");
byte[] byteSignature = rsa.SignData(byteData, "SHA256");
```

- In this code, the SignData() generates a digital signature and returns the signature in a byte array.
- Once you have created the digital signature, you can verify it.
- For that, you need to first create a CspParameters object and use the same key container name that you used while creating the signature.
- This will ensure that when you later create the RSACryptoServiceProvider object, it will be initialized with the same key pair that was used to generate the signature.

- Following code snippet shows creating a CspParameters object:

Snippet

```
CspParameters param = new CspParameters();
param.KeyContainerName = "SignatureContainer101";
```

- After creating a CspParameters object, you can create the RSACryptoServiceProvider object initialized with the CspParameters object.
- You can then, call the VerifyData() method that accepts three different types of parameters, such as the **byte array of the data** that needs to be verified against the signature, the **hash algorithm** that is used to generate the signature, and the **byte array of the generated signature**.

- The VerifyData() method returns a Boolean true value if the signature is successfully verified, or false.
- Following code snippet shows using the VerifyData() method that returns a Boolean true value:

Snippet

```
bool isSuccess = false;
using (RSACryptoServiceProvider rsa = new
RSACryptoServiceProvider(param))
{
 isSuccess = rsa.VerifyData(byteData, "SHA256", byteSignature);
}
```

- This code creates an RSACryptoServiceProvider object and then, calls the VerifyData() method to verify the signature.

Summary

- HTML encoding is a process that converts potentially unsafe characters to their HTML-encoded equivalent.
- The ASP.NET MVC Framework provides a request validation feature that examines an HTTP request to check and prevent potentially malicious content.
- The user-generated token approach enables storing a user-generated token using a HTML hidden field in the user session.
- SQL injection is a form of attack in which a user posts SQL code to an application, thus it creates an SQL statement that will execute with malicious intent.
- Encryption is a technique that ensures data confidentiality by converting data in plain text to cipher (secretly coded) text.
- Hashing is a technique that allows generating a unique hash value of data by using a hashing algorithm.
- Salting is a security technique that you can apply on hashing to create and add a random string to the input data before generating a hash value.