PRODUCT

Secure Coding Standards for .Net

Amendment History

Version Number	Date	Author/Changed by	Description of Changes Made
0.1			Draft
0.2	18-Oct-12	Smitha Nair	Formatting done for the existing document
1.0	19-Oct-12	EPG Subgroup	Baselined for publishing.

Contents

1. Introduction:

A coding standard is a set of guidelines, rules and regulations on how to write code which will help programmers/developer quickly read and understand source code conforming to the style as well as helping to avoid introducing faults and misunderstanding.

Coding standards are important because they provide greater consistency and uniformity in writing code between programmers

Advantages of coding standard are:

- Improve the readability of the code.
- Easy to understand and maintain by others.
- Separate documents appear more appropriate.
- Maintainable applications.
- Remove complexity.

This document covers the following topics:

- Naming conventions and Standards
- Indentation and Spacing
- Exception Handling
- Comments
- Good Programming practices
- Guidelines for Secure Coding
- Security Checklist: .NET Framework

2. Naming Conventions and Standards

Pascal Casing - First character of all words are Upper Case and other characters are lower case. Example: BackColor

Camel Casing - First character of all words, except the first word are Upper Case and other characters are lower case. Example: backColor

1. Use Pascal casing for Class names

```
public class HelloWorld

{

      ...

}
```

2. Use Pascal casing for Method names

```
void SayHello(string name)

{

      ...

}
```

3. Use Camel casing for variables and method parameters

```
int totalCount = 0;

void SayHello(string name)

{

      string fullMessage = "Hello " + name;

      ...

}
```

4. Use the prefix “I” with Camel Casing for interfaces (Example: IEntity)

5. Use Meaningful, descriptive words to name variables. Do not use abbreviations.

Correct:

```
string address

int salary
```

Avoid:

```
string nam

string addr

int sal
```

6. Do not use single character variable names like i, n, s etc. Use names like index, temp

   One exception in this case would be variables used for iterations in loops:

```
for ( int i = 0; i < count; i++ )

{

      ...

}
```

7. Do not use underscores (_) for local variable names.

8. All member variables must be prefixed with underscore (_) so that they can be identified from other local variables.

9. Do not use variable names that resemble keywords.

10. Prefix boolean variables, properties and methods with "is" or similar prefixes.

    Ex: private bool _isFinished

11. Namespace names should follow the standard pattern

```
<company name>.<product name>.<top level module>.<bottom
level module>
```

12. Use appropriate prefix for the UI elements so that you can identify them from the rest of the variables.

Control	Prefix
Label	Lbl
TextBox	Txt
DataGrid	Dtg
Button	Btn
ImageButton	Imb
Hyperlink	Hlk
DropDownList	Ddl
ListBox	Lst
DataList	Dtl
Repeater	Rep
Checkbox	Chk
CheckBoxList	Cbl
RadioButton	Rdo
RadioButtonList	Rbl
Image	Img
Panel	Pnl
PlaceHolder	Phd
Table	Tbl
Validators	Val

13. File name should match with class name.

For example, for the class HelloWorld, the file name should be helloworld.cs (or, helloworld.vb)

14. Use Pascal Case for file names.

3. Indentation and Spacing

1. Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.

2. Comments should be in the same level as the code (use the same level of indentation).

Correct:

```
// Format a message and display

string fullMessage = "Hello " + name;

DateTime currentTime = DateTime.Now;

string message = fullMessage + ", the time is : " + currentTime.ToShortTimeString();

MessageBox.Show ( message );
```

Avoid:

```
// Format a message and display

string fullMessage = "Hello " + name;

DateTime currentTime = DateTime.Now;

string message = fullMessage + ", the time is : " + currentTime.ToShortTimeString();

MessageBox.Show ( message );
```

3. Curly braces ({}) should be in the same level as the code outside the braces.

```
If ( … )

{

	\\ Do something

}
```

4. Use one blank line to separate logical groups of code.

Correct:

```
      bool SayHello ( string name )
      {
            string fullMessage = "Hello " + name;
            DateTime currentTime = DateTime.Now;

            string message = fullMessage + ", the time is
: "
                            +
currentTime.ToShortTimeString();
            MessageBox.Show ( message );
            if ( ... )
            {
                  // Do something
                  // ...

                  return false;
            }
            return true;
      }
```

Avoid:

```
      bool SayHello (string name)
      {
            string fullMessage = "Hello " + name;
            DateTime currentTime = DateTime.Now;
            string message = fullMessage + ", the time is
: " + currentTime.ToShortTimeString();
            MessageBox.Show ( message );
            if ( ... )
            {
```

```
                        // Do something

                        // ...

                        return false;

                }

                return true;

        }
```

5. There should be one and only one single blank line between each method inside the class.

6. The curly braces should be on a separate line and not in the same line as if, for etc.

Correct:

```
                if ( ... )

                {

                        // Do something

                }
```

Avoid:

```
                if ( ... ) {

                        // Do something

                }
```

7. Use a single space before and after each operator and brackets.

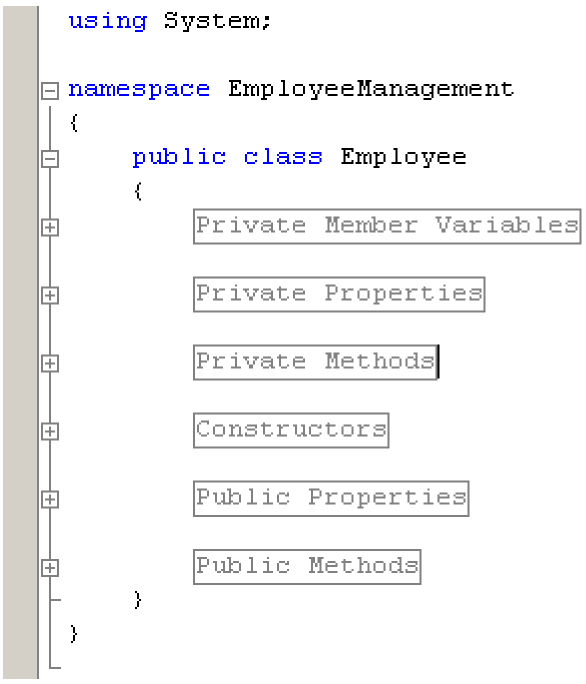
Correct:

```
                if ( showResult == true )

                {

                        for ( int i = 0; i < 10; i++ )

                        {

                                //

                        }
                }
```

Avoid:

```
                if(showResult==true)
                {
                        for(int         i= 0;i<10;i++)
                        {
                                //
                        }
                }
```

8. Use #region to group related pieces of code together. If you use proper grouping using #region, the page should like this when all definitions are collapsed.



```
using System;

namespace EmployeeManagement
{
    public class Employee
    {
        Private Member Variables

        Private Properties

        Private Methods

        Constructors

        Public Properties

        Public Methods
    }
}
```

4. Exception Handling

1. Never do a 'catch exception and do nothing' should log the exception and proceed.

2. In case of exceptions, give a friendly message to the user, but log the actual error with all possible details about the error, including the time it occurred, method and class name etc.

3. Always catch only the specific exception, not generic exception.

Correct:

```
      void ReadFromFile ( string fileName )

      {

            try

            {

                  // read from file.

            }

            catch (FileIOException ex)

            {

                  // log error.

                  //re-throw  exception  depending  on  your
case.

                  throw;

            }

      }
```

Avoid:

```
void ReadFromFile ( string fileName )

{

   try

   {

      // read from file.
```

```
    }

    catch (Exception ex)

    {

        // Catching general exception is bad... we will
never know whether

        // it was a file error or some other error.

        // Here you are hiding an exception.

        // In this case no one will ever know that an
exception happened.

        return "";

    }

}
```

4. When re throwing an exception, use the throw statement without specifying the original exception. This way, the original call stack is preserved.

   Correct:

```
catch

{

       // do whatever you want to handle the exception

       throw;

}
```

   Avoid:

```
catch (Exception ex)

{

       // do whatever you want to handle the exception

       throw ex;

}
```

5. Do not write try-catch in all the methods. Use it only if there is a possibility that a specific exception may occur and it cannot be prevented by any other means.

6. Do not write very large try-catch blocks. If required, write separate try-catch for each task you perform and enclose only the specific piece of code inside the try-catch. This will helps to find which piece of code generated the exception.

7. Do not derive own custom exceptions from the base class SystemException. Instead, inherit from ApplicationException.

5. Comments

1. All source code must include the following comments at the very top:

```
/***********************************************************
*Author:
*Date:
*Description:
*Revision:
***********************************************************/
```

2. All comments should be written in English(like in U.S. English).

3. Comments lines should begin with // indented at the same level as the code they are documenting.

4. Do not use /* … */ blocks for comments.

5. Source code comments should be written in clear, concise, English sentences.

6. Comments should be included in source code to indicate what the code does and to break up large sections of code.7..

7. Comments should not be used to describe obvious code. The purpose of comments is to increase your code readability. Often using good variable and method names make the code easier to read than if too many comments are present.

9. Never "comment out" code without also including a description of why you did so. If the code isn't necessary it should be deleted.

10. Do not use "clutter" comments, such as an entire line of asterisks. Instead, use a single blank line white space line to separate comments from code.

11. Do not "draw flower boxes" around comments using rows of asterisks or other characters.

12. Comments should not include humorous remarks. Your comments may seem funny to you when you type them, but they won't to the person who has to understand and fix your code during a late-night debugging session.

13. Include Task-List keyword flags to enable comment filtering.

Example: // TODO: Place Database Code Here

14. Always include <summary> comments. Include <param>, <return> and <exception> comment sections where applicable

15. Always add CDATA tags to comments containing code and other embedded markup in order to avoid encoding issues.

16. Indent comment at the same level of indentation as the code you are documenting.

17. All comments should pass spell checking. Misspelled comments indicate sloppy development.

6. Good Programming practices

1. Do not hard code strings/ numeric. Instead of that use constants as shown below.

   Bad Practice:

```
int Count;
 Count = 100;
 if(  Count   ==   0 )
 {
  // DO something…
 }
```

   Good Practice:

```
int Count;
 Count = 100;
 private static const int ZERO   =   0;
 if(  Count   ==   ZERO )
  {
    // DO something…
  }
```

2. For string comparison - Use String. Empty instead of “”

   Good Practice:

```
If ( name == String.Empty )
{
      // do something
}
```

   Bad Practice:

```
If ( name == "" )
{
      // do something
}
```

3. By default keep the scope of member variables to ‘private’, based on the needs expand the scope either to protected or public or internal.

4. For manipulating the strings inside a loop, use StringBuilder instead of string as shown below.

   Bad Practice :

```
String  temp = String.Empty;
 for( int I = 0 ; I<= 100; i++)
 {
   Temp += i.ToString();
 }
```

   Good Practice :

```
StringBuilder sb = new StringBuilder();
 for ( int I = 0 ; I<= 100; i++)
  {
    sb.Append(i.ToString());
   }
```

5. Use the appropriate data types.

   For ex: For checking for any status, prefer bool rather than int.

   Bad Practice:

```
int Check = 0;
 if(Check == 0)
  {
    // DO something
   }
```

   Good Practice :

```
bool Check = false;
 if(!Check)
```

```
    {
      // DO something
    }
```

6. Use 'as' operator for type casting and before using that resultant value check for null.

```
class A
 {
 }
class B : A
 {
 }
B objB = new B();
A objA1  = (A) objB;
A objA 2 = objB as A;
if( objA2 != null)
{
 //Do something
}
```

7. Avoid declaring the 'destructor' for every class. This will increase the life time of the class which un necessarily makes them long lived

8. Do not make any calls to a method from the loop.

   Bad Practice :

```
for( int i = 0; i<= 100; i++)
{
      Calculate(i);
}
```

   Good Practice:

```
for( int i = 0; i<= 100; i++)
{
   //Inline the body of Calculate.

}
```

9. Do not handle exceptions inside a loop, rather handle looping logic inside try/catch

   Bad Practice :

```
for(int i = 0 ; i<= 100; i++){

   try{

   }

   catch(Exception ex){

      throw ex;

   }

}
```

Good Practice:

```
 try{

    for(int i = 0 ; i<= 100; i++){

    }

  }

  catch(Exception ex){

   throw ex;

  }
```

10. Do not handle application logic by using Exception.

    Bad Practice:

```
try{

   int  x,y,z;

   x = 0;

   y = 10;

   z = y/x;

  }
```

```
catch(DevideByZeroException ex){

  throw ex;

}
```

Good Practice:

```
private static const int ZERO  =  0;

 try{

   int x,y,z;

   x = 0;

   y = 10;

   if( x != ZERO){

     z = y/x;

   }

  }

 catch(Exception ex){

  }
```

11. Prefer for/while loop over foreach

12. Method name should tell what it does. Do not use mis-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

Good Practice:

```
      void SavePhoneNumber ( string phoneNumber )

      {

            // Save the phone number.

      }
```

Bad Practice:

```
      // This method will save the phone number.

      void SaveDetails ( string phoneNumber )

      {
            // Save the phone number.

      }
```

13. A method should do only 'one job'. Do not combine more than one job in a single method, even if those jobs are very small.

Good Practice:

```
      // Save the address.

      SaveAddress (  address );



      // Send an email to the supervisor to inform that the
address is updated.

      SendEmail ( address, email );



      void SaveAddress ( string address )

      {

            // Save the address.

            // ...

      }

      void SendEmail ( string address, string email )

      {

            // Send an email to inform the supervisor that the
address is changed.

            // ...

      }
```

Bad Practice:

```
      // Save address and send an email to the supervisor to
inform that

// the address is updated.

      SaveAddress ( address, email );
```

```
        void SaveAddress ( string address, string email )

        {

                // Job 1.

                // Save the address.

                // ...

                // Job 2.

                // Send an email to inform the supervisor that the
address is changed.

                // ...

        }
```

14. Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods.

15. Use enum wherever required. Do not use numbers or strings to indicate discrete values.

Good Practice:

```
        enum MailType

        {

                Html,

                PlainText,

                Attachment

        }

        void SendMail (string message, MailType mailType)

        {

                switch ( mailType )

                {

                        case MailType.Html:

                                // Do something

                                break;

                        case MailType.PlainText:

                                // Do something
                        break;
                  case MailType.Attachment:
                        // Do something
                        break;
                  default:
                        // Do something
                        break;
            }
      }
```

Bad Practice:

```
      void SendMail (string message, string mailType)
      {
            switch ( mailType )
            {
                  case "Html":
                        // Do something
                        break;
                  case "PlainText":
                        // Do something
                        break;
                  case "Attachment":
                        // Do something
                        break;
                  default:
                        // Do something
                        break;
            }
      }
```

16. Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.

17. In the application start up, do some kind of "self check" and ensure all required files and dependancies are available in the expected locations. Check for database connection in start up, if required. Give a friendly message to the user in case of any problems.

18. If the required configuration file is not found, application should be able to create one with default values.

19. If a wrong value found in the configuration file, application should throw an error or give a message and also should tell the user what are the correct values.

20. Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."

21. When displaying error messages, in addition to telling what is wrong, the message should also tell what should the user do to solve the problem. Instead of message like "Failed to update database.", suggest what should the user do: "Failed to update database. Please make sure the login id and password are correct."

22. Show short and friendly message to the user. But log the actual error with all possible information. This will help a lot in diagnosing problems.

23. Do not have more than one class in a single file.

24. Avoid having very large files. If a single file has more than 1000 lines of code, Split them logically into two or more classes.

25. Avoid public methods and properties, unless they really need to be accessed from outside the class. Use “internal” if they are accessed only within the same assembly.

26. Avoid passing too many parameters to a method.

27. If a method returning a collection, return an empty collection instead of null, if there is no data to return.

28. Use the AssemblyInfo file to fill information like version number, description, company name, copyright notice etc.

29. Make sure there is a good logging class which can be configured to log errors, warning or traces.

30. If we are opening database connections, sockets, file stream etc, always close them in the finally block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the finally block.

7. Guidelines for Secure Coding

Writing secure code is an essential part of secure software development. While doing the coding unintentional ignorance of known vulnerabilities and insecure coding practices can generate a security flaw in a program. Such a flaw could be responsible for crashing a program or enabling a denial of service attack. In any case, an effective approach must be adopted to detect and eliminate such flaws.

7.1 Architecture and Design

Early in the architecture and design stage of development, should start the threat modeling activity to identify threats and vulnerabilities relevant to the application.

- **Identifying and evaluate threats**

  Rather than applying security in a haphazard manner, use threat modeling to identify threats systematically.

  For more information about threat modeling, see Threat Modeling Web Applications.

- **Creating designs that meet security objectives**

  Use tried and tested design patterns and principles. Focus on those areas that are critical from a security perspective, such as authentication, authorization, input / data validation, exception management, sensitive data, auditing and logging, configuration management, and session management. Also pay attention to deployment issues at design time. Give consideration to deployment topologies, network infrastructure, security policies, and procedures.

  For more information, see Design Guidelines for Secure Web Applications.

- **Perform a security architecture and design review**

  To perform security-focused architecture and design reviews, need to consider three major elements:

  - **Deployment and infrastructure**. Review the design of the application as it relates to the target deployment environment and the associated security policies. Consider the constraints imposed by the underlying infrastructure-layer security and the operational practices in use.
  - **Security frame**. Review the security approach that was used for the critical areas of the application. An effective way to do this is to focus on the set of categories that have the most impact on security, particularly at an architectural and design level, and where mistakes are most often made. These include authentication, authorization, input validation, exception management, and other areas.

- **Layer-by-layer analysis**. Review the logical layers of application, and evaluate the security choices in presentation, business, and data-access logic.

For more information, see Security Architecture and Design Review.

7.2 Development

During development, regular security code-reviews should be conducted to identify any security issues before code reaches the test team.

- **Writing secure managed code**

  For writing managed code with fewer security issues, should adhere to solid, object-oriented design principles and apply secure coding guidelines. For example, Use role-based authorization to provide access controls on public classes and class members. Use structured exception handling to return generic errors to the client. Restrict type and member visibility to limit the code that is publicly accessible. Do not store sensitive data, such as connection strings, encryption keys, and user account credentials in code.

  For more secure coding guidelines, see Security Guidelines for ASP.NET 2.0.

- **Review code for security issues**

  A properly conducted code review can increase the security of the application more than nearly any other activity. To perform a security code review, start by identifying security code-review objectives to establish goals and constrains for the review .Review the most vulnerable areas of code thoroughly to find security issues that are common to many applications. Finally, review for security issues that are unique to the architecture of the application.

  For more information, see patterns & practices Security Code Review Index.

7.3 Deployment

Before deploying the code, perform security deployment reviews to ensure that weak or invalid configuration settings do not expose security vulnerabilities.

- **Deploying applications securely**

  Before deploying application, perform a security deployment review to identify potential security vulnerabilities introduced by inappropriate configuration settings. Make sure that connection strings are encrypted: use protected configuration. Review any other configuration file element that could contain sensitive data, such as account credentials, and ensure that the section is encrypted if it contains sensitive data. Make sure that trace and debug are disabled. If application accesses a database, make sure that it uses a login with limited permissions in the database. If application uses the **ActiveDirectoryMembershipProvider** class, encrypt the configuration section if it contains connection credentials.

For more information, see How To: Perform a Security Deployment Review for ASP.NET 2.0.

- **Protecting forms authentication and authorization during deployment**

  Forms authentication is protected by appropriate configuration of the **forms** and **machineKey** elements. We should ensure that authentication and role cookies are encrypted and checked for integrity and also make sure that the **authorization** element is configured appropriately to restrict access to application's sensitive pages. Configure file upload limits in the **maxRequestLength** attribute on the **httpRuntime** element, and make sure that we prevent the download of unused file types by removing unnecessary MIME type mappings from IIS and by mapping unused ASP.NET file extensions to the **HttpForbiddenHandler** class in the **httpHandlers** section. Use health monitoring to log security-related events, such as failed login attempts.

  For more information, see How To: Perform a Security Deployment Review for ASP.NET 2.0.

7.4 Input and Data Validation

Proper input validation has to be implemented for defense against today's application attacks. Proper input validation is an effective countermeasure that can help prevent cross-site scripting, SQL injection, buffer overflows, and other input attacks.

- **Applying constraints for file I/O**

  The code access security has to be used for applying constraints in order to determine which files or directories an application can access. This can be limited using **FileIOPermission**.

  Note: The ASP.NET web application should be set to run in medium trust for limiting file access to the application’s virtual directory hierarchy.

- **Applying constraints input for length, range, format, and type**

  Validation controls, such as **RangeValidator**, **CustomValidator**, and **RegularExpressionValidator has to be used**, to constrain the input by length, range, format, and type, when input is from a server.

  For more information about using regular expressions, see How To: Use Regular Expressions to Constrain Input in ASP.NET.

- **Create an input and data validation architecture**

  Develop a library of validation routines, this will help us to ensure that data is validated in a consistent way throughout the application and provide a single point of maintenance

  For more information about protecting Web applications from injection attacks, see How To: Protect from Injection Attacks in ASP.NET.

7.5 Authorization

Proper role base authorization has to be used otherwise it can leads to information disclosure and data tampering. Make use of **Security**, **AccessRule**, and **AuditRule** classes to modify access control programmatically, to create new objects by copying ACLs from one object to another, or to set security audits on various system objects.

7.6 Exception Management

Robust exception-handling code is essential to help ensure application stability and to help prevent sensitive exception details, which could be useful to an attacker, from reaching the client.

- **Handling exceptions securely**

  When application throws an exception, make sure that it fails without disclosing sensitive information, denies access, and is not left in an unsecured state. Do not log sensitive or private data, such as passwords, that could be compromised. Do not reveal internal system or application details, such as stack traces, SQL statements, and table or database names.

- **Use structured exception handling**

  Structured exception-handling has to be implemented to avoid unhandled exceptions. For releasing all resources, disposing all objects regardless of whether exception occurs always make use of **Finally** block. The following code example shows to use a **finally** block to ensure that a database connection is closed promptly.

```
using System.Data.SqlClient;

using System.Security;

SqlConnection conn = new SqlConnection("...");

try

{

    conn.Open();

    // Do some operation that might cause an exception.

    // Calling Close as early as possible.

    conn.Close();

    // ... other potentially long operations.

}

finally

{
    if (conn.State==ConnectionState.Open)

        conn.Close();   // ensure that the connection is
closed.

}
```

7.7 Communication Security

Communication channel has to be protected properly while transferring application specific sensitive data over the network.

- **Protecting communication between desktop and server**

  SSL has to be used to encrypt the communication channel between specific client applications and a server.

- **Protecting communication between server and server**

  Internet Protocol security (IPsec) to be implemented to encrypt the communication channel between two servers and to restrict which computers can communicate with one another.

7.8 Data Access

While developing data access code, main issues to be considered are protecting connection strings, authentication with database, preventing SQL injection attacks.

- **Protecting database connection strings**

  Database connection strings should always be stored in connectionStrings section of configuration file and then use protected configuration to encrypt this section

  For more information, see the section, How to Encrypt Configuration Data in Configuration Files,

- **Use Windows authentication to connect to a SQL Server database**

  For using Windows authentication, configure a SQL Server database for Windows authentication, and then use a connection string that contains either **"Trusted_Connection=Yes"** or **"Integrated Security=SSPI"**, as shown in the following code example. The two strings are equivalent and both result in Windows authentication.

  "server=MySQL; Integrated Security=SSPI; database=Northwind"
  "server=MySQL; Trusted_Connection=Yes; database=Northwind"

- **Preventing SQL injection attacks**

For preventing SQL injection attacks, Validate input for type, length, format, and range. Use regular expressions to validate text input. Also use parameterized stored procedures for data access. This ensures that input values are checked for type and length. Parameters are also treated as safe literal values and not as executable code within the database. Avoid stored procedures that accept a single parameter that has the query to execute. Instead, pass query parameters only.

7.9 Unmanaged Code

Threat of buffer overflow attacks should be considered if application calls unmanaged code as unmanaged code is not subject to code access security and can perform any operation that is subject to operating-system security..

- **Protecting calls to unmanaged code**

  Restrict access to unmanaged code by demanding custom permissions before asserting the unmanaged code permission. Before calling unmanaged code, validate input that is passed to unmanaged APIs and guard against potential buffer overflows and array boundary violations. Validate the lengths of input and output string parameters, validate array bounds, and check file path lengths. Inspect unmanaged code for unsafe APIs.

- **Isolating and name unmanaged APIs**

  Isolate all unmanaged API calls in a wrapper assembly. This makes it easier to review code that calls unmanaged code, and it enables to easily determine the set of unmanaged APIs on which application depends.

7.10 Sensitive Data

Applications that deal with private user information, such as credit card numbers, addresses, medical records, and so on should take special steps to make sure that the data remains private and unaltered. In addition, sensitive data that is used by the application's implementation, such as passwords and database connection strings, must be protected.

- **Encrypting configuration data in configuration files**

  Sensitive data stored in configuration section of Web.config file should be encrypted with the help ASP.NET IIS registration tool.

  For more information, see How To: Encrypt Configuration Sections in ASP.NET 2.0 Using DPAPI and How To: Encrypt Configuration Sections in ASP.NET 2.0 Using RSA.

- **Protecting passwords**

  For protecting password if application uses Windows authentication then it would be better to use Windows or Active Directory password policy. Password must be encrypted if stored in configuration file. Password stored in database should not be store in plaintext or encrypted form. Instead, store non-reversible hashes with added salt.

- **Protecting secrets in memory**

  Sensitive data in memory by using the **ProtectedMemory** class . The **ProtectedMemory** class is a managed wrapper for DPAPI.

  The following code example shows how to use the **ProtectedMemory** class to encrypt and decrypt data in memory.

```
using System.Security.Cryptography;

...

byte[] optionalEntropy = {7,5,4,9,0};

byte[]                    dataToBeEncrypted                    =
Encoding.Unicode.GetBytes("Test String 1211");

// Encrypt the data in memory

ProtectedMemory.Protect(dataToBeEncrypted,

                        MemoryProtectionScope.SameLogon);

// Decrypt the data in memory

ProtectedMemory.Unprotect(dataToBeEncrypted,


MemoryProtectionScope.SameLogon);

string                   originalData                          =
Encoding.Unicode.GetString(dataToBeEncrypted);
```

- **Using DPAPI to protect sensitive data**

  To use DPAPI to encrypt sensitive data in configuration files, use the **DataProtectionConfigurationProvider** class. To encrypt sensitive data with DPAPI in ASP.NET Web.config files, use Aspnet_regiis.exe. For Windows applications, use the protected configuration API. For more information, see the section, How to Encrypt Configuration Data in Configuration Files, in this document.

  To use DPAPI to encrypt data in other locations, such as the registry, use the **ProtectedData** class.

```
using System.Security.Cryptography;

// Define an entropy value (use of entropy is optional).

byte[] entropy = new byte[] { 0x23, 0x06, 0x08, 0x09,
0x22, 0x03, 0x25 };

// Get the original data in a byte array.
```

```
byte[] toEncrypt = UnicodeEncoding.ASCII.GetBytes(

                    "The secret data to be encrypted");

// Encrypt the data by the using ProtectedData class.
This example uses

// the local machine key store.

byte[] encryptedData = ProtectedData.Protect(toEncrypt,
entropy,

                        DataProtectionScope.LocalMachine);
```

- **Using X.509 certificates to encrypt XML data**

  For encrypting data in digital certificates, use the X.509 certificate support in the **System.Security.Cryptography.X509Certificates** namespace to encrypt XML data.

  To use X.509 certificates, access the certificate from the certificate store, create a new instance of the **EncryptedXml** class, call its **Encrypt** method, and pass the certificate. Finally, replace the clear text XML element with the **EncrytedData** element.

  To be able to decrypt the data, must have the private key associated with the public key in the X.509 certificate.

  For more information, see How to: Encrypt XML Elements with X.509 Certificates.

- **Using XML signatures**

  XML digital signatures can be used to ensure the integrity and authenticity of origin of XML documents. Use of the classes in the **System.Security.Cryptography.Xml** namespace to sign an XML document or part of an XML document with a digital signature.

  The following code example shows how to use an RSA signing key to sign an XML document.

```
using System.Security.Cryptography;

using System.Security.Cryptography.Xml;

using System.Xml;

// Create a new RSA key container and save it in a key
container.

CspParameters cspParams = new CspParameters();
```

```
cspParams.KeyContainerName = "MyRSASigningKey";

// Create a new RSA signing key and save it in the container.

RSACryptoServiceProvider rsaKey = new


RSACryptoServiceProvider(cspParams);

//  Create a SignedXml object and add the key to it.

SignedXml signedXml = new SignedXml(doc);

signedXml.SigningKey = Key;

// Create a Reference object that describes what to sign.

Reference reference = new Reference();

reference.Uri = ""; // "" means sign the whole document

// Add an enveloped transformation to the reference, add the reference

// to the signed XML document, and then compute the signature.

XmlDsigEnvelopedSignatureTransform env = new


XmlDsigEnvelopedSignatureTransform();

reference.AddTransform(env);

// Add the reference to the SignedXml object.

signedXml.AddReference(reference);

// Compute the signature.

signedXml.ComputeSignature();

//  Get the XML representation of the signature and save it in an XmlElement object. Append the element to the original XML document.

XmlElement xmlDigitalSignature = signedXml.GetXml();

// Append the element to the XML document.

Doc.DocumentElement.AppendChild(doc.ImportNode(xmlDigitalSignature, true));
```

7.11 Code Access Security

Code access security has to be used to restrict what your code can do, and to restrict which code can call your code. Code access security is a resource-constraint model designed to restrict the types of system resources that code can access and the types of privileged operations that the code can perform. These restrictions are independent of the user who calls the code or the user account under which the code runs.

Code access security is applicable in the following scenarios:

- Using ClickOnce to deploy and execute applications.
- Running partial trust ASP.NET Web applications, for example on hosted servers.
- Running managed controls inside Internet Explorer.
- Running code from an intranet file share.
- Running code inside a Microsoft Office application.
- Developing a class library to be used by other applications, including partial trust applications.

- **Use the AllowPartiallyTrustedCallersAttribute attribute (APTCA)**

  APTCA can be used to override the implicit link demand for full trust that is placed on every publicly accessible member of a strong-named assembly. APTCA is an assembly-level attribute, as shown in the following code example.

```
[assembly: AllowPartiallyTrustedCallersAttribute()]
```

- **Use security transparency**

  When the assembly does not contain any critical code and does not elevate the privileges of the call stack in any way add transparency to an assembly with the following attribute:

```
 [assembly: SecurityTransparent]
```

  For performing security-critical actions , explicitly mark another Security Critical attribute on the code that will perform the critical action , as shown in the following code example.

```
[assembly: SecurityCritical]

public class A

{

   [SecurityCritical]

   public void Critical()
```

```
    {
        // critical
    }
    public int SomeProp
    {
        get {/* transparent */ }
        set {/* transparent */ }
    }
}
public class B
{
    internal string SomeOtherProp
    {
        get { /* transparent */ }
        set { /* transparent */ }
    }
}
```

The above code is transparent (this is the default setting, even with the assembly-level **SecurityCritical** attribute) except for the **Critical** method, which is explicitly marked as critical.

- **Using declarative and imperative permission requests**

  The following attribute ensures that only users who are members of the Manager role can call the **GetCustomerDetails** method.

```
using System.Security.Permissions;
using System.Threading;
...
[PrincipalPermissionAttribute(SecurityAction.Demand, Role="Manager")]
public void GetCustomerDetails(int CustId)
```

```
{

}
```

The following code example uses explicit role checks.

```
public void GetCustomerDetails(int CustId)

{

    if(!Thread.CurrentPrincipal.IsInRole("Manager"))

    {

    . . .

    }

}
```

8. Security Checklist: .NET Framework

Assembly Design Considerations

Check	Description
☐	Target trust environment is identified. Permissions available to partial trust code and APIs that require additional permissions are identified.
☐	Design exposes a minimal number of public interfaces to limit the assembly's attack surface.

Class Design Considerations

Check	Description
☐	To reduce visibility, classes and members use the most restrictive access modifier possible.
☐	Base classes that are not intended to be derived from are sealed.
☐	Strong naming or code access security is used to restrict code access.
☐	Input is not trusted. Input is validated for type, range, format and length.
☐	Fields are private. Properties are used to expose fields.
☐	Properties are read-only unless write access is specifically required.
☐	Where appropriate, private default constructors are used to prevent object instantiation.
☐	Static constructors are private.

Strong Names

Check	Description
☐	If required, strong names are used
☐	Strong names are not relied upon to create tamper-proof assemblies.
☐	Delay signing is used to reduce the chance of private key compromise or to enable the use of a single public key across a team.
☐	In full trust scenarios, **StrongNameIdentityPermission** is not relied upon to restrict code that can call the assembly.

APTCA

Check	Description
☐	Except where necessary, APTCA usage is avoided.
☐	Assemblies marked with APTCA are subjected to thorough security code review.
☐	**SecurityTransparent** and **SecurityCritical** attributes are used appropriately.

Exception Management

Check	Description
☐	Structured exception handling is used instead of returning error codes.
☐	Sensitive data is not logged.
☐	System or sensitive application information is not revealed. Only generic error messages are returned to the end user.
☐	Code is not subject to exception filter issues where the filter higher in the call stack executes before code in a **finally** block.

☐	Where appropriate, an exception management system is used.
☐	Code fails early to avoid unnecessary processing.

File I/O

Check	Description
☐	Code avoids untrusted input for file names and file paths.
☐	If file names must be accepted through input, the names and locations are first validated.
☐	Security decisions are not based on user-supplied file names.
☐	Where possible, absolute file paths are used.
☐	Where appropriate, file I/O is constrained within the application's context.

Registry

Check	Description
☐	Sensitive data stored in HKEY_LOCAL_MACHINE is protected by ACLs.
☐	Sensitive data in the registry is encrypted.

Communication Security

Check	Description
☐	Transport-level encryption is used to protect secrets over the network. IPSec is used to protect the communication channel between two servers, and SSL is used for more granular channel protection for an application.

☐	Where appropriate, the **System.Net.Security.NegotiateStream** class is used for a TCP channel with .NET remoting.

Event Log

Check	Description
☐	Sensitive data is not logged in the event log.
☐	Event log data is not exposed to unauthorized users.

Data Access

Check	Description
☐	Connection strings are not hard coded. Connection strings are stored in configuration files.
☐	Connection strings are encrypted if they contain credentials.
☐	To prevent SQL injection, input is validated and parameterized stored procedures are used.

Delegates

Check	Description
☐	Delegates are not accepted from untrusted sources.
☐	Where appropriate, permissions to the delegate are restricted.

☐	Permissions are not asserted before delegate is called.

Serialization

Check	Description
☐	The **ISerializable** interface or the **NonSerialized** attribute are used to control serialization of sensitive data.
☐	Serialized data streams are validated when they are deserialized.

Threading

Check	Description
☐	Multithreaded code does not cache the results of security checks.
☐	Impersonation tokens are not lost; they flow to the newly created thread.
☐	Static class constructors are synchronized.
☐	**Dispose** methods are synchronized.

Reflection

Check	Description
☐	Full assembly names are used when **Activator.CreateInstance** loads add-ins.
☐	Separate, low-trust application domains are used for assemblies created with user input.
☐	Assemblies are not loaded dynamically based on user input for assembly or type names.
☐	Untrusted code does not use **Reflection.Emit** to create dynamic assemblies.

☐	Unless required, dynamic assemblies created by **Reflection.Emit** are not persisted.
☐	**Assembly.ReflectionOnlyLoadFrom** is used only if you need to inspect code.

Sensitive Data

Check	Description
☐	Where appropriate, **SecureString** is used rather than **System.String**.
☐	Secrets are held in memory for only a limited time.
☐	Protected configuration is used to protect sensitive data and secrets in configuration files.

Unmanaged Code

Check	Description
☐	Naming conventions are used (safe, native, unsafe) to identify unmanaged APIs.
☐	Unmanaged API calls are isolated in a wrapper assembly.
☐	String parameters that are passed to native code are constrained and validated to reduce the risk of buffer overrun, integer overflow, and other vulnerabilities.
☐	Array bounds are validated when an array is used to pass input to a native API.
☐	File path lengths are checked when a file name and path are passed to an unmanaged API.
☐	Unmanaged code is compiled with the **/GS** switch to enable stack probes.
☐	Unmanaged code is inspected for potentially dangerous APIs.
☐	Unmanaged types or handles are not exposed to partially trusted code.

☐	The **SuppressUnmanagedCode** attribute is used only if assembly takes precautions to ensure that malicious code cannot coerce it into performing unwanted operations.
☐	Pointers are held in private fields to prevent access violation or attempt to dereference them to gain access to sensitive information.

9. References:

For more information on above listed points please refer to the following links

Security Practices: .NET Framework 2.0 Security Practices at a Glance

Security Checklist: .NET Framework 2.0

Naming Guidelines

Secure Coding Guidelines for .NET Framework 1.1