Solutions for C# Programmers

2nd Edition
Updated for C# 2.0
and .NET 2.0

C# Cookbook™

O'REILLY®

Jay Hilyard & Stephen Teilhet

C# Cookbook™, Second Edition
by Jay Hilyard and Stephen Teilhet

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: John Osborn
Developmental Editor: Ralph Davis
Production Editor: Mary Brady
Copyeditor: Norma Emory
Proofreader: Genevieve Rajewski

Indexer: Ellen Troutman Zaig
Cover Designer: Emma Colby
Interior Designer: David Futato
Illustrators: Robert Romano, Jessamyn Read, and Lesley Borash

Printing History:

January 2004: First Edition.
January 2006: Second Edition.

ISBN: 0-596-10063-9
[M]

CHAPTER 17

Security

17.0 Introduction

There are many ways to secure different parts of your application. The security of running code in .NET revolves around the concept of Code Access Security (CAS). CAS determines the trustworthiness of an assembly based upon its origin and the characteristics of the assembly itself, such as its hash value. For example, code installed locally on the machine is more trusted than code downloaded from the Internet. The runtime will also validate an assembly's metadata and type safety before that code is allowed to run.

There are many ways to write secure code and protect data using the .NET Framework. In this chapter, we explore such things as controlling access to types, encryption and decryption, random numbers, securely storing data, and using programmatic and declarative security.

17.2 Encrypting and Decrypting a String

Problem

You have a string you want to be able to encrypt and decrypt—perhaps a password or software key—which will be stored in some form accessible by users, such as in a file, the registry, or even a field, that may be open to attack from malicious code.

Solution

Encrypting the string will prevent users from being able to read and decipher the information. The `CryptoString` class shown in Example 17-5 contains two static methods to encrypt and decrypt a string and two static properties to retrieve the generated key and inititialization vector (IV—a random number used as a starting point to encrypt data) after encryption has occurred.

Example 17-5. CryptoString class

```
using System;
using System.Security.Cryptography;

public sealed class CryptoString
{
    private CryptoString( ) {}

    private static byte[] savedKey = null;
    private static byte[] savedIV = null;

     public static byte[] Key
     {
      get { return savedKey; }
      set { savedKey = value; }
     }

     public static byte[] IV
     {
      get { return savedIV; }
      set { savedIV = value; }
     }

    private static void RdGenerateSecretKey(RijndaelManaged rdProvider)
    {
        if (savedKey == null)
        {
            rdProvider.KeySize = 256;
            rdProvider.GenerateKey( );
            savedKey = rdProvider.Key;
        }
    }

    private static void RdGenerateSecretInitVector(RijndaelManaged rdProvider)
    {
```

Example 17-5. CryptoString class (continued)

```
        if (savedIV == null)
        {
            rdProvider.GenerateIV( );
            savedIV = rdProvider.IV;
        }
    }

    public static string Encrypt(string originalStr)
    {
        // Encode data string to be stored in memory.
        byte[] originalStrAsBytes = Encoding.ASCII.GetBytes(originalStr);
        byte[] originalBytes = {};

        // Create MemoryStream to contain output.
        using (MemoryStream memStream = new
                 MemoryStream(originalStrAsBytes.Length))
        {
            using (RijndaelManaged rijndael = new RijndaelManaged( ))
            {
                // Generate and save secret key and init vector.
                RdGenerateSecretKey(rijndael);
                RdGenerateSecretInitVector(rijndael);

                if (savedKey == null || savedIV == null)
                {
                    throw (new NullReferenceException(
                            "savedKey and savedIV must be non-null."));
                }

                // Create encryptor and stream objects.
                using (ICryptoTransform rdTransform =
                       rijndael.CreateEncryptor((byte[])savedKey.
                       Clone( ),(byte[])savedIV.Clone( )))
                {
                    using (CryptoStream cryptoStream = new CryptoStream(memStream,
                          rdTransform, CryptoStreamMode.Write))
                    {
                        // Write encrypted data to the MemoryStream.
                        cryptoStream.Write(originalStrAsBytes, 0,
                                    originalStrAsBytes.Length);
                        cryptoStream.FlushFinalBlock( );
                        originalBytes = memStream.ToArray( );
                    }
                }
            }
        }

        // Convert encrypted string.
        string encryptedStr = Convert.ToBase64String(originalBytes);
        return (encryptedStr);
    }
```

Example 17-5. CryptoString class (continued)

```
    public static string Decrypt(string encryptedStr)
    {
        // Unconvert encrypted string.
        byte[] encryptedStrAsBytes = Convert.FromBase64String(encryptedStr);
        byte[] initialText = new Byte[encryptedStrAsBytes.Length];

        using (RijndaelManaged rijndael = new RijndaelManaged( ))
        {
            using (MemoryStream memStream = new MemoryStream(encryptedStrAsBytes))
            {
                if (savedKey == null || savedIV == null)
                {
                    throw (new NullReferenceException(
                            "savedKey and savedIV must be non-null."));
                }

                // Create decryptor, and stream objects.
                using (ICryptoTransform rdTransform =
                     rijndael.CreateDecryptor((byte[])savedKey.
                     Clone( ),(byte[])savedIV.Clone( )))
                {
                    using (CryptoStream cryptoStream = new CryptoStream(memStream,
                     rdTransform, CryptoStreamMode.Read))
                    {
                    // Read in decrypted string as a byte[].
                    cryptoStream.Read(initialText, 0, initialText.Length);
                    }
                }
            }
        }

        // Convert byte[] to string.
        string decryptedStr = Encoding.ASCII.GetString(initialText);
        return (decryptedStr);
    }
}
```

Discussion

The `CryptoString` class contains only static members, except for the private instance constructor, which prevents anyone from directly creating an object from this class.

This class uses the *Rijndael algorithm* to encrypt and decrypt a string. This algorithm is found in the `System.Security.Cryptography.RijndaelManaged` class. This algorithm requires a secret key and an initialization vector; both are `byte` arrays. A random secret key can be generated for you by calling the `GenerateKey` method on the `RijndaelManaged` class. This method accepts no parameters and returns `void`. The generated key is placed in the `Key` property of the `RijndaelManaged` class. The

GenerateIV method generates a random initialization vector and places this vector in the IV property of the RijndaelManaged class.

The byte array values in the Key and IV properties must be stored for later use and not modified. This is due to the nature of private-key encryption classes, such as RijndaelManaged. The Key and IV values must be used by both the encryption and decryption routines to successfully encrypt and decrypt data.

The SavedKey and SavedIV private static fields contain the secret key and initialization vector, respectively. The secret key is used by both the encryption and decryption methods to encrypt and decrypt data. This is why there are public properties for these values, so they can be stored somewhere secure for later use. This means that any strings encrypted by this object must be decrypted by this object. The initialization vector is used to prevent anyone from attempting to decipher the secret key.

Two methods in the CryptoString class, RdGenerateSecretKey and RdGenerateSecretInitVector, are used to generate a secret key and initialization vector when none exists. The RdGenerateSecretKey method generates the secret key, which is placed in the SavedKey field. Likewise, the RdGenerateSecretInitVector generates the initialization vector, which is placed in the SavedIV field. There is only one key and one IV generated for this class. This enables the encryption and decryption routines to have access to the same key and IV information at all times.

The Encrypt and Decrypt methods of the CryptoString class do the actual work of encrypting and decrypting a string. The Encrypt method accepts a string that you want to encrypt and returns an encrypted string. The following code calls this method and passes in a string to be encrypted:

```
string encryptedString = CryptoString.Encrypt("MyPassword");
Console.WriteLine("encryptedString: {0}", encryptedString);
// Get the key and IV used so you can decrypt it later.
byte [] key = CryptoString.Key;
byte [] IV = CryptoString.IV;
```

Once the string is encrypted, the key and IV are stored for later decryption. This method displays:

```
encryptedString: Ah4vkmVKpwMYRT97Q8cVgQ==
```

Note that your output may differ since you will be using a different key and IV value. The following code sets the key and IV used to encrypt the string, then calls the Decrypt method to decrypt the previously encrypted string:

```
CryptoString.Key = key;
CryptoString.IV = IV;
string decryptedString = CryptoString.Decrypt(encryptedString);
Console.WriteLine("decryptedString: {0}", decryptedString);
```

This method displays:

```
decryptedString: MyPassword
```

There does not seem to be any problem with using escape sequences such as `\r`, `\n`, `\r\n`, or `\t` in the string to be encrypted. In addition, using a quoted string literal, with or without escaped characters, works without a problem:

```
@"MyPassword"
```

See Also

See Recipe 17.3; see the "System.Cryptography Namespace," "MemoryStream Class," "ICryptoTransform Interface," and "RijndaelManaged Class" topics in the MSDN documentation.

17.4 Cleaning up Cryptography Information

Problem

You will be using the cryptography classes in the FCL to encrypt and/or decrypt data. In doing so, you want to make sure that no data (e.g., seed values or keys) is left in memory for longer than you are using the cryptography classes. Hackers can sometimes find this information in memory and use it to break your encryption or, worse, to break your encryption, modify the data, and then reencrypt the data and pass it on to your application.

Solution

In order to clear out the key and initialization vector (or seed), you need to call the `Clear` method on whichever `SymmetricAlgorithm`- or `AsymmetricAlgorithm`-derived class you are using. `Clear` reinitializes the `Key` and `IV` properties, preventing them from being found in memory. This is done after saving the key and IV so that you can decrypt later. Example 17-7 encodes a string, then cleans up immediately afterward to provide the smallest window possible for potential attackers.

Example 17-7. Cleaning up cryptography information

```
using System;
using System.Text;
using System.IO;
using System.Security.Cryptography;

string originalStr = "SuperSecret information";
// Encode data string to be stored in memory.
byte[] originalStrAsBytes = Encoding.ASCII.GetBytes(originalStr);

// Create MemoryStream to contain output.
MemoryStream memStream = new MemoryStream(originalStrAsBytes.Length);

RijndaelManaged rijndael = new RijndaelManaged( );

// Generate secret key and init vector.
rijndael.KeySize = 256;
rijndael.GenerateKey( );
rijndael.GenerateIV( );

// Save the key and IV for later decryption.
byte [] key = rijndael.Key;
byte [] IV = rijndael.IV;

// Create encryptor, and stream objects.
ICryptoTransform transform = rijndael.CreateEncryptor(rijndael.Key,
    rijndael.IV);
CryptoStream cryptoStream = new CryptoStream(memStream, transform,
    CryptoStreamMode.Write);
```

Example 17-7. Cleaning up cryptography information (continued)

```
// Write encrypted data to the MemoryStream.
cryptoStream.Write(originalStrAsBytes, 0, originalStrAsBytes.Length);
cryptoStream.FlushFinalBlock( );

// Release all resources as soon as we are done with them
// to prevent retaining any information in memory.
memStream.Close( );
cryptoStream.Close( );
transform.Dispose( );
// This clear statement regens both the key and the init vector so that
// what is left in memory is no longer the values you used to encrypt with.
rijndael.Clear( );
```

You can also make your life a little easier by taking advantage of the `using` statement, instead of having to remember to manually call each of the `Close` methods individually. This code block shows how to use the `using` statement:

```
public static void CleanUpCryptoWithUsing( )
{
    string originalStr = "SuperSecret information";
    // Encode data string to be stored in memory.
    byte[] originalStrAsBytes = Encoding.ASCII.GetBytes(originalStr);
    byte[] originalBytes = { };

    // Create MemoryStream to contain output.
    using (MemoryStream memStream = new MemoryStream(originalStrAsBytes.Length))
    {
        using (RijndaelManaged rijndael = new RijndaelManaged( ))
        {
            // Generate secret key and init vector.
            rijndael.KeySize = 256;
            rijndael.GenerateKey( );
            rijndael.GenerateIV( );

            // Save off the key and IV for later decryption.
            byte[] key = rijndael.Key;
            byte[] IV = rijndael.IV;

            // Create encryptor, and stream objects.
            using (ICryptoTransform transform =
                rijndael.CreateEncryptor(rijndael.Key, rijndael.IV))
            {
                using (CryptoStream cryptoStream = new
                       CryptoStream(memStream, transform,
                        CryptoStreamMode.Write))
                {
                    // Write encrypted data to the MemoryStream.
                    cryptoStream.Write(originalStrAsBytes, 0,
                            originalStrAsBytes.Length);
                    cryptoStream.FlushFinalBlock( );
                }
```

```
            }
        }
    }
}
```

Discussion

To make sure your data is safe, you need to close the `MemoryStream` and `CryptoStream` objects as soon as possible, as well as calling `Dispose` on the `ICryptoTransform` implementation to clear out any resources used in this encryption. The `using` statement makes this process much easier, makes your code easier to read, and leads to fewer programming mistakes.

See Also

See the "SymmetricAlgorithm.Clear Method" and "AsymmetricAlgorithm.Clear Method" topics in the MSDN documentation.

17.7 A Better Random Number Generator

Problem

You need a random number with which to generate items such as a sequence of session keys. The random number must be as unpredictable as possible so that the likelihood of predicting the sequence of keys is as low as possible.

Solution

Use the class `System.Security.Cryptography.RNGCryptoServiceProvider`.

The `RNGCryptoServiceProvider` is used to populate a random byte array using the `GetBytes` method that is then printed out as a string in the following example:

```
public static void BetterRandomString( )
{
    // Create a stronger hash code using RNGCryptoServiceProvider.
    byte[] random = new byte[64];
    RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider( );
    // Populate with random bytes.
    rng.GetBytes(random);

    // Convert random bytes to string.
    string randomBase64 = Convert.ToBase64String(random);
    // Display.
    Console.WriteLine("Random string: {0} ",randomBase64);
}
```

The output of this method is shown here:

```
Random string:
xDNitrreUpMmlO7Opd6AFvMC8VIG9+sAGfyvdZr2lEY1M3n2v3Ap4JIkYfJWW+sZaJjJMxj475VlVQFoRKvFI
g==
```

Discussion

`Random` provides methods like `Next`, `NextBytes`, and `NextDouble` to generate random information for integers, arrays of bytes, and doubles, respectively. These methods can produce a moderate level of unpredictability, but to truly generate a more unpredictable random series, you need to use the `RNGCryptoServiceProvider`.

`RNGCryptoServiceProvider` can be customized to use any of the underlying Win32 Crypto API providers. You pass a `CspParameters` class in the constructor to determine exactly which provider is responsible for generating the random byte sequence. `CspParameters` allows you to customize items such as the key container name, the provider type code, the provider name, and the key number used.

The `GetBytes` method populates the entire length of the byte array with random bytes.

See Also

See the "RNGCryptoServiceProvider Class," "CspParameters Class," and "Cryptographic Provider Types" topics in the MSDN documentation.

17.9 Making a Security Assert Safe

Problem

You want to assert that at a particular point in the call stack, a given permission is available for all subsequent calls. However, doing this can easily open a security hole to allow other malicious code to spoof your code or to create a back door into your component. You want to assert a given security permission, but you want to do so in a secure and efficient manner.

Solution

In order to make this approach secure, you need to call `Demand` on the permissions that the subsequent calls need. This makes sure that code that doesn't have these permissions can't slip by due to the `Assert`. The `Demand` is done to ensure that you have indeed been granted this permission before using the `Assert` to short-circuit the stackwalk. This is demonstrated by the function `CallSecureFunctionSafelyAndEfficiently`, which performs a `Demand` and an `Assert` before calling `SecureFunction`, which in turn does a `Demand` for a `ReflectionPermission`.

The code listing for `CallSecureFunctionSafelyAndEfficiently` is shown in Example 17-14.

Example 17-14. CallSecureFunctionSafelyAndEfficiently function

```
public static void CallSecureFunctionSafelyAndEfficiently( )
{

    // Set up a permission to be able to access nonpublic members
    // via reflection.
    ReflectionPermission perm =
        new ReflectionPermission(ReflectionPermissionFlag.MemberAccess);

    // Demand the permission set we have compiled before using Assert
    // to make sure we have the right before we Assert it.  We do
    // the Demand to ensure that we have checked for this permission
    // before using Assert to short-circuit stackwalking for it, which
    // helps us stay secure, while performing better.
    perm.Demand( );

    // Assert this right before calling into the function that
    // would also perform the Demand to short-circuit the stack walk
    // each call would generate.  The Assert helps us to optimize
    // our use of SecureFunction.
    perm.Assert( );

    // We call the secure function 100 times but only generate
    // the stackwalk from the function to this calling function
    // instead of walking the whole stack 100 times.
    for(int i=0;i<100;i++)
    {
```

Example 17-14. CallSecureFunctionSafelyAndEfficiently function (continued)

```
        SecureFunction( );
    }
}
```

The code listing for SecureFunction is shown here:

```
public static void SecureFunction( )
{
    // Set up a permission to be able to access nonpublic members
    // via reflection.
    ReflectionPermission perm =
        new ReflectionPermission(ReflectionPermissionFlag.MemberAccess);

    // Demand the right to do this and cause a stackwalk.
    perm.Demand( );

    // Perform the action here...
}
```

Discussion

In the demonstration function CallSecureFunctionSafelyAndEfficiently, the function you are calling (SecureFunction) performs a Demand on a ReflectionPermission to ensure that the code can access nonpublic members of classes via reflection. Normally, this would result in a stackwalk for every call to SecureFunction. The Demand in CallSecureFunctionSafelyAndEfficiently is there only to protect against the usage of the Assert in the first place. To make this more efficient, you can use Assert to state that all functions issuing Demands that are called from this one do not have to stackwalk any further. The Assert says stop checking for this permission in the call stack. In order to do this, you need the permission to call Assert.

The problem comes in with this Assert as it opens up a potential luring attack where SecureFunction is called via CallSecureFunctionSafelyAndEfficiently, which calls Assert to stop the Demand stackwalks from SecureFunction. If unauthorized code without this ReflectionPermission were able to call CallSecureFunctionSafelyAndEfficiently, the Assert would prevent the SecureFunction Demand call from determining that there is some code in the call stack without the proper rights. This is the power of the call stack checking in the CLR when a Demand occurs.

In order to protect against this, you issue a Demand for the ReflectionPermission needed by SecureFunction in CallSecureFunctionSafelyAndEfficiently to close this hole before issuing the Assert. The combination of this Demand and the Assert causes you to do one stack walk instead of the original 100 that would have been caused by the Demand in SecureFunction but to still maintain secure access to this functionality.

Security optimization techniques, such as using Assert in this case (even though it isn't the primary reason to use Assert), can help class library as well as control developers who are trusted to perform Asserts in order to speed the interaction of their

code with the CLR; but if used improperly, these techniques can also open up holes in the security picture. This example shows that you can have both performance and security where secure access is concerned.

If you are using `Assert`, be mindful that stackwalk overrides should never be made in a class constructor. Constructors are not guaranteed to have any particular security context, nor are they guaranteed to execute at a specific point in time. This lack leads to the call stack not being well defined, and `Assert` used here can produce unexpected results.

One other thing to remember with `Assert` is that you can have only one active `Assert` in a function at a given time. If you `Assert` the same permission twice, a `SecurityException` is thrown by the CLR. You must revert the original `Assert` first using `RevertAssert`. Then you can declare the second `Assert`.

See Also

See the "CodeAccessSecurity.Assert Method," "CodeAccessSecurity.Demand Method," "CodeAccessSecurity.RevertAssert Method," and "Overriding Security Checks" topics in the MSDN documentation.

17.12 Minimizing the Attack Surface of an Assembly

Problem

Someone attacking your assembly will first attempt to find out as many things as possible about your assembly and then use this information in constructing the attack(s). The more surface area you give to attackers, the more they have to work with. You need to minimize what your assembly is allowed to do so that, if an attacker is successful in taking it over, the attacker will not have the necessary privileges to do any damage to the system.

Solution

Use the `SecurityAction.RequestRefuse` enumeration member to indicate, at an assembly level, the permissions that you do not wish this assembly to have. This will force the CLR to refuse these permissions to your code and will ensure that, even if another part of the system is compromised, your code cannot be used to perform functions that it does not need the rights to do.

The following example allows the assembly to perform file I/O as part of its minimal permission set but explicitly refuses to allow this assembly to have permissions to skip verification:

```
[assembly: FileIOPermission(SecurityAction.RequestMinimum,Unrestricted=true)]
[assembly: SecurityPermission(SecurityAction.RequestRefuse,
             SkipVerification=false)]
```

Discussion

Once you have determined what permissions your assembly needs as part of your normal security testing, you can use `RequestRefuse` to lock down your code. If this seems extreme, think of scenarios in which your code could be accessing a data store containing sensitive information, such as Social Security numbers or salary information. This proactive step can help you show your customers that you take security seriously and can help defend your interests in case a break-in occurs on a system that your code is part of.

One serious consideration with this approach is that the use of `RequestRefuse` marks your assembly as partially trusted. This in turn prevents it from calling any strong-named assembly that hasn't been marked with the `AllowPartiallyTrustedCallers` attribute.

See Also

See Chapter 8 of Microsoft Patterns & Practices Group: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh08.asp*; see the "SecurityAction Enumeration" and "Global Attributes" topics in the MSDN documentation.

17.13 Obtaining Security/Audit Information

Problem

You need to obtain the security rights and/or audit information for a file or registry key.

Solution

When obtaining security/audit information for a file, use the static `GetAccessControl` method of the `File` class to obtain a `System.Security.AccessControl.FileSecurity` object. Use the `FileSecurity` object to access the security and audit information for the file. These steps are demonstrated in Example 17-15.

Example 17-15. Obtaining security audit information

```
public static void ViewFileRights( )
{
    // Get security information from a file.
    string file = @"c:\FOO.TXT";
    FileSecurity fileSec = File.GetAccessControl(file);
    DisplayFileSecurityInfo(fileSec);
}

public static void DisplayFileSecurityInfo(FileSecurity fileSec)
{
    Console.WriteLine("GetSecurityDescriptorSddlForm:  {0}",
        fileSec.GetSecurityDescriptorSddlForm(AccessControlSections.All));

    foreach (FileSystemAccessRule ace in
            fileSec.GetAccessRules(true, true, typeof(NTAccount)))
    {
        Console.WriteLine("\tIdentityReference.Value: {0}",
                          ace.IdentityReference.Value);
        Console.WriteLine("\tAccessControlType: {0}", ace.AccessControlType);
        Console.WriteLine("\tFileSystemRights: {0}", ace.FileSystemRights);
        Console.WriteLine("\tInheritanceFlags: {0}", ace.InheritanceFlags);
        Console.WriteLine("\tIsInherited: {0}", ace.IsInherited);
        Console.WriteLine("\tPropagationFlags: {0}", ace.PropagationFlags);

        Console.WriteLine("-----------------\r\n\r\n");
    }


    foreach (FileSystemAuditRule ace in
            fileSec.GetAuditRules(true, true, typeof(NTAccount)))
    {
        Console.WriteLine("\tIdentityReference.Value: {0}",
                          ace.IdentityReference.Value);
        Console.WriteLine("\tAuditFlags: {0}", ace.AuditFlags);
        Console.WriteLine("\tFileSystemRights: {0}", ace.FileSystemRights);
        Console.WriteLine("\tInheritanceFlags: {0}", ace.InheritanceFlags);
```

Example 17-15. Obtaining security audit information (continued)

```
        Console.WriteLine("\tIsInherited: {0}", ace.IsInherited);
        Console.WriteLine("\tPropagationFlags: {0}", ace.PropagationFlags);

        Console.WriteLine("-----------------\r\n\r\n");
    }

    Console.WriteLine("GetGroup(typeof(NTAccount)).Value: {0}",
                      fileSec.GetGroup(typeof(NTAccount)).Value);
    Console.WriteLine("GetOwner(typeof(NTAccount)).Value: {0}",
                      fileSec.GetOwner(typeof(NTAccount)).Value);

    Console.WriteLine("---------------------------------------\r\n\r\n\r\n");
}
```

These methods produce the following output:

```
    GetSecurityDescriptorSddlForm:  O:BAG:SYD:PAI(A;;FA;;;SY)(A;;FA;;;BA)
        IdentityReference.Value: NT AUTHORITY\SYSTEM
        AccessControlType: Allow
        FileSystemRights: FullControl
        InheritanceFlags: None
        IsInherited: False
        PropagationFlags: None
    -----------------


        IdentityReference.Value: BUILTIN\Administrators
        AccessControlType: Allow
        FileSystemRights: FullControl
        InheritanceFlags: None
        IsInherited: False
        PropagationFlags: None
    -----------------


    GetGroup(typeof(NTAccount)).Value: NT AUTHORITY\SYSTEM
    GetOwner(typeof(NTAccount)).Value: BUILTIN\Administrators
    ---------------------------------------
```

When obtaining security/audit information for a registry key, use the `GetAccessControl` instance method of the `Microsoft.Win32.RegistryKey` class to obtain a `System.Security.AccessControl.RegistrySecurity` object. Use the `RegistrySecurity` object to access the security and audit information for the registry key. These steps are demonstrated in Example 17-16.

Example 17-16. Getting security or audit information for a registry key

```
public static void ViewRegKeyRights( )
{
    // Get security information from a registry key.
    using (RegistryKey regKey =
        Registry.LocalMachine.OpenSubKey(@"SOFTWARE\MyCompany\MyApp"))
    {
```

Example 17-16. Getting security or audit information for a registry key (continued)

```
        RegistrySecurity regSecurity = regKey.GetAccessControl( );
        DisplayRegKeySecurityInfo(regSecurity);
    }
}


public static void DisplayRegKeySecurityInfo(RegistrySecurity regSec)
{
    Console.WriteLine("GetSecurityDescriptorSddlForm:  {0}",
        regSec.GetSecurityDescriptorSddlForm(AccessControlSections.All));

    foreach (RegistryAccessRule ace in
            regSec.GetAccessRules(true, true, typeof(NTAccount)))
    {
        Console.WriteLine("\tIdentityReference.Value: {0}",
                          ace.IdentityReference.Value);
        Console.WriteLine("\tAccessControlType: {0}", ace.AccessControlType);
        Console.WriteLine("\tRegistryRights: {0}", ace.RegistryRights.ToString( ));
        Console.WriteLine("\tInheritanceFlags: {0}", ace.InheritanceFlags);
        Console.WriteLine("\tIsInherited: {0}", ace.IsInherited);
        Console.WriteLine("\tPropagationFlags: {0}", ace.PropagationFlags);

        Console.WriteLine("-----------------\r\n\r\n");
    }


    foreach (RegistryAuditRule ace in
            regSec.GetAuditRules(true, true, typeof(NTAccount)))
    {
        Console.WriteLine("\tIdentityReference.Value: {0}",
                          ace.IdentityReference.Value);
        Console.WriteLine("\tAuditFlags: {0}", ace.AuditFlags);
        Console.WriteLine("\tRegistryRights: {0}", ace.RegistryRights.ToString( ));
        Console.WriteLine("\tInheritanceFlags: {0}", ace.InheritanceFlags);
        Console.WriteLine("\tIsInherited: {0}", ace.IsInherited);
        Console.WriteLine("\tPropagationFlags: {0}", ace.PropagationFlags);

        Console.WriteLine("-----------------\r\n\r\n");
    }

    Console.WriteLine("GetGroup(typeof(NTAccount)).Value: {0}",
                      regSec.GetGroup(typeof(NTAccount)).Value);
    Console.WriteLine("GetOwner(typeof(NTAccount)).Value: {0}",
                      regSec.GetOwner(typeof(NTAccount)).Value);

    Console.WriteLine("-------------------------------------\r\n\r\n\r\n");
}
```

These methods produce the following output:

```
GetSecurityDescriptorSddlForm:  O:S-1-5-21-329068152-1383384898-682003330-1004G:S-1-
5-21-329068152-1383384898-682003330-513D:
AI(A;ID;KR;;;BU)(A;CIIOID;GR;;;BU)(A;ID;KA;;;BA)(A;CIIOID;GA;;;BA)(A;ID;KA;;;SY)(A;CI
IOID;GA;;;SY)(A;ID;KA;;;S-1-5-21-329068152-1383384898-682003330-
1004)(A;CIIOID;GA;;;CO)
    IdentityReference.Value: BUILTIN\Users
    AccessControlType: Allow
    RegistryRights: ReadKey
    InheritanceFlags: None
    IsInherited: True
    PropagationFlags: None
-----------------


    IdentityReference.Value: BUILTIN\Users
    AccessControlType: Allow
    RegistryRights: -2147483648
    InheritanceFlags: ContainerInherit
    IsInherited: True
    PropagationFlags: InheritOnly
-----------------


    IdentityReference.Value: BUILTIN\Administrators
    AccessControlType: Allow
    RegistryRights: FullControl
    InheritanceFlags: None
    IsInherited: True
    PropagationFlags: None
-----------------


    IdentityReference.Value: BUILTIN\Administrators
    AccessControlType: Allow
    RegistryRights: 268435456
    InheritanceFlags: ContainerInherit
    IsInherited: True
    PropagationFlags: InheritOnly
-----------------


    IdentityReference.Value: NT AUTHORITY\SYSTEM
    AccessControlType: Allow
    RegistryRights: FullControl
    InheritanceFlags: None
    IsInherited: True
    PropagationFlags: None
-----------------


    IdentityReference.Value: NT AUTHORITY\SYSTEM
    AccessControlType: Allow
    RegistryRights: 268435456
    InheritanceFlags: ContainerInherit
```

```
    IsInherited: True
    PropagationFlags: InheritOnly
-----------------


    IdentityReference.Value: OPERATOR-C1EFE0\Admin
    AccessControlType: Allow
    RegistryRights: FullControl
    InheritanceFlags: None
    IsInherited: True
    PropagationFlags: None
-----------------


    IdentityReference.Value: CREATOR OWNER
    AccessControlType: Allow
    RegistryRights: 268435456
    InheritanceFlags: ContainerInherit
    IsInherited: True
    PropagationFlags: InheritOnly
-----------------


GetGroup(typeof(NTAccount)).Value: OPERATOR-C1EFE0\None
GetOwner(typeof(NTAccount)).Value: OPERATOR-C1EFE0\Admin
---------------------------------------
```

Discussion

The essential method that is used to obtain the security information for a file or registry key is the `GetAccessControl` method. When this method is called on the `RegistryKey` object, a `RegistrySecurity` object is returned. However, when this method is called on a `File` class, a `FileSecurity` object is returned. The `RegistrySecurity` and `FileSecurity` objects essentially represent a Discretionary Access Control List (DACL), which is what developers writing code in unmanaged languages such as C++ are used to working with.

The `RegistrySecurity` and `FileSecurity` objects each contains a list of security rules that has been applied to the system object that it represents. The `RegistrySecurity` object contains a list of `RegistryAccessRule` objects, and the `FileSecurity` object contains a list of `FileSystemAccessRule` objects. These rule objects are the equivalent of the Access Control Entries (ACE) that make up the list of security rules within a DACL.

System objects other than just the `File` class and `RegistryKey` object allow security privileges to be queried. Table 17-1 lists all the .NET Framework classes that return a security object type and what that type is. In addition, the rule-object type that is contained in the security object is also listed.

*Table 17-1. List of all *Security and *AccessRule objects and the types to which they apply*

Class	Object returned by the GetAccessControl method	Rule-object type contained within the security object
`Directory`	`DirectorySecurity`	`FileSystemAccessRule`
`DirectoryInfo`	`DirectorySecurity`	`FileSystemAccessRule`
`EventWaitHandle`	`EventWaitHandleSecurity`	`EventWaitHandleAccessRule`
`File`	`FileSecurity`	`FileSystemAccessRule`
`FileInfo`	`FileSecurity`	`FileSystemAccessRule`
`FileStream`	`FileSecurity`	`FileSystemAccessRule`
`Mutex`	`MutexSecurity`	`MutexAccessRule`
`RegistryKey`	`RegistrySecurity`	`RegistryAccessRule`
`Semaphore`	`SemaphoreSecurity`	`SemaphoreAccessRule`

The abstraction of a system object's DACL through the `*Security` objects and the abstraction of a DACL's ACE through the `*AccessRule` objects allows easy access to the security privileges of that system object. In previous versions of the .NET Framework, these DACLs and their ACEs would have been accessible only in unmanaged code. With the latest .NET Framework, you now have access to view and program these objects.

See Also

See Recipe 17.14; see the "System.IO.File.GetAccessControl Method," "System.Security.AccessControl.FileSecurity Class," "Microsoft.Win32.RegistryKey.GetAccessControl Method," and "System.Security.AccessControl.RegistrySecurity Class" topics in the MSDN documentation.

17.14 Granting/Revoking Access to a File or Registry Key

Problem

You need to change the security privileges of either a file or registry key programmatically.

Solution

The code shown in Example 17-17 grants and then revokes the ability to perform write actions on a registry key.

Example 17-17. Granting and revoking the right to perform write actions on a registry key

```
public static void GrantRevokeRegKeyRights( )
{
    NTAccount user = new NTAccount(@"WRKSTN\ST");

    using (RegistryKey regKey = Registry.LocalMachine.OpenSubKey(
                            @"SOFTWARE\MyCompany\MyApp"))
    {
        GrantRegKeyRights(regKey, user, RegistryRights.WriteKey,
           InheritanceFlags.None, PropagationFlags.None, AccessControlType.Allow);
        RevokeRegKeyRights(regKey, user, RegistryRights.WriteKey,
                        InheritanceFlags.None, PropagationFlags.None,
                        AccessControlType.Allow)
    }
}

public static void GrantRegKeyRights(RegistryKey regKey,
                                     NTAccount user,
                                     RegistryRights rightsFlags,
                                     InheritanceFlags inherFlags,
                                     PropagationFlags propFlags,
                                     AccessControlType actFlags)
{
    RegistrySecurity regSecurity = regKey.GetAccessControl( );

    RegistryAccessRule rule = new RegistryAccessRule(user, rightsFlags, inherFlags,
                                                     propFlags, actFlags);
    regSecurity.AddAccessRule(rule);
    regKey.SetAccessControl(regSecurity);
}

public static void RevokeRegKeyRights(RegistryKey regKey,
                                      NTAccount user,
                                      RegistryRights rightsFlags,
                                      InheritanceFlags inherFlags,
                                      PropagationFlags propFlags,
                                      AccessControlType actFlags)
{
    RegistrySecurity regSecurity = regKey.GetAccessControl( );

    RegistryAccessRule rule = new RegistryAccessRule(user, rightsFlags, inherFlags,
                                                     propFlags, actFlags);
    regSecurity.RemoveAccessRuleSpecific(rule);

    regKey.SetAccessControl(regSecurity);
}
```

The code shown in Example 17-18 grants and then revokes the ability to delete a file.

Example 17-18. Granting and revoking the right to delete a file

```
public static void GrantRevokeFileRights( )
{
```

Example 17-18. Granting and revoking the right to delete a file (continued)

```
    NTAccount user = new NTAccount(@"WRKSTN\ST");

    string file = @"c:\FOO.TXT";
    GrantFileRights(file, user, FileSystemRights.Delete, InheritanceFlags.None,
                    PropagationFlags.None, AccessControlType.Allow);
    RevokeFileRights(file, user, FileSystemRights.Delete, InheritanceFlags.None,
                     PropagationFlags.None, AccessControlType.Allow);
}

public static void GrantFileRights(string file,
                                   NTAccount user,
                                   FileSystemRights rightsFlags,
                                   InheritanceFlags inherFlags,
                                   PropagationFlags propFlags,
                                   AccessControlType actFlags)
{
    FileSecurity fileSecurity = File.GetAccessControl(file);

    FileSystemAccessRule rule = new FileSystemAccessRule(user, rightsFlags,
                                                         inherFlags, propFlags,
                                                         actFlags);
    fileSecurity.AddAccessRule(rule);
    File.SetAccessControl(file, fileSecurity);
}

public static void RevokeFileRights(string file,
                                    NTAccount user,
                                    FileSystemRights rightsFlags,
                                    InheritanceFlags inherFlags,
                                    PropagationFlags propFlags,
                                    AccessControlType actFlags)
{
    FileSecurity fileSecurity = File.GetAccessControl(file);

    FileSystemAccessRule rule = new FileSystemAccessRule(user, rightsFlags,
                                                         inherFlags, propFlags,
                                                         actFlags);
    fileSecurity.RemoveAccessRuleSpecific(rule);
    File.SetAccessControl(file, fileSecurity);
}
```

Discussion

When granting or revoking access rights on a file or registry key, you need two things. The first is a valid `NTAccount` object. This object essentially encapsulates a user or group account. A valid `NTAccount` object is required in order to create either a new `RegistryAccessRule` or a new `FileSystemAccessRule`. The `NTAccount` identifies the user or group this access rule will apply to. Note that the string passed in to the `NTAccount` constructor must be changed to a valid user or group name that exists on your machine. If you pass in the name of an existing user or group account that has

been disabled, an `IdentityNotMappedException` will be thrown with the message "Some or all identity references could not be translated."

The second item that is needed is either a valid `RegistryKey` object, if you are modifying security access to a registry key or a string containing a valid path and filename to an existing file. These objects will have security permissions either granted to them or revoked from them.

Once these two items have been obtained, you can use the second item to obtain a security object, which contains the list of access-rule objects. For example, the following code obtains the security object for the registry key HKEY-LOCAL_MACHINE\SOFTWARE\MyCompany\MyApp:

```
RegistryKey regKey = Registry.LocalMachine.OpenSubKey(
                            @"SOFTWARE\MyCompany\MyApp");
RegistrySecurity regSecurity = regKey.GetAccessControl( );
```

The following code obtains the security object for the *FOO.TXT* file:

```
string file = @"c:\FOO.TXT";
FileSecurity fileSecurity = File.GetAccessControl(file);
```

Now that you have your particular security object, you can create an access-rule object that will be added to this security object. To do this, you need to create a new access rule. For a registry key, you have to create a new `RegistryAccessRule` object, and for a file, you have to create a new `FileSystemAccessRule` object. To add this access rule to the correct security object, you call the `SetAccessControl` method on the security object. Note that `RegistryAccessRule` objects can be added only to `RegistrySecurity` objects and `FileSystemAccessRule` objects can be added only to `FileSecurity` objects.

To remove an access-rule object from a system object, you follow the same set of steps, except that you call the `RemoveAccessRuleSpecific` method instead of `AddAccessRule`. `RemoveAccessRuleSpecific` accepts an access-rule object and attempts to remove the rule that exactly matches this rule object from the security object. As always, you must remember to call the `SetAccessControl` method to apply any changes to the actual system object.

For a list of other classes that allow security permissions to be modified programmatically, see Recipe 17.13.

See Also

See Recipe 17.13; see the "System.IO.File.GetAccessControl Method," "System.Security.AccessControl.FileSecurity Class," "System.Security.AccessControl.FileSystemAccessRule Class," "Microsoft.Win32.RegistryKey.GetAccessControl Method," "System.Security.AccessControl.RegistrySecurity Class," and "System.Security.AccessControl.RegistryAccessRule Class" topics in the MSDN documentation.

17.15 Protecting String Data with Secure Strings

Problem

You need to store sensitive information, such as a Social Security number, in a string. However, you do not want prying eyes to be able to view this data in memory.

Solution

Use the `SecureString` object. To place text from a stream object within a `SecureString` object, use the following method:

```
public static SecureString CreateSecureString(StreamReader secretStream)
{
    SecureString secretStr = new SecureString( );
    char buf;

    while (secretStream.Peek( ) >= 0)
    {
        buf = (char)secretStream.Read( );
        secretStr.AppendChar(buf);
    }

    // Make the secretStr object read-only.
    secretStr.MakeReadOnly( );

    return (secretStr);
}
```

To pull the text out of a `SecureString` object, use the following method:

```
public static void ReadSecureString(SecureString secretStr)
{
    // In order to read back the string, you need to use some special methods.
    IntPtr secretStrPtr = Marshal.SecureStringToBSTR(secretStr);
    string nonSecureStr = Marshal.PtrToStringBSTR(secretStrPtr);

    // Use the unprotected string.
    Console.WriteLine("nonSecureStr = {0}", nonSecureStr);

    Marshal.ZeroFreeBSTR(secretStrPtr);

    if (!secretStr.IsReadOnly( ))
    {
        secretStr.Clear( );
    }
}
```

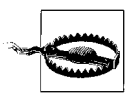
Discussion

A `SecureString` object is designed specifically to contain string data that you want to keep secret. Some of the data you may want to store in a `SecureString` object would

be a Social Security number, a credit card number, a PIN number, a password, an employee ID, or any other type of sensitive information.

This string data is automatically encrypted immediately upon being added to the `SecureString` object, and it is automatically decrypted when the string data is extracted from the `SecureString` object. The encryption is one of the highlights of using this object. In addition to encryption, there will be only one copy of a `SecureString` object in memory at any one time. This is in direct contrast to a `String` object, which creates multiple copies in memory whenever the text in the `String` object is modified.

Another feature of a `SecureString` object is that when the `MakeReadOnly` method is called, the `SecureString` becomes immutable. Any attempt to modify the string data within the read-only `SecureString` object causes an `InvalidOperationException` to be thrown. Once a `SecureString` object is made read-only, it cannot go back to a read/write state. However, you need to be careful when calling the `Copy` method on an existing `SecureString` object. This method will create a new instance of the `SecureString` object on which it was called, with a copy of its data. However, this new `SecureString` object is now readable and writable. You should review your code to determine if this new `SecureString` object should be made read-only similarly to its original `SecureString` object.



The `SecureString` object can be used only on Windows 2000 (with Service Pack 3 or greater) or later operating systems.

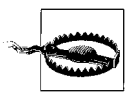
In this recipe you create a `SecureString` object from data read in from a stream. This data could also come from a `char*` using unsafe code. The `SecureString` object contains a constructor that accepts a parameter of this type in addition to an integer parameter that takes a length value, which determines the number of characters to pull from the `char*`.

Getting data out of a `SecureString` object is not obvious at first glance. There are no methods to return the data contained within a `SecureString` object. In order to accomplish this, you must use two static methods on the `Marshal` class. The first is the `SecureStringToBSTR`, which accepts your `SecureString` object and returns an `IntPtr`. This `IntPtr` is then passed into the `PtrToStringBSTR` method, also on the `Marshal` class. The `PtrToStringBSTR` method then returns an unsecure `String` object containing your decrypted string data.

Once you are done using the `SecureString` object, you should call the static `ZeroFreeBSTR` method on the `Marshal` class to zero out any memory allocated when extracting the data from the `SecureStirng`. As an added safeguard, you should call the `Clear` method of the `SecureString` object to zero out the encrypted string from memory. If you have made your `SecureString` object read-only, you will not be able to call the `Clear` method to wipe out its data. In this situation, you must either call

the Dispose method on the SecureString object or rely on the garbage collector to remove the SecureString object and its data from memory.

Notice that when you pull a SecureString object into an unsecure String, its data becomes viewable by a malicious hacker. So it may seem pointless to go through the trouble of using a SecureString when you are just going to convert it into an unsecure String. However, by using a SecureString, you narrow the window of opportunity for a malicious hacker to view this data in memory. In addition, some APIs accept a SecureString as a parameter so that you don't have to convert it to an unsecure String. The ProcessStartInfo, for example, accepts a password in its Password property as a SecureString object.



The SecureString object is not a silver bullet for securing your data. It is, however, another layer of defense you can add to your application.

See Also

See the "SecureString Class" topic in the MSDN documentation.

17.16 Securing Stream Data

Problem

You want to use the TCP server in Recipe 16.1 to communicate with the TCP client in Recipe 16.2. However, you need the communication to be secure.

Solution

Replace the NetworkStream class with the more secure SslStream class on both the client and the server. The code for the more secure TCP client, TCPClient_SSL, is shown in Example 17-19 (changes are highlighted).

Example 17-19. TCPClient_SSL class

```
class TCPClient_SSL
{
    private TcpClient _client = null;
    private IPAddress _address = IPAddress.Parse("127.0.0.1");
    private int _port = 5;
    private IPEndPoint _endPoint = null;

    public TCPClient_SSL(string address, string port)
    {
        _address = IPAddress.Parse(address);
        _port = Convert.ToInt32(port);
        _endPoint = new IPEndPoint(_address, _port);
    }
```

Example 17-19. TCPClient_SSL class (continued)

```
public void ConnectToServer(string msg)
{
    try
    {
        using (client = new TcpClient( ))
        {
            client.Connect(_endPoint);

            using (SslStream sslStream = new SslStream(_client.GetStream( ),
                            false, new RemoteCertificateValidationCallback(
                                CertificateValidationCallback)))
            {
                sslStream.AuthenticateAsClient("MyTestCert2");

                // Get the bytes to send for the message.
                byte[] bytes = Encoding.ASCII.GetBytes(msg);

                // Send message.
                Console.WriteLine("Sending message to server: " + msg);
                sslStream.Write(bytes, 0, bytes.Length);

                // Get the response.
                // Buffer to store the response bytes.
                bytes = new byte[1024];

                // Display the response.
                int bytesRead = sslStream.Read(bytes, 0, bytes.Length);
                string serverResponse = Encoding.ASCII.GetString(bytes, 0,
                      bytesRead);
                Console.WriteLine("Server said: " + serverResponse);
            }
        }
    }
    catch (SocketException e)
    {
        Console.WriteLine("There was an error talking to the server: {0}",
        e.ToString( ));
    }
}

private bool CertificateValidationCallback(object sender,
                                           X509Certificate certificate,
                                           X509Chain chain,
                                           SslPolicyErrors sslPolicyErrors)
{
    if (sslPolicyErrors == SslPolicyErrors.None)
    {
        return true;
    }
    else
    {
        if (sslPolicyErrors == SslPolicyErrors.RemoteCertificateChainErrors)
```

Example 17-19. TCPClient_SSL class (continued)

```
            {
                Console.WriteLine("The X509Chain.ChainStatus returned an array " +
                   "of X509ChainStatus objects containing error information.");
            }
            else if (sslPolicyErrors ==
                     SslPolicyErrors.RemoteCertificateNameMismatch)
            {
                 Console.WriteLine("There was a mismatch of the name " +
                   "on a certificate.");
            }
            else if (sslPolicyErrors ==
                     SslPolicyErrors.RemoteCertificateNotAvailable)
            {
                 Console.WriteLine("No certificate was available.");
            }
            else
            {
                Console.WriteLine("SSL Certificate Validation Error!");
            }
        }

        Console.WriteLine(Environment.NewLine +
                          "SSL Certificate Validation Error!");
        Console.WriteLine(sslPolicyErrors.ToString( ));

        return false;
    }
}
```

The new code for the more secure TCP server, `TCPServer_SSL`, is shown in Example 17-20 (changes are highlighted).

Example 17-20. TCPServer_SSL class

```
class TCPServer_SSL
{
    private TcpListener _listener = null;
    private IPAddress _address = IPAddress.Parse("127.0.0.1");
    private int _port = 55555;

    #region CTORs
    public TCPServer_SSL( )
    {
    }

    public TCPServer_SSL(string address, string port)
    {
        _port = Convert.ToInt32(port);
        _address = IPAddress.Parse(address);
    }
    #endregion // CTORs
```

Example 17-20. TCPServer_SSL class (continued)

```
    #region Properties
    public IPAddress Address
    {
        get { return _address; }
        set { _address = value; }
    }

    public int Port
    {
        get { return _port; }
        set { _port = value; }
    }
    #endregion

    public void Listen( )
    {
        try
        {
           _using_(listener = new TcpListener(_address, _port))
            {
                // Fire up the server.
                listener.Start( );

                // Enter the listening loop.
                while (true)
                {
                    Console.Write("Looking for someone to talk to... ");

                    // Wait for connection.
                    TcpClient newClient = _listener.AcceptTcpClient( );
                    Console.WriteLine("Connected to new client");

                    // Spin a thread to take care of the client.
                    ThreadPool.QueueUserWorkItem(new WaitCallback(ProcessClient),
                                                 newClient);
                }
            }
        }
        catch (SocketException e)
        {
            Console.WriteLine("SocketException: {0}", e);
        }
        finally
        {
            // Shut it down.
            _listener.Stop( );
        }

        Console.WriteLine("\nHit any key (where is ANYKEY?) to continue...");
        Console.Read( );
    }

    private void ProcessClient(object client)
```

Example 17-20. TCPServer_SSL class (continued)

```
    {
        using (TcpClient newClient = (TcpClient)client)
        {
            // Buffer for reading data.
            byte[] bytes = new byte[1024];
            string clientData = null;

            using (SslStream sslStream = new SslStream(newClient.GetStream( )))
            {
               sslStream.AuthenticateAsServer(GetServerCert("MyTestCert2"), false,
                                       SslProtocols.Default, true);

                // Loop to receive all the data sent by the client.
                int bytesRead = 0;
                while ((bytesRead = sslStream.Read(bytes, 0, bytes.Length)) != 0)
                {
                    // Translate data bytes to an ASCII string.
                    clientData = Encoding.ASCII.GetString(bytes, 0, bytesRead);
                    Console.WriteLine("Client says: {0}", clientData);

                    // Thank them for their input.
                    bytes = Encoding.ASCII.GetBytes("Thanks call again!");

                    // Send back a response.
                    sslStream.Write(bytes, 0, bytes.Length);
                }
            }
        }
    }

    private static X509Certificate GetServerCert(string subjectName)
    {
        X509Store store = new X509Store(StoreName.My, StoreLocation.LocalMachine);
        store.Open(OpenFlags.ReadOnly);
        X509CertificateCollection certificate =
                store.Certificates.Find(X509FindType.FindBySubjectName,
                                        subjectName, true);

        if (certificate.Count > 0)
            return (certificate[0]);
        else
            return (null);
    }
}
```

Discussion

For more information about the inner workings of the TCP server and client and how to run these applications, see Recipes 16.1 and 16.2. In this recipe, you will cover only the changes needed to convert the TCP server and client to use the `SslStream` object for secure communication.

The `SslStream` object uses the SSL protocol to provide a secure encrypted channel on which to send data. However, encryption is just one of the security features built into the `SslStream` object. Another feature of `SslStream` is that it prevents malicious or even accidental modification to the data. Even though the data is encrypted, it may become modified during transit. To determine if this has occurred, the data is signed with a hash before it is sent. When it is received, the data is rehashed and the two hashes are compared. If both hashes are equivalent, the message arrived intact; if the hashes are not equivalent, then something modified the data during transit.

The `SslStream` object also has the ability to use client and/or server certificates to authenticate the client and/or the server. These certificates are used to prove the identity of the issuer. For example, if a client attaches to a server using SSL, the server must provide a certificate to the client that is used to prove that the server is who it says it is. The `SslStream` object also allows the client to pass a certificate to the server if the client also needs to prove who it is to the server.

To allow the TCP server and client to communicate successfully, you need to set up an X.509 certificate that will be used to authenticate the TCP server. To do this, you set up a test certificate using the *makecert.exe* utility. This utility can be found in the *<drive>:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin* directory. The syntax for creating a simple certificate is as follows:

```
makecert -r -pe -n "CN=MyTestCert2" -e 01/01/2036
        -sr localMachine c:\MyAppTestCert.cer
```

The options are defined as follows:

`-r`
: The certificate will be self-signed.

`-pe`
: The certificate's private key will be exportable so that it can be included in the certificate.

`-n "CN=MyTestCert2"`
: The publisher's certificate name. The name follows the `"CN="` text.

`-e 01/01/2036`
: The date at which this certificate expires.

`-sr localMachine`
: The store where this certificate will be located. In this case, it is `localMachine`. However, you can also specify `currentUser` (which is the default if this switch is omitted).

The final argument to the *makecert.exe* utility is the output filename, in this case *c:\MyAppTestCert.cer*. This will create the certificate in the *c:\MyAppTestCert.cer* file on the hard drive.

The next step involves opening Windows Explorer and right-clicking on the *c:\MyAppTestCert.cer* file. This will display a pop-up menu with the Install Certificate

menu item. Click this menu item and a wizard will be started to allow you to import this *.cer* file into the certificate store. The first dialog box of the wizard is shown in Figure 17-2. Click the Next button to go to the next step in the wizard.

Figure 17-2. The first step of the Certificate Import Wizard

The next step in the wizard allows you to choose the certificate store in which you want to install your certificate. This dialog is shown in Figure 17-3. Keep the defaults and click the Next button.

The final step in the wizard is shown in Figure 17-4. On this dialog, click the Finish button.

After you click the Finish button, the message box shown in Figure 17-5 is displayed, warning you to verify the certificate that you wish to install. Click the Yes button to install the certificate.

Finally, the message box in Figure 17-6 is displayed, indicating that the import was successful.

At this point you can run the TCP server and client and they should communicate successfully.

To use the `SslStream` in the TCP server project, you need to create a new `SslStream` object to wrap the `TcpClient` object:

```
SslStream sslStream = new SslStream(newClient.GetStream( ));
```

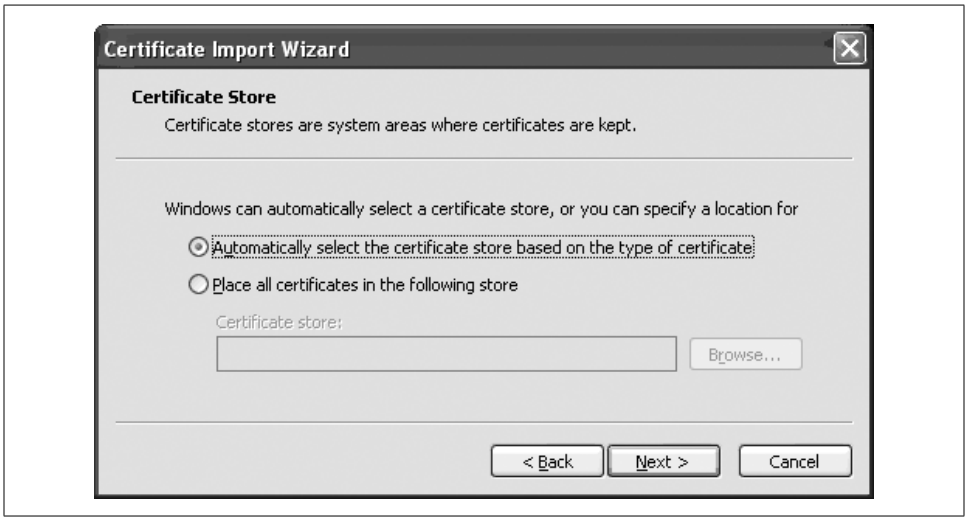


Figure 17-3. Specifying a certificate store in the Certificate Import Wizard

Figure 17-4. The last step of the Certificate Import Wizard

Before you can use this new stream object, you must authenticate the server using the following line of code:

```
sslStream.AuthenticateAsServer(GetServerCert("MyTestCert2"),
                               false, SslProtocols.Default, true);
```

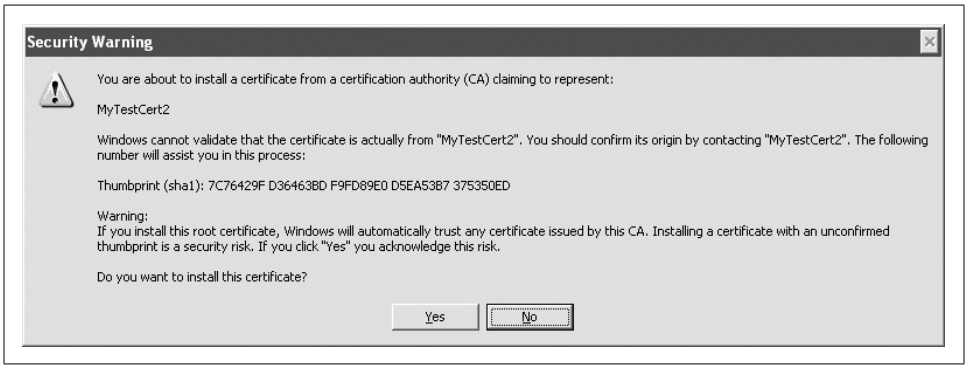


Figure 17-5. The security warning

Figure 17-6. The import successful message

The `GetServerCert` method finds the server certificate used to authenticate the server. Notice the name passed in to this method; it is the same as the publisher's certificate name switch used with the *makecert.exe* utility (see the *–n* switch). This certificate is returned from the `GetServerCert` method as an `X509Certificate` object. The next argument to the `AuthenticateAsServer` method is `false`, indicating that a client certificate is not required. The `SslProtocols.Default` argument indicates that the authentication mechanism (SSL 2.0, SSL 3.0, TLS 1.0, or PCT 1.0) is chosen based on what is available to the client and server. The final argument indicates that the certificate will be checked to see whether it has been revoked.

To use the `SslStream` in the TCP client project, you create a new `SslStream` object, a bit differently from how it was created in the TCP server project:

```
SslStream sslStream = new SslStream(_client.GetStream(), false,
      new RemoteCertificateValidationCallback(CertificateValidationCallback));
```

This constructor accepts a stream from the `_client` field, a `false` indicating that the stream associated with the `_client` field will be closed when the `Close` method of the `SslStream` object is called, and a delegate that validates the server certificate. The `CertificateValidationCallback` method is called whenever a server certificate needs to be validated. The server certificate is checked and any errors are passed into this delegate method to allow you to handle them as you wish.

The `AuthenticateAsClient` method is called next to authenticate the server:

```
sslStream.AuthenticateAsClient("MyTestCert2");
```

As you can see, with a little extra work, you can replace the current stream type you are using with the `SslStream` to gain the benefits of the SSL protocol.

See Also

See the "SslStream Class" topic in the MSDN documentation.

17.17 Encrypting web.config Information

Problem

You need to encrypt data within a *web.config* file programmatically.

Solution

To encrypt data within a *web.config* file section, use the following method:

```
public static void EncryptWebConfigData(string appPath,
                                        string protectedSection,
                                        string dataProtectionProvider)
{
    System.Configuration.Configuration webConfig =
                WebConfigurationManager.OpenWebConfiguration(appPath);
    ConfigurationSection webConfigSection = webConfig.GetSection(protectedSection);

    if (!webConfigSection.SectionInformation.IsProtected)
    {
        webConfigSection.SectionInformation.ProtectSection(dataProtectionProvider);
        webConfig.Save( );
    }
}
```

To decrypt data within a *web.config* file section, use the following method:

```
public static void DecryptWebConfigData(string appPath, string protectedSection)
{
    System.Configuration.Configuration webConfig =
                WebConfigurationManager.OpenWebConfiguration(appPath);
    ConfigurationSection webConfigSection = webConfig.GetSection(protectedSection);

    if (webConfigSection.SectionInformation.IsProtected)
    {
        webConfigSection.SectionInformation.UnprotectSection( );
        webConfig.Save( );
    }
}
```

You will need to add the `System.Web` and `System.Configuration` DLLs to your project before this code will compile.

Discussion

To encrypt data, you can call the `EncryptWebConfigData` method with the following arguments:

```
EncryptWebConfigData("/WebApplication1", "appSettings",
                     "DataProtectionConfigurationProvider");
```

The first argument is the virtual path to the web application, the second argument is the section that you want to encrypt, and the last argument is the data protection provider that you want to use to encrypt the data.

The `EncryptWebConfigData` method uses the virtual path passed into it to open the *web.config* file. This is done using the `OpenWebConfiguration` static method of the `WebConfigurationManager` class:

```
System.Configuration.Configuration webConfig =
    WebConfigurationManager.OpenWebConfiguration(appPath);
```

This method returns a `System.Configuration.Configuration` object, which you use to get the section of the *web.config* file that you wish to encrypt. This is accomplished through the `GetSection` method:

```
ConfigurationSection webConfigSection = webConfig.GetSection(protectedSection);
```

This method returns a `ConfigurationSection` object that you can use to encrypt the section. This is done through a call to the `ProtectSection` method:

```
webConfigSection.SectionInformation.ProtectSection(dataProtectionProvider);
```

The `dataProtectionProvider` argument is a string identifying which data protection provider you want to use to encrypt the section information. The two available providers are `DpapiProtectedConfigurationProvider` and `RsaProtectedConfigurationProvider`. The `DpapiProtectedConfigurationProvider` class makes use of the Data Protection API (DPAPI) to encrypt and decrypt data. The `RsaProtectedConfigurationProvider` class makes use of the `RsaCryptoServiceProvider` class in the .NET Framework to encrypt and decrypt data.

The final step to encrypting the section information is to call the `Save` method of the `System.Configuration.Configuration` object. This saves the changes to the *web.config* file. If this method is not called, the encrypted data will not be saved.

To decrypt data within a *web.config* file, you can call the `DecryptWebConfigData` method with the following parameters:

```
DecryptWebConfigData("/WebApplication1", "appSettings");
```

The first argument is the virtual path to the web application; the second argument is the section that you want to encrypt.

The `DecryptWebConfigData` method operates very similarly to the `EncryptWebConfigData` method, except that it calls the `UnprotectSection` method to decrypt the encrypted data in the *web.config* file:

```
webConfigSection.SectionInformation.UnprotectSection( );
```

If you encrypt data in the *web.config* file using this technique, the data will automatically be decrypted when the web application accesses the encrypted data in the *web.config* file.

See Also

See the "System.Configuration.Configuration Class" topic in the MSDN documentation.

17.19 Achieving Secure Unicode Encoding

Problem

You want to make sure that your `UnicodeEncoding` or `UTF8Encoding` class detects any errors, such as an invalid sequence of bytes.

Solution

Use the constructor for the `UnicodeEncoding` class that accepts three parameters:

```
UnicodeEncoding encoding = new UnicodeEncoding(false, true, true);
```

Or use the constructor for the `UTF8Encoding` class that accepts two parameters:

```
UTF8Encoding encoding = new UTF8Encoding(true, true);
```

Discussion

The final argument to both these constructors should be `true`. This turns on error detection for this class. Error detection will help when an attacker somehow is able to access and modify a Unicode- or a UTF8-encoded stream of characters. If the attacker is not careful she can invalidate the encoded stream. If error detection is turned on, it will be a first defense in catching these invalid encoded streams.

When error detection is turned on, errors such as the following are dealt with by throwing an `ArgumentException`:

- Leftover bytes that do not make up a complete encoded character sequence exist.
- An invalid encoded start character was detected. For example, a UTF8 character does not fit into one of the following classes: Single-Byte, Double-Byte, Three-Byte, Four-Byte, Five-Byte, or Six-Byte.
- Extra bits are found after processing an extra byte in a multibyte sequence.
- The leftover bytes in a sequence could not be used to create a complete character.
- A high surrogate value is not followed by a low surrogate value.
- In the case of the `GetBytes` method, the `byte[]` that is used to hold the resulting bytes is not large enough.
- In the case of the `GetChars` method, the `char[]` that is used to hold the resulting characters is not large enough.

If you use a constructor other than the one shown in this recipe or if you set the last parameter in this constructor to `false`, any errors in the encoding sequence are ignored and no exception is thrown.

See Also

See the "UnicodeEncoding Class" and "UTF8Encoding Class" topic in the MSDN documentation.

17.20 Obtaining a Safer File Handle

Problem

You want more security when manipulating an unmanaged file handle than a simple `IntPtr` can provide.

Solution

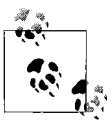
Use the `Microsoft.Win32.SafeHandles.SafeFileHandle` object to wrap an existing unmanaged file handle:

```
public static void WriteToFileHandle(IntPtr hFile)
{
    // Wrap our file handle in a safe handle wrapper object.
    using (Microsoft.Win32.SafeHandles.SafeFileHandle safeHFile =
        new Microsoft.Win32.SafeHandles.SafeFileHandle(hFile, true))
    {
        // Open a FileStream object using the passed-in safe file handle.
        using (FileStream fileStream = new FileStream(safeHFile,
               FileAccess.ReadWrite))
        {
           // Flush before we start to clear any pending unmanaged actions.
           fileStream.Flush( );

           // Operate on file here.
           string line = "Using a safe file handle object";

           // Write to the file.
           byte[] bytes = Encoding.ASCII.GetBytes(line);
           fileStream.Write(bytes,0,bytes.Length);
        }
    }
    // Note that the hFile handle is invalid at this point.
}
```

The `SafeFileHandle` constructor takes two arguments. The first is an `IntPtr` that contains a handle to an unmanaged resource. The second argument is a Boolean value, where `true` indicates that the handle will always be released during finalization and `false` indicates that the safeguards that force the handle to be released during finalization are turned off. Unless you have an extremely good reason to turn off these safeguards, it is recommended that you always set this Boolean value to `true`.

Discussion

A `SafeFileHandle` object contains a single handle to an unmanaged file resource. This class has two major benefits over using an `IntPtr` to store a handle—critical finalization and prevention of handle recycling attacks. The `SafeFileHandle` is seen by the garbage collector as a critical finalizer, due to the fact that one of the `SafeFileHandle`'s base classes is `CriticalFinalizerObject`. The garbage collector separates finalizers into two categories: critical and noncritical. The noncritical finalizers are run first, followed by the critical finalizers. If a `FileStream`'s finalizer flushes any data, it can assume that the `SafeFileHandle` object is still valid, because the `SafeFileHandle` finalizer is guaranteed to run after the `FileStream`'s.



The `Close` method on the `FileStream` object will also close its underlying `SafeFileHandle` object.

Since the `SafeFileHandle` falls under critical finalization, it means that the underlying unmanaged handle is always released (i.e., the `SafeFileHandle.ReleaseHandle` method is always called), even in situations in which the AppDomain is corrupted and/or shutting down or the thread is being aborted. This will prevent resource handle leaks.

The `SafeFileHandle` object also helps to prevent handle recycling attacks. The operating system aggressively tries to recycle handles, so it is possible to close one handle and open another soon afterward and get the same value for the new handle. One way an attacker will take advantage of this is by forcing an accessible handle to close on one thread while it is possibly still being used on another in the hope that the handle will be recycled quickly and used as a handle to a new resource, possibly one that the attacker does not have permission to access. If the application still has this original handle and is actively using it, data corruption could be an issue.

Since this class inherits from the `SafeHandleZeroOrMinusOneIsInvalid` class, a handle value of zero or minus one is considered an invalid handle.

See Also

See the "Microsoft.Win32.SafeHandles.SafeFileHandle Class" topic in the MSDN documentation.