Cryptography Application Block Hands-On Lab for Enterprise Library

Working through this lab, you will learn about the Enterprise Library Cryptography Application Block and have the chance to exercise its capabilities in various application contexts to help you understand how you can keep your secret data secret.

Keeping secrets is important and difficult. Probably the most common secrets you want to keep are connection strings, which often contain user names and passwords; however, the task of securing that data is handled by the Configuration Application Block and is discussed in the Data Access Application Block Hands-On Lab. This lab looks at how you can protect other types of data, and how you can use hashing to securely handle passwords submitted by users.

This hands-on lab includes the following two exercises:

- Lab 1: Encrypt and Decrypt Secrets
- Lab 2: Use a Hash Provider to Store a One-Way Hashed Password

After completing this lab, you will be able to do the following:

- Keep non-configuration data secret.
- Store passwords or other data in a secure manner.

The estimated completion time for all the labs is **30 minutes**.

Authors

These Hands-On Labs were produced by the following individuals:

- Product/Program Management: Grigori Melnik (Microsoft Corporation)
- Development: Chris Tavares (Microsoft Corporation), Nicolas Botto (Digit Factory), Olaf Conijn (Olaf Conijn BV), Fernando Simonazzi (Clarius Consulting), Erik Renaud (nVentive Inc.)
- Testing: Rick Carr (DCB Software Testing, Inc) plus everyone above
- Documentation: Alex Homer and RoAnn Corbisier (Microsoft Corporation) and Dennis DeWitt (Linda Werner & Associates Inc)

All of the Hands-On Labs use a simplified approach to object generation through Unity and the Enterprise Library container. The recommended approach when developing applications is to generate instances of Enterprise Library objects using dependency injection to inject instances of the required objects into your application classes, thereby realizing all of the advantages that this technique offers.

However, to simplify the examples and make it easier to see the code that uses the features of each of the Enterprise Library Application Blocks, the examples in the Hands-On Labs use the simpler approach to resolve Enterprise Library objects from the container by using the **GetInstance** method of the container service locator. You will see this demonstrated in each of the examples.

To learn more about using dependency injection to create instances of Enterprise Library objects, see the documentation installed with the Enterprise Library, or available on MSDN® at http://msdn.microsoft.com/entlib/.

Lab 1: Encrypt and Decrypt Secrets

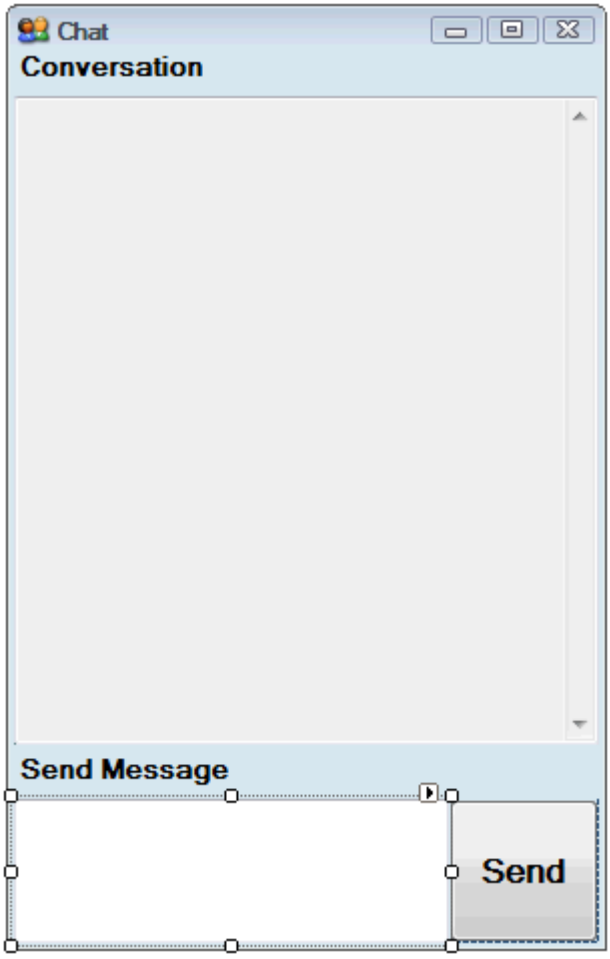
In this lab, you will look at protecting information that is not part of your configuration. You will review an example application that simulates a chat (instant messenger) application and upgrade it to communicate using encrypted strings instead of plain text strings.

To begin this exercise, open the ChatterBox.sln file located in the ex01\begin folder.

To review the application

1. Select the **Chat.cs** file in the Solution Explorer. Select the **View | Designer** menu command.

   The Chat form, shown below, is used for sending and receiving instant messages. The top, larger text box displays the conversation trace. The bottom, smaller text box is used to send new messages.

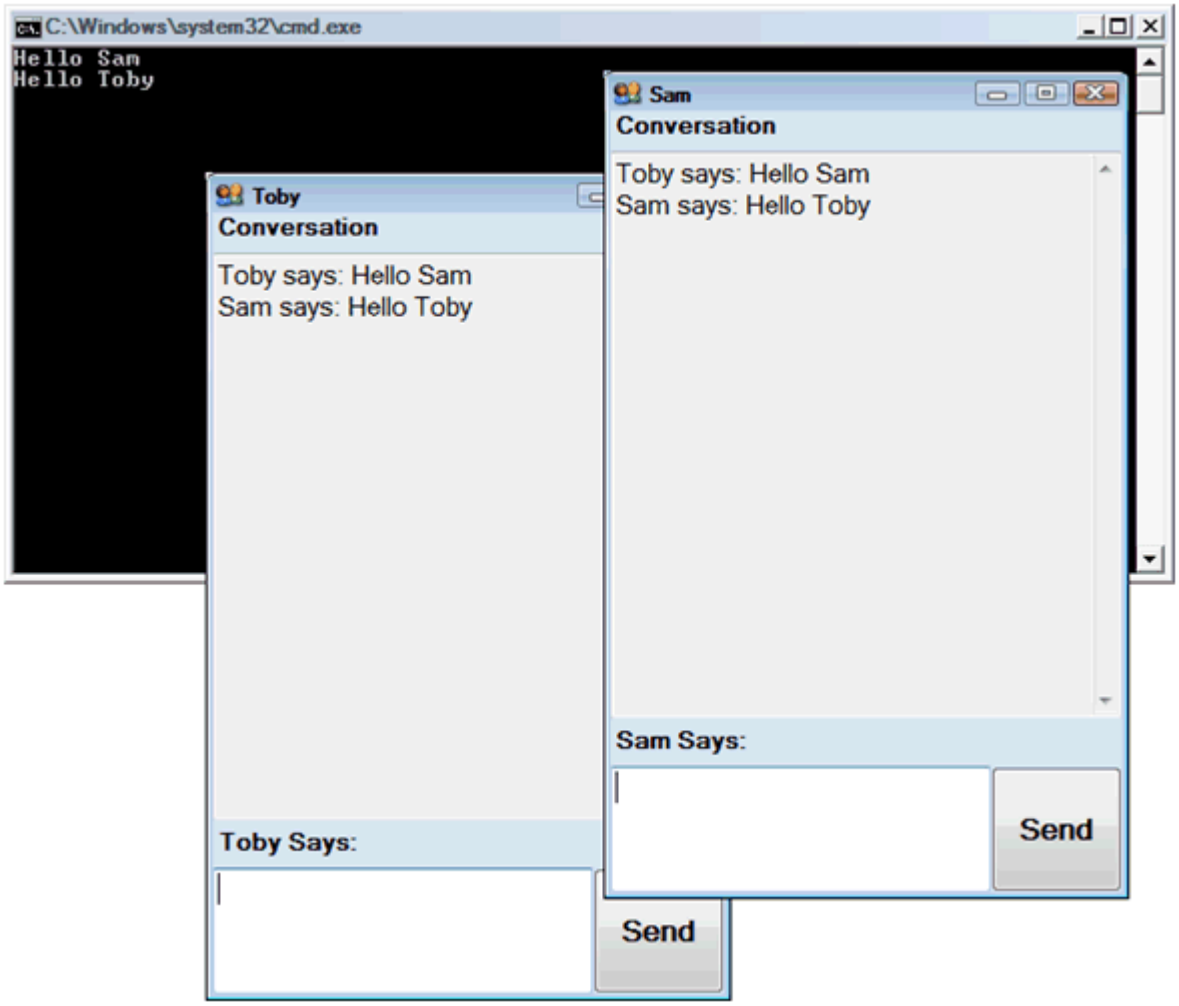


2. Select the **Debug | Start Without Debugging** menu command to run the application.

   This opens two chat windows (one is named Sam, and one is named Toby). Messages can be passed between these windows. Select the Toby window. Type some text in the **Toby Says** text box, and then click the **Send** button. Do the same in the Sam window (see screen shot below).

The conversation appears in both chat windows.

A console window is also displayed to act as a conversation monitor. Every message is displayed there.



The messages are being sent between the windows as plain text. You will use the Cryptography Application Block to encrypt and decrypt these communications using a symmetric key.

3. Close any application window to close the application.

To add encryption and decryption

1. Select the **Project | Add Reference …** menu command. Select the **Browse** tab, and then select the following assemblies located in the Enterprise Library bin folder (typically found at C:\Program Files\Microsoft Enterprise Library 5.0\Bin):
    - Microsoft.Practices.EnterpriseLibrary.Security.Cryptography.dll
    - Microsoft.Practices.EnterpriseLibrary.Common.dll

- Microsoft.Practices.ServiceLocation.dll

2. Select the **Chat.cs** file.
3. Select the **View | Code** menu command.

   Add the following namespace inclusion to the list of namespaces at the top of the file:

```
using Microsoft.Practices.EnterpriseLibrary.Security.Cryptography;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
```

4. Add the following highlighted code to the Chat class.

```
public partial class Chat : Form
{
  // TODO: Configuration symmetric algorithm provider name
  private const string symmProvider = "ChatProvider";

  ...
}
```

   The constant must match a named symmetric cryptography provider (see the symmetricCryptoProviders section in the App.config file, later in the lab).

5. Modify the **SendMessage** method to encrypt the message using the Cryptography Manager (changes in **bold**).

```
private void SendMessage(string message)
{
  // TODO: Encrypt message
  CryptographyManager crypto
    = EnterpriseLibraryContainer.Current.GetInstance<CryptographyManager>();
  string encrypted = crypto.EncryptSymmetric(symmProvider, message);

  // Raise SendingMessage Event
  if (this.SendingMessage != null)
      this.SendingMessage(new MessageEventArgs(this._name, encrypted));
}
```

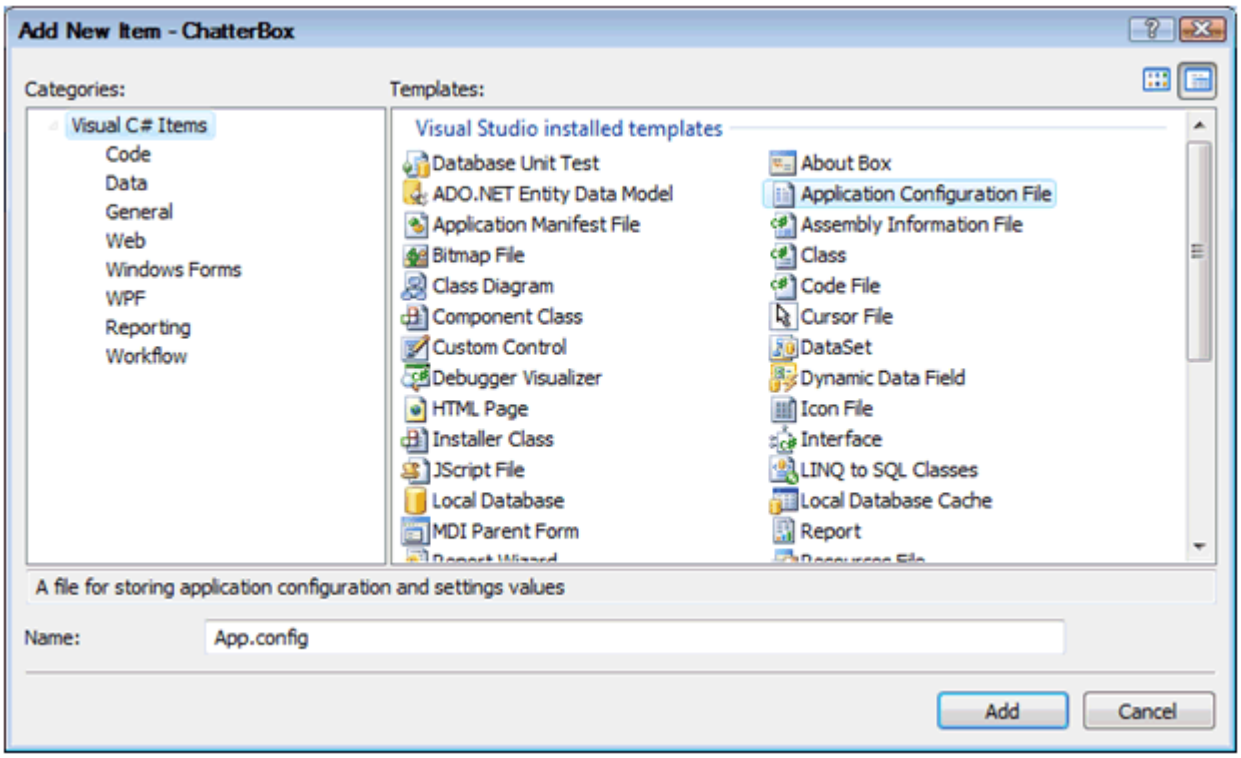
6. Modify the **MessageReceived** method to decrypt the message using the Cryptography Manager (changes in **bold**).

```
private void MessageReceived(MessageEventArgs args)
{
  string message = args.Message;

  // TODO: Decrypt message
  CryptographyManager crypto
    = EnterpriseLibraryContainer.Current.GetInstance<CryptographyManager>();
  string plainText = crypto.DecryptSymmetric(symmProvider, message);

  this.txtMessages.AppendText(
```

```
        args.Sender + " says: " + plainText + Environment.NewLine);
}
```

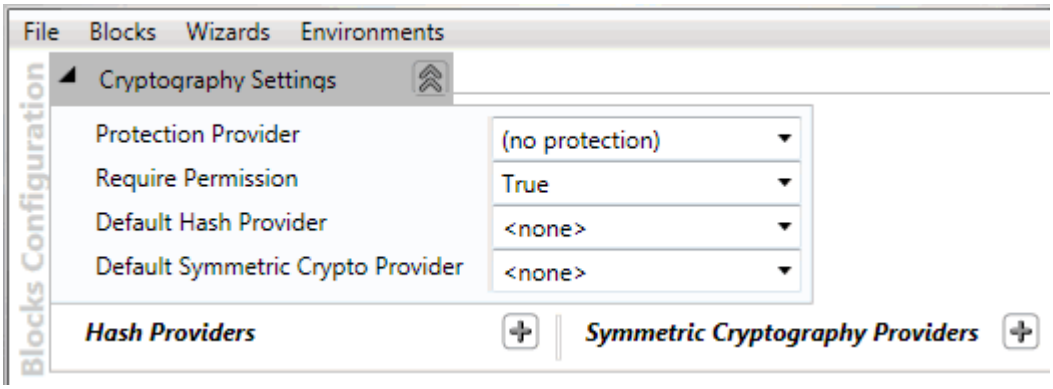
To add Enterprise Library configuration

1. Add a new application configuration file (App.config) to the ChatterBox project. In Microsoft® Visual Studio®, select the **ChatterBox** project. Select the **Project | Add New Item…** menu command. Select **Application Configuration File** and leave the Name as **App.config**, as shown below. Then click **Add**.



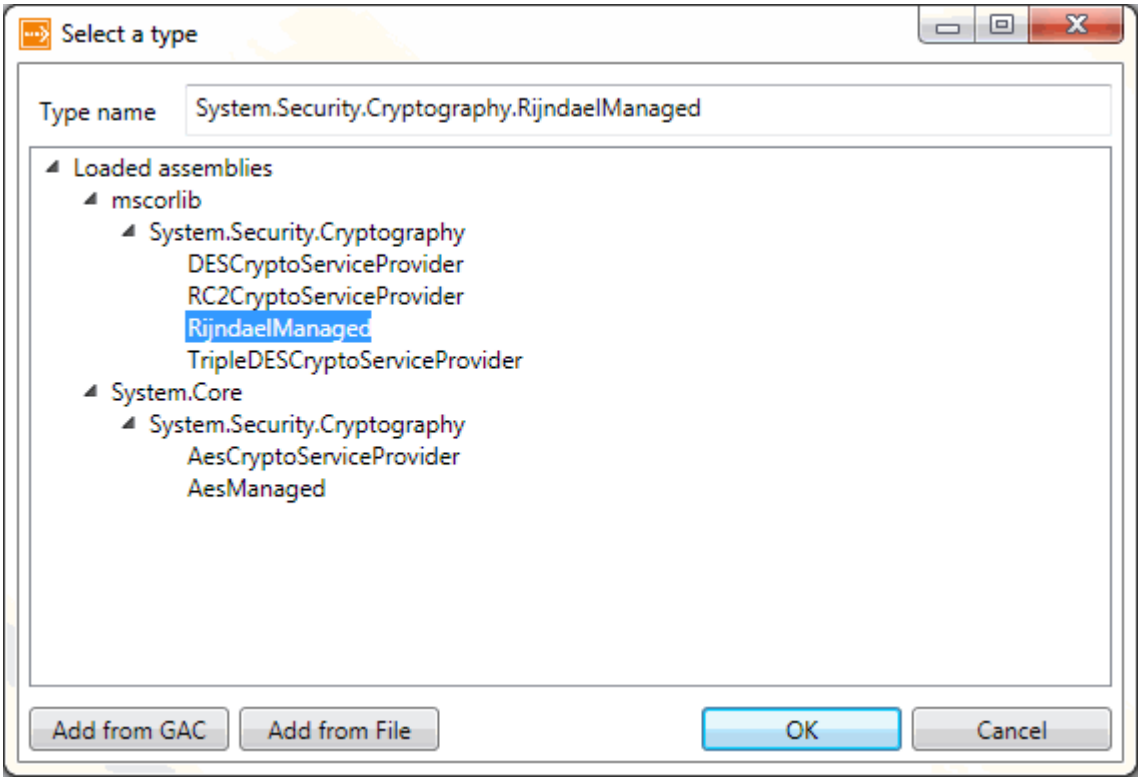
2. Use the Enterprise Library configuration tool to configure your application. Open the application configuration in the configuration editor using one of these approaches:
   - Start the appropriate version of the stand-alone tool from the Windows® Start menu (select **All Programs | Microsoft patterns & practices | Enterprise Library 5.0 | Enterprise Library Configuration**) and open the App.config file.
   - In Visual Studio, right-click on the App.config file in Solution Explorer and click **Edit Enterprise Library V5 Configuration**.
3. Click the **Blocks** menu and select **Add Cryptography Settings**. This adds a section to the tool containing two empty columns: one each for hash providers and symmetric cryptography providers.

4. Click the chevron arrow to the right of the **Cryptography Settings** title, as shown below, to show the settings for this section. These are some of the settings that you can change to tune cryptography performance for your application. For now, leave the settings as they are.



To configure the application to use a symmetric key cryptography provider

1. Click the plus-sign icon in the **Symmetric Cryptography Providers** column, point to **Add Symmetric Cryptography Providers**, and click **Add Symmetric Algorithm Provider**.
2. The type selector dialog is displayed, as shown in the following screen shot. Expand the mscorlib item until you see a list of algorithm providers. Select the **RijndaelManaged** type and then click **OK**.



An encryption algorithm provides no security if it is cracked or is vulnerable to brute force cracking. Custom algorithms are particularly vulnerable if they have not been tested.

Therefore, make sure to use published, well-known encryption algorithms that have withstood years of rigorous attacks and scrutiny.

As computing power grows, so do the recommended key lengths. Encryption key lengths that range from 128 bits to 256 bits are currently considered to be secure. Most modern algorithms use keys that are at least 128 bits long.

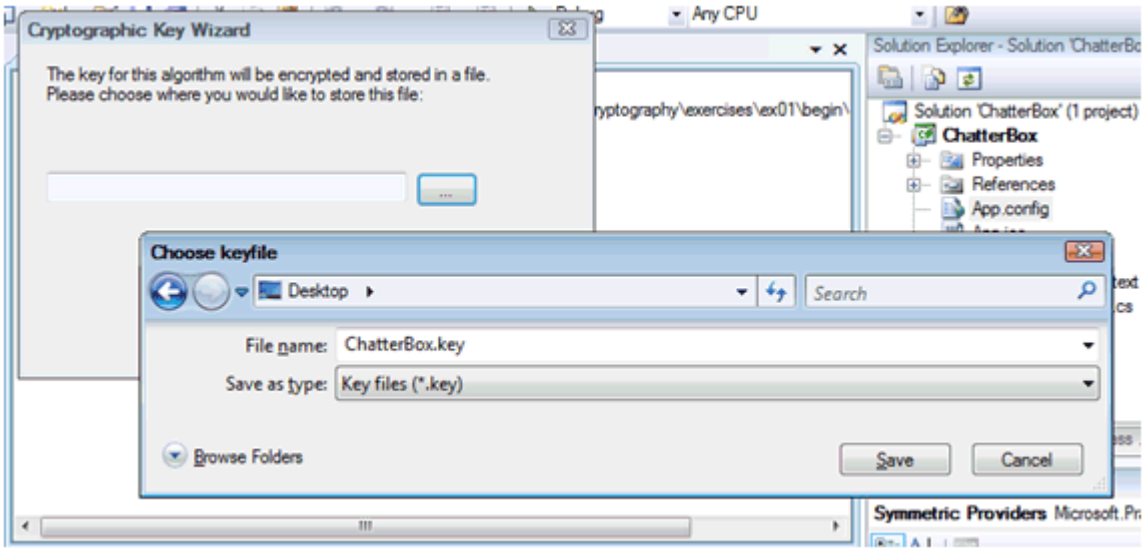
For symmetric algorithms, AES, also known as Rijndael, is recommended. This algorithm supports key lengths of 128, 192, 256 bits. The DES algorithm is not recommended.

3. The Cryptographic Key Wizard is displayed (see below). In the first screen of the wizard, select the **Create a new key** option, and then click the **Next** button.

The wizard will lead you through the process of creating and protecting a cryptographic key.

4. Click the **Generate** button to generate a new key, and then click the **Next** button, as you see here.

5. Click the ellipsis (**…**) button and choose a key file location. For this lab, the Windows Desktop may prove to be a convenient location. Select a file name (for example, ChatterBox.key, as shown below), and then click the **Next** button.



The key is no longer stored in the configuration file. Each key is stored in a separate file that is protected with the Windows data protection API (DPAPI).

6. Select **User mode** and then click **Finish**, as you see here.

When you create the key, you choose either machine mode or user mode to limit access to the key. Use machine mode in the following situations:

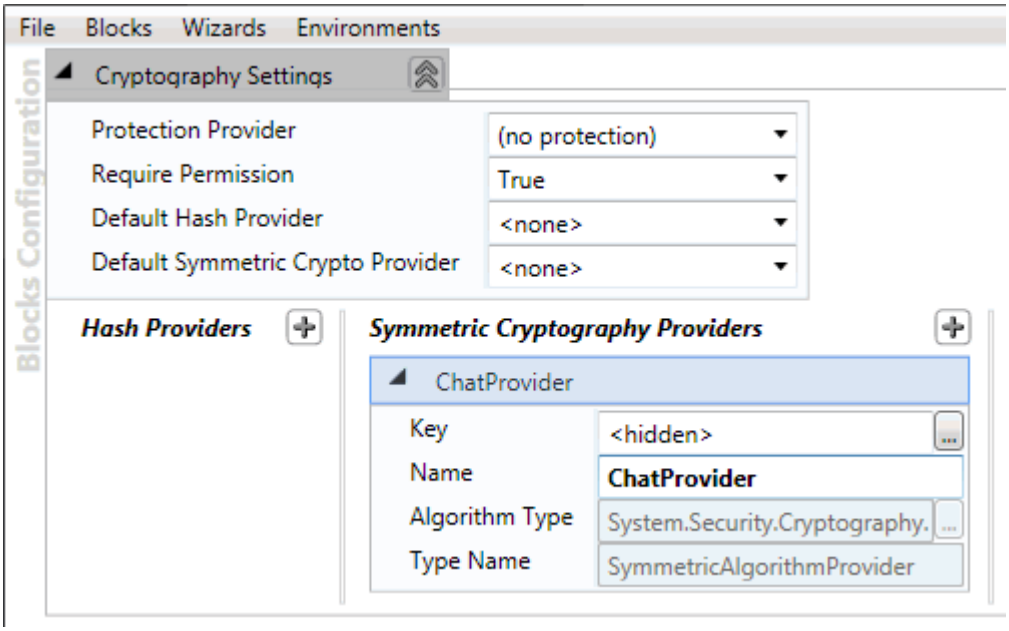
- When your application runs on its own dedicated server with no other applications.
- When you have multiple applications that run on the same server and you want those applications to be able to share sensitive information.

Choose user mode if you run your application in a shared hosting environment and you want to make sure that your application's sensitive data is not accessible to other applications on

the server. In this situation, each application should run under a separate identity, and the resources for the application, such as files and databases, should be restricted to that identity.

> If you use DPAPI with machine mode, the encrypted string is specific to a particular computer, so you must generate the encrypted data on every computer. Do not copy the encrypted data across computers that are in a server farm or cluster.

7. The **RijndaelManager** provider is added to the configuration and displayed in the configuration tool. Select the property **Name** and change the value to **ChatProvider**. The title bar of the provider item changes to show the new name, as illustrated here.



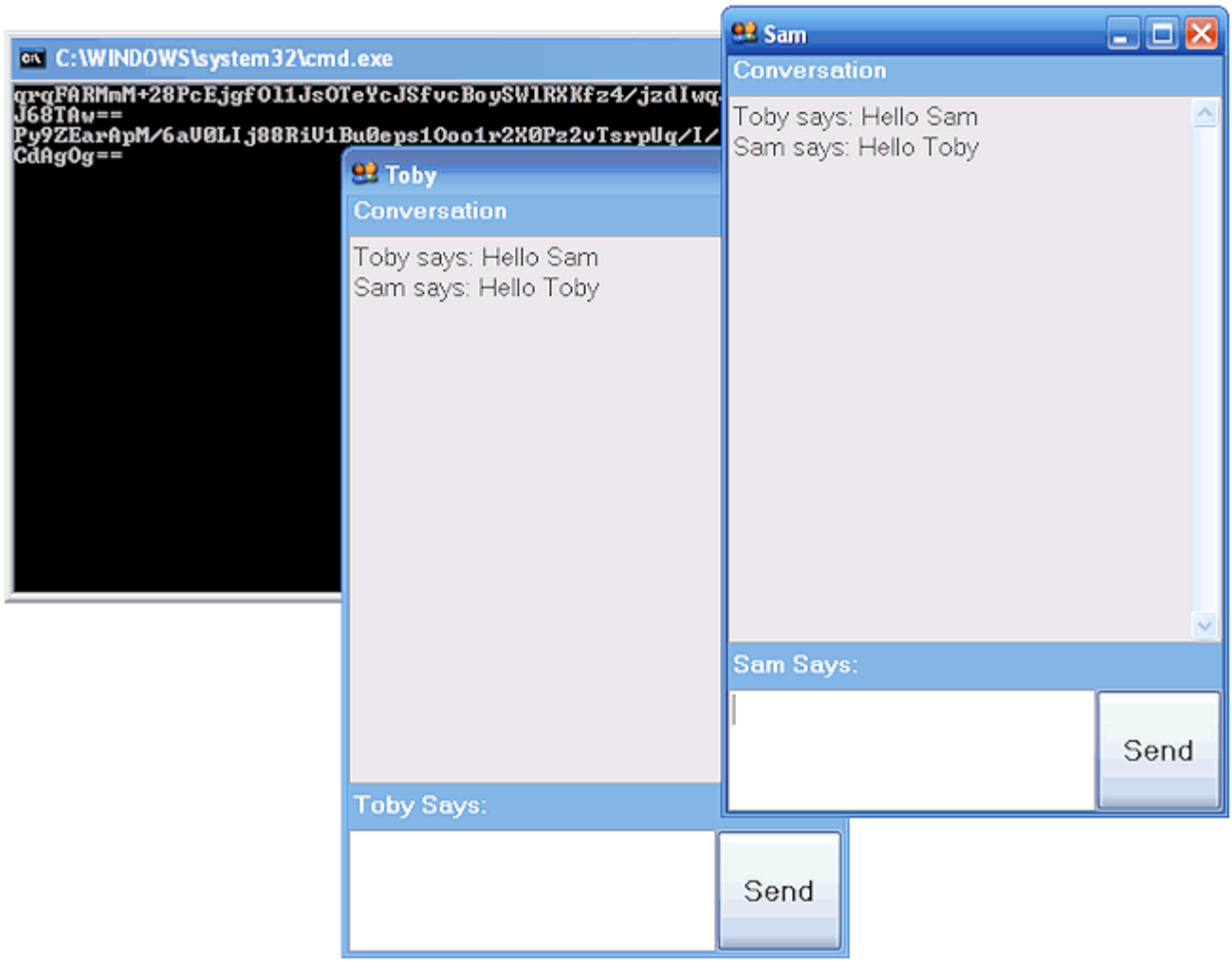
ChatProvider is the name of the symmetric provider we used earlier in the Chat.cs code.

8. Save the application configuration.

To run the application

1. Select the **Debug | Start Without Debugging** menu command.

   Pass messages between Toby and Sam. The messages are passed encrypted (see the encrypted messages in the console window), but they are decrypted by the receiver.



2. You may or may not have noticed that the application has become a little less stable now that you are using cryptography (to see what we mean, try sending an empty string). Usually, you would add some code to check the strings before you try to encrypt or decrypt them. For example, you should check for zero-length strings in both the **SendMessage** and **MessageReceived** methods. The **MessageReceived** method should also check that the strings you are about to decrypt are a multiple of four bytes long and only contain valid Base64 characters. These checks have been omitted for clarity.

3. Close the application.

To add error handling

1. The best way to handle exceptions is to make sure that they do not happen in the first place with some guard code. First, make sure that you do not try to encrypt a zero-length string.

   Select the **Chat.cs** file in the Solution Explorer. Select the **View | Code** menu command. Add the following highlighted code to the **SendMessage** method.

```
private void SendMessage(string message)
```

```
{
  if ((message != null) && (message.Trim().Length > 0))
  {
    // TODO: Encrypt message
    CryptographyManager crypto =
      EnterpriseLibraryContainer.Current.GetInstance<CryptographyManager>();

    string encrypted = crypto.EncryptSymmetric(symmProvider, message);

    // Fire SendingMessage Event
    if (this.SendingMessage != null)
        this.SendingMessage(new MessageEventArgs(this._name, encrypted));
  }
}
```

To verify that you have completed the exercise correctly, you can use the solution provided in the ex01\end folder.

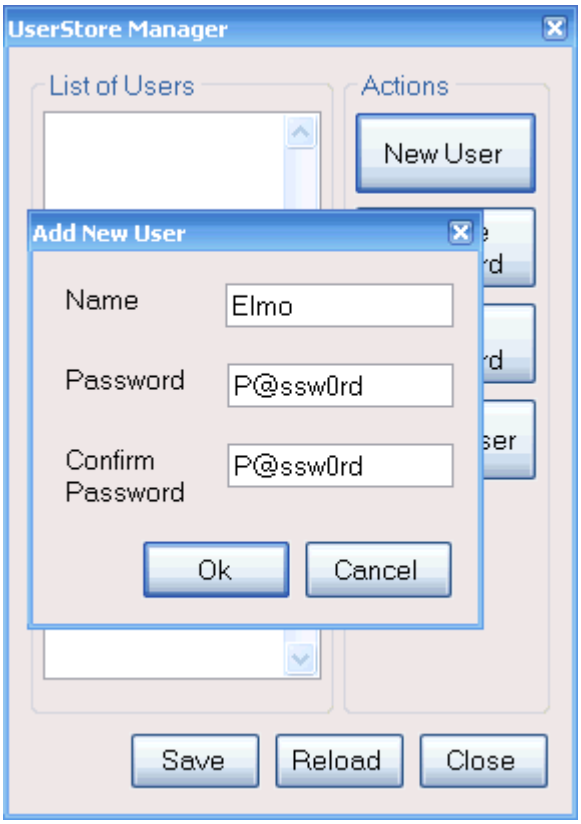
Lab 2: Use a Hash Provider to Store a One-Way Hashed Password

In this lab, you will use a one-way hashing algorithm to encrypt a password stored in an XML file.

To begin this exercise, open the UserUI.sln file located in the ex02\begin folder. The application allows you to manage user names and passwords in an XML configuration file.

To review the application

1. Select the **Debug | Start Without Debugging** menu command to run the application.
2. Add a new user named Elmo, as shown below. Click the **New User** button. Enter the name Elmo, and leave the passwords as the default (**P@ssw0rd**). Click the **Ok** button.



3. Repeat the preceding step for a new user named Zoe.
4. Click the **Save** button to save your changes to the UserStore.config file.
5. Click the **Close** button to close the application.
6. Select the **UserStore.config** (in the **UserUI** project) file in the Solution Explorer. Select the **View | Open** menu command.

   You can see the password in plain text, which is not what you want.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
```

```
    <section name="userStore"
             type="UserStore.Configuration.UserSettings, UserStore,
             Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
  </configSections>
  <userStore>
    <users>
      <add name="Elmo" password="P@ssw0rd" />
      <add name="Zoe" password="P@ssw0rd" />
    </users>
  </userStore>
</configuration>
```

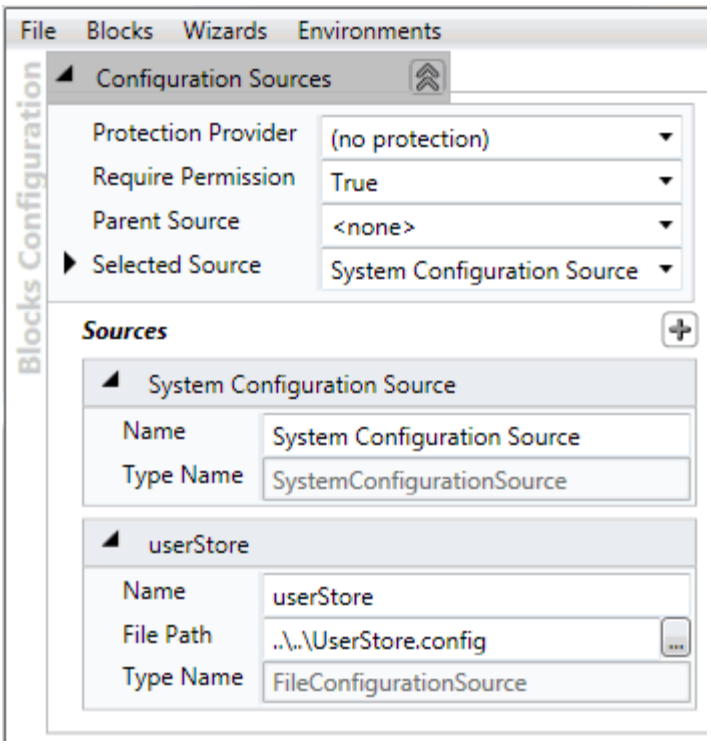
It is not necessarily a good idea to encrypt passwords using a symmetric key like the one you used in the previous exercise because if the key is compromised, so are all your passwords.

One-way hashing algorithms are usually used with passwords. You would hash the password before comparing it with a hashed password stored somewhere (usually in a database). This way, even if the database is compromised, your passwords are still safe.

Adding extra information, called Salt, to the password before hashing also makes the password much harder to crack. This is the default process in the Cryptography Application Block's hashing providers.

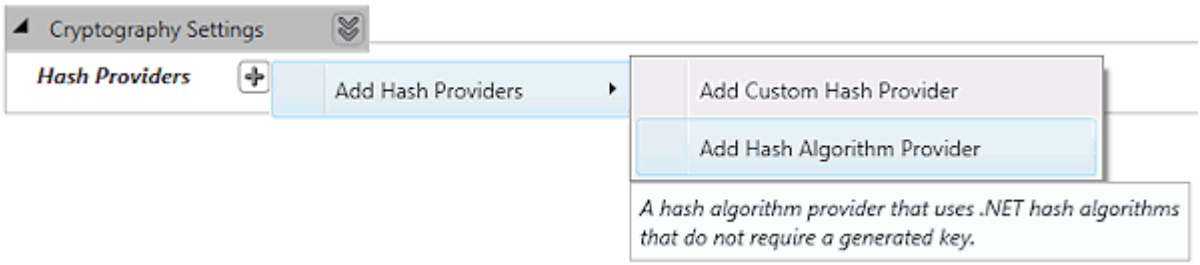
To configure the hash provider

1. Open the application configuration file located in the **UserUI** project in the configuration editor using one of these approaches:
    - Start the appropriate version of the stand-alone tool from the Windows Start menu (select **All Programs | Microsoft patterns & practices | Enterprise Library 5.0 | Enterprise Library Configuration**) and open the App.config file.
    - In Visual Studio, right-click on the App.config file in Solution Explorer and click **Edit Enterprise Library V5 Configuration**.
2. The application configuration already has two configuration sources defined (as seen below). The application is using the Enterprise Library wrapper classes for configuration to manage the UserStore.config location and contents.

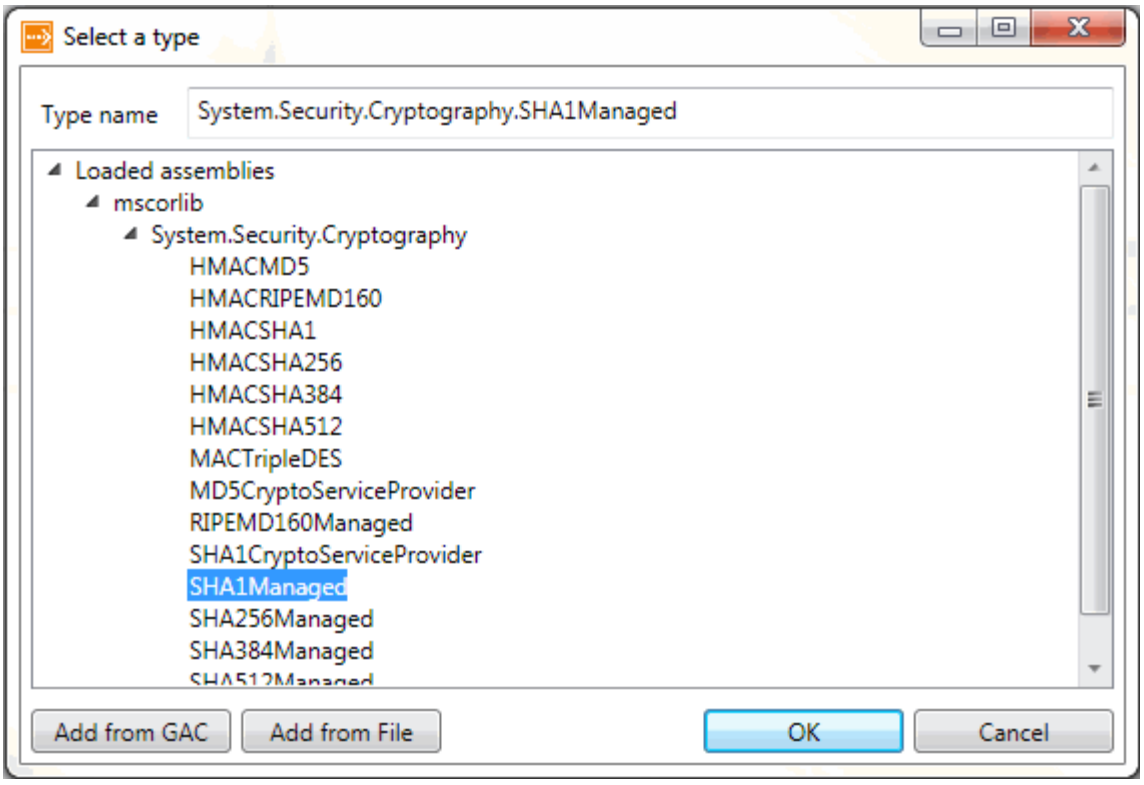


3. Open the **Blocks** menu and click **Add Cryptography Settings**.

4. Click the plus sign icon in the **Hash Providers** column, point to **Add Hash Providers**, and click **Add Hash Algorithm Provider**.

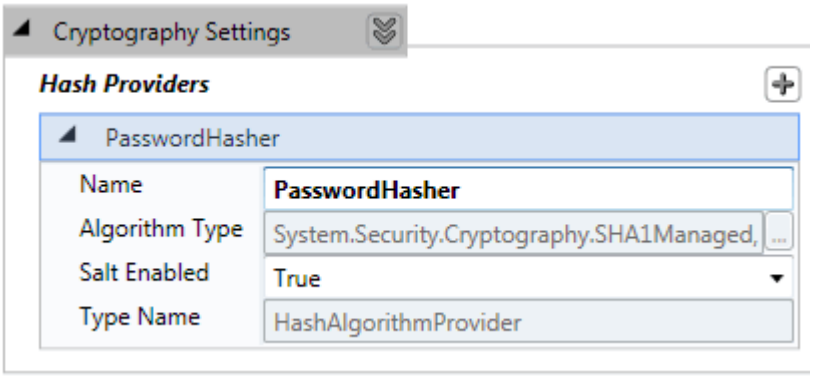


5. The type selector dialog (for hash algorithm types) is displayed. Expand the mscorlib entry until you see a list of providers, and select **SHA1Managed**, as in the image below. Then click **OK**.



For hashing algorithms, the SHA256Managed algorithm is recommended. This algorithm uses a hash size of 256 bits. The hash size of the SHA1Managed hashing algorithm is 160 bits. This algorithm is acceptable but not encouraged. The MD4 and MD5 algorithms are no longer recommended.

6. Back in the configuration tool, change the **Name** property of the hashing provider to **PasswordHasher** and ensure that the **Salt Enabled** property is set to **True**.



7. Save the application configuration.

To use the hash provider

1. Select the **UserStore** project in the Solution Explorer. Select the **Project | Add Reference …** menu command. Select the **Browse** tab and select the following assembly located in the Enterprise Library **bin** folder (typically C:\Program Files\Microsoft Enterprise Library 5.0\Bin):

- Microsoft.Practices.EnterpriseLibrary.Security.Cryptography.dll
- Microsoft.Practices.ServiceLocation.dll

2. Select the **Security | HashHelper.cs** file (in the **UserStore** project) in the Solution Explorer. Select the **View | Code** menu.

   Add the following namespace inclusions:

```
using Microsoft.Practices.EnterpriseLibrary.Security.Cryptography;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
```

3. Add the following highlighted code to the **HashHelper** class.

```
class HashHelper
{
  private HashHelper() { }

  // TODO: Hash provider name must match app.config
  private const string hashProvider = "PasswordHasher";

  ...
}
```

   The constant must match a named hash provider (see the hashProviders section in the App.config file).

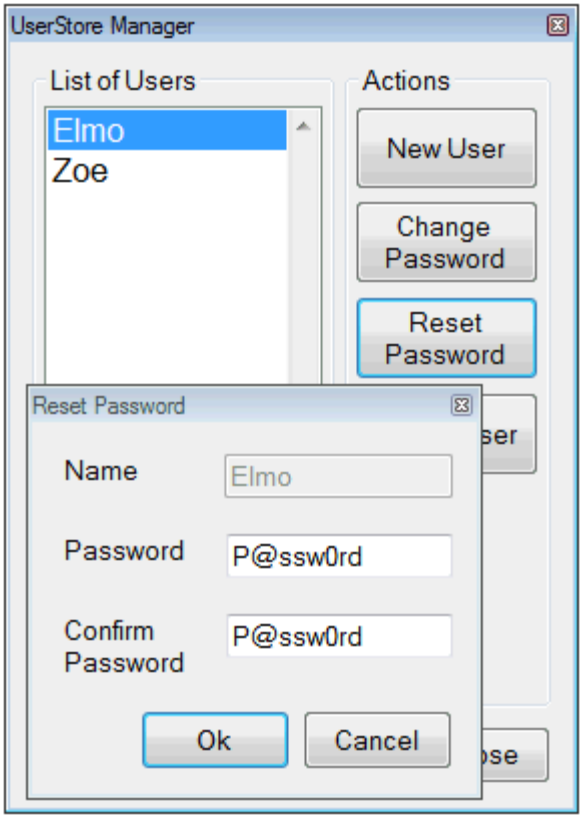
4. Modify the following code in the **CreateHash** method.

```
public static string CreateHash(string plainText)
{
  string hash = null;

  // TODO: Hash the plain text
  CryptographyManager crypto
    = EnterpriseLibraryContainer.Current.GetInstance<CryptographyManager>();
  hash = crypto.CreateHash(hashProvider, plainText);

  return hash;
}
```

   Keep in mind that sensitive data should be cleared in memory as soon as possible. Leaving sensitive data unencrypted in memory can expose the data to security risks. You should note that data in memory may also end up on the hard disk, because the operating system can write data to a swap file. Also, if the computer crashes, the operating system can dump the contents of memory to disk.

5. Select the **Debug | Start Without Debugging** menu command to run the application.

   Reset the password for **Elmo** by selecting the name in the list of users and clicking the **Reset Password** button. Leave the passwords as the default (**P@ssw0rd**). Click the **Ok** button.



This replaces the existing plain text password in the UserStore.config file with a password hash.

6. Repeat the preceding step for the user named Zoe.
7. Click the **Save** button to save your changes to the UserStore.config file.
8. Attempt to change Elmo's password, which will validate the existing password. Click the **Change Password** button. The Old Password is **P@ssw0rd** (the default). Use any new password you like (not zero length) and click the **Ok** button.

   Why does the validation of the existing password fail? We'll explain why shortly.
9. Click the **Close** button to close the application.
10. Select the **UserStore.config** (in the **UserUI** project) file in the Solution Explorer. Select the **View | Open** menu command.

    You can see that the passwords have been hashed.

    > Notice the hashing is different in each case, even though the actual password is the same. The difference results from the addition of the Salt. Therefore, to test the validity of a password, you cannot simply reapply the hashing to the plain text password and string compare to two hashed values.
11. Select the **Security | HashHelper.cs** file (in the **UserStore** project) in the Solution Explorer. Select the **View | Code** menu.

Modify the following highlighted code in the **CompareHash** method.

```
public static bool CompareHash(string plainText, string hashedText)
{
  bool compare = false;

  // TODO: Compare plain text with hash
  CryptographyManager crypto
    = EnterpriseLibraryContainer.Current.GetInstance<CryptographyManager>();
  compare = crypto.CompareHash(hashProvider, plainText, hashedText);

  return compare;
}
```

12. Select the **Debug | Start Without Debugging** menu.
13. Change Elmo's password, which will validate the existing password. Click the **Change Password** button. The Old Password is **P@ssw0rd** (the default). Use any new password you like (not zero length) and click the **Ok** button.

    The password is changed successfully.

To verify that you have completed the exercise correctly, you can use the solution provided in the ex02\end folder.

Copyright

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes

© 2010 Microsoft. All rights reserved.

Microsoft, MSDN, Visual Studio, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.