If you run the sample code, the encoded value of 1,2,3,4,5 is AQIDBAU=. In binary, 1,2,3,4,5 is represented as follows.

```
00000001|00000010|00000011|00000100|00000101
```

This value is broken up into 6-bit blocks to get the following.

```
000000|010000|001000|000011|000001|000000|0101
```

To get the first base 64 encoded character, you use the first 6 bits to look up a value. The first 6 bits is 000000, which translates to the letter A. The second character gets its value from the next 6-bit block, 010000 (16), which translates to Q. The next character gets its value from the third 6-bit block, 001000 (8), which translates to the letter I. The fourth letter comes from 000011(3), which translates to the letter D. The next letter comes from 000001(1), which translates to the letter B. The next letter comes from 000000, which translates to the letter A. The next letter comes from the last block of bits, but there are not enough bits to make this character, so additional zero bits are added to the end. Therefore, 0101(5) becomes 010100(20), which translates to the letter U. In addition, the equals sign is added to the end to signal that the quantity of input bytes is not evenly divisible by 3. The shortage of 1 byte is represented by one equals sign. If you were short by two bytes, two equals signs would be added to the end.

Symmetric Cryptography

Symmetric cryptography provides the fastest and most basic type of encryption. In this type of encryption, you use the same secret key to encrypt and decrypt data. Symmetric cryptography dates back to ancient Egypt, and the Caesar Cipher was a well-known way of performing symmetric cryptography that dates back to Julius Caesar. You've heard of the secret decoder ring; you use the same secret decoder ring to encrypt messages and decrypt messages.

Symmetric cryptography is fast. There are many symmetric algorithms, and such an algorithm is complex but efficient, which is why it's still at the core of many encrypted communication protocols.

Symmetric cryptography uses relatively small keys, usually fewer than 256 bits. What's the significance of small keys? Generally speaking, the larger the key, the stronger the encryption, but this also requires more resources to perform the encryption and decryption. To help you understand, consider the extreme: if a key is only one bit, you could break the cryptography by trying to decrypt with 0 for the key and then trying to decrypt with 1 for the key. Because of the trade-off between strength and resources, you don't usually create keys that are, for instance, one megabyte in size. This might provide super-strong encryption, but it would take a significant length of time to encrypt or decrypt a message.

The big disadvantage of symmetric cryptography is key management and distribution. If you use the same key to encrypt and decrypt data, how do you transfer the key to someone so he or she can decrypt the data you encrypted? If a malicious user intercepts the key, he or she can decrypt your messages.

Another disadvantage of symmetric cryptography is that you don't know whether the author of a message is the person from whom the message was sent or if it was intercepted by a malicious user. If an unexpected user intercepts the key, he or she can also encrypt messages in an effort to provide disinformation.

You can mitigate the disadvantages of symmetric cryptography by adding layers of security on top of symmetric cryptography.

The Federal Information Processing Standard (FIPS) 140-1 provides a means to validate vendors' cryptographic products. It provides standard implementations of several widely used cryptographic algorithms, and it judges whether a vendor's products implement the algorithms according to the standard. If you plan to sell your software to the U.S. federal government, you must use FIPS 140-1–approved algorithms. Many of the algorithms in the .NET Framework are FIPS 140-1 approved, but some, such as Data Encryption Standard (DES), are in the .NET Framework for backward compatibility and are not FIPS approved.

To protect your data and keep it secret, the protection algorithm must be public. What sounds like a paradox can be best understood by analyzing the selection process for the new Advanced Encryption Standard (AES). When the American National Institute of Standards and Technology (NIST) started looking for a successor to the widespread DES, it invited anyone to submit a more complex encryption algorithm that would be called Advanced Encryption Standard. After 15 potential algorithms were submitted, NIST made the algorithms public and invited everyone to break them. It was amazing how quickly many of these algorithms were broken, meaning that someone was able to decrypt a message without knowing the secret key. Although the submitters were sure that their algorithm couldn't be broken for at least 30 years, as required by NIST, the first candidate was broken almost immediately. Three other candidates were eliminated soon after that, and just five algorithms made it to the final selection round.

This example demonstrates that there is no better measure of security than extensive public testing. A group of technicians might not think of a way to break an algorithm, but that doesn't guarantee that at least one unconventional user somewhere in the world can't. Only time and real-life testing by as many people as possible can show whether an algorithm really serves its purpose. That's what makes encryption standards so valuable: although widely deployed and, thereby, often a target of malicious users, they have been shown to withstand these attacks. DES, for example, started its career in 1977. It took twenty years and an enormous increase in computing power before it was shown in 1997 that DES had become vulnerable.

AES is the current encryption standard (FIPS-197), intended to be used by U.S. government organizations to protect sensitive (and even secret and top secret) information. It is also becoming the unofficial global standard for commercial software and hardware that uses encryption or other security features.

On October 2, 2000, NIST selected Rijndael as the Advanced Encryption Standard (FIPS-197) and thus destined it for massive worldwide usage. The .NET Framework contains the Rijndael algorithm, which you should use for all new projects that require symmetric cryptography.

You can use the *RijndaelManaged* class to perform symmetric encryption and decryption. This class requires you to provide a key and *initialization vector* (*IV*). The IV helps ensure that encrypting the same message multiple times produces different *ciphertexts*. The encrypted message is commonly known as the *ciphertext*. To decrypt data, you must use the same key and IV you used to encrypt the data.

The key must consist of data bytes that make up the total key length. For example, a 128-bit key comprises 16 bytes. If the key is generated by the application, using bytes for the key is not a problem; however, if a human being is generating a key, it's common to want the key to consist of a word or phrase. This can be accomplished by using the *Rfc2898DeriveBytes* class, which can derive a key from a password you provide.

The following code sample demonstrates the use of the *RijndaelManaged* class to encrypt data.

Sample of Visual Basic Code

```
Dim myData = "hello"
Dim myPassword = "OpenSesame"
Dim cipherText As Byte()
Dim salt() As Byte = {&H0, &H1, &H2, &H3, &H4, &H5, &H6, &H5, &H4, &H3, &H2, &H1, &H0}

Private Sub mnuSymmetricEncryption_Click(ByVal sender As System.Object, _
     ByVal e As System.Windows.RoutedEventArgs) _
     Handles mnuSymmetricEncryption.Click

   Dim key As New Rfc2898DeriveBytes(myPassword, salt)

   ' Encrypt the data.
   Dim algorithm = New RijndaelManaged()
   algorithm.Key = key.GetBytes(16)
   algorithm.IV = key.GetBytes(16)
   Dim sourceBytes() As Byte = New System.Text.UnicodeEncoding().GetBytes(myData)
   Using sourceStream = New MemoryStream(sourceBytes)
      Using destinationStream As New MemoryStream()
         Using crypto As New CryptoStream(sourceStream, _
                                          algorithm.CreateEncryptor(), _
                                          CryptoStreamMode.Read)
            moveBytes(crypto, destinationStream)
            cipherText = destinationStream.ToArray()
         End Using
      End Using
   End Using
   MessageBox.Show(String.Format( _
                  "Data:{0}{1}Encrypted and Encoded:{2}", _
                   myData, Environment.NewLine, _
                   Convert.ToBase64String(cipherText)))
End Sub
```

```
Private Sub moveBytes(ByVal source As Stream, ByVal dest As Stream)
   Dim bytes(2048) As Byte
   Dim count = source.Read(bytes, 0, bytes.Length - 1)
   While (0 <> count)
      dest.Write(bytes, 0, count)
      count = source.Read(bytes, 0, bytes.Length - 1)
   End While
End Sub
```

Sample of C# Code

```
private string myData = "hello";
private string myPassword = "OpenSesame";
private byte[] cipherText;
private byte[] salt =
   { 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x5, 0x4, 0x3, 0x2, 0x1, 0x0 };

private void mnuSymmetricEncryption_Click(
   object sender, RoutedEventArgs e)
{
   var key = new Rfc2898DeriveBytes(myPassword, salt);

   // Encrypt the data.
   var algorithm = new RijndaelManaged();
   algorithm.Key = key.GetBytes(16);
   algorithm.IV = key.GetBytes(16);
   var sourceBytes = new System.Text.UnicodeEncoding().GetBytes(myData);
   using (var sourceStream = new MemoryStream(sourceBytes))
   using (var destinationStream = new MemoryStream())
   using (var crypto = new CryptoStream(sourceStream,
                                        algorithm.CreateEncryptor(),
                                        CryptoStreamMode.Read))
   {
      moveBytes(crypto, destinationStream);
      cipherText = destinationStream.ToArray();
   }
   MessageBox.Show(String.Format(
      "Data:{0}{1}Encrypted and Encoded:{2}",
      myData, Environment.NewLine,
      Convert.ToBase64String(cipherText)));
}

private void moveBytes(Stream source, Stream dest)
{
   byte[] bytes = new byte[2048];
   var count = source.Read(bytes, 0, bytes.Length);
   while (0 != count)
   {
      dest.Write(bytes, 0, count);
      count = source.Read(bytes, 0, bytes.Length);
   }
}
```

In this code sample, the data to be encrypted is stored in *myData*, and the result ends up in *ciphertext*. In between, *myPassword* and *salt* create a key, thanks to the *Rfc2898DeriveBytes* class. You must use the same *salt* value to decrypt. Next, the *RijndaelManaged* class is instantiated, and the *Key* and *IV* properties are populated from the *key* object. It might look like *Key* and *IV* are populated with the same value, but each time you call *key.GetBytes*, you get different bytes back, so to decrypt your message, you must perform the same sequence: First call *Key*, then call *GetBytes* to populate *Key*, and, finally, call *key.GetBytes* to populate *IV*.

The next task is to convert the data to binary so it can be encrypted, which is done using the *UnicodeEncoding* class.

Three streams are created for moving the data. The first stream is a *MemoryStream* object and represents the source that is populated with the bytes of the data. The next stream is a *CryptoStream* object, and this is where the work is accomplished. The *CryptoStream* object's constructor accepts a *Stream* object, which represents the stream to which the *CryptoStream* object will be bound (*sourceStream* in this example). The second parameter is an object that implements the *ICryptoTransform* interface, which is created by using the *algorithm* object's *CreateEncryptor* method. The third parameter is an enumeration value that indicates the type of binding to the stream that is passed in, which is *Read* or *Write*. Because the *sourceStream* object is passed in, *Read* is passed as the third parameter. In this case, you are pulling encrypted bytes from the *CryptoStream*, and the *CryptoStream* will in turn pull (*Read*) bytes from *sourceStream* until the end of the stream is reached. You could write this code to bind *CryptoStream* to *destinationStream*. To do this, you would pass the destination stream as the first parameter, and *Write* would be the third parameter. In that scenario, you would be pushing bytes into *CryptoStream*, which would encrypt the bytes and push (*Write*) them into the *destinationStream* object.

A *moveBytes* helper method copies the bytes from a source stream to a destination stream. The *crypto* stream object is passed to *moveBytes* as the source, and the *destinationStream* object is passed to *moveBytes* as the destination.

After calling *moveBytes*, the *destinationStream* object contains the encrypted data. This data is retrieved from the *destinationStream* object by calling its *ToArray* method, and the resulting bytes are stored in the *cipherText* variable.

Finally, a message is displayed containing *myData* and the encrypted and encoded *cipherText*.

Decrypting *cipherText* is almost the same as encrypting *myData*, as shown in the following code sample.

Sample of Visual Basic Code

```
Private Sub mnuSymmetricDecryption_Click(ByVal sender As System.Object, _
       ByVal e As System.Windows.RoutedEventArgs) _
       Handles mnuSymmetricDecryption.Click

   If (cipherText Is Nothing) Then
      MessageBox.Show("Encrypt Data First!")
      Return
   End If
   Dim key As New Rfc2898DeriveBytes(myPassword, salt)

   ' Try to decrypt, thus showing it can be round-tripped.
   Dim algorithm = New RijndaelManaged()
   algorithm.Key = key.GetBytes(16)
   algorithm.IV = key.GetBytes(16)
   Using sourceStream = New MemoryStream(cipherText)
      Using destinationStream As New MemoryStream()
         Using crypto As New CryptoStream(sourceStream, _
                                           algorithm.CreateDecryptor(), _
                                           CryptoStreamMode.Read)
            moveBytes(crypto, destinationStream)
            Dim decryptedBytes() As Byte = destinationStream.ToArray()
            Dim decryptedMessage = New UnicodeEncoding().GetString( _
                  decryptedBytes)
            MessageBox.Show(decryptedMessage)
         End Using
      End Using
   End Using
End Sub
```

Sample of C# Code

```
private void mnuSymmetricDecryption_Click(
   object sender, RoutedEventArgs e)
{
   if (cipherText == null)
   {
      MessageBox.Show("Encrypt Data First!");
      return;
   }

   var key = new Rfc2898DeriveBytes(myPassword, salt);

   // Try to decrypt, thus showing it can be round-tripped.
   var algorithm = new RijndaelManaged();
   algorithm.Key = key.GetBytes(16);
   algorithm.IV = key.GetBytes(16);
   using (var sourceStream = new MemoryStream(cipherText))
   using (var destinationStream = new MemoryStream())
   using (var crypto = new CryptoStream(sourceStream,
                                       algorithm.CreateDecryptor(),
                                       CryptoStreamMode.Read))
   {
      moveBytes(crypto, destinationStream);
      var decryptedBytes = destinationStream.ToArray();
      var decryptedMessage = new UnicodeEncoding().GetString(
         decryptedBytes);
      MessageBox.Show(decryptedMessage);
   }
}
```

This code sample starts by checking to see whether you encrypted the data first. If *cipherText* is *Nothing* (C# *null*), a message is displayed to indicate that you need to encrypt first, and then you exit the method.

Next, the *key* object and the *algorithm* object are created just like in the encryption sample.

After that, the streams are created: a *MemoryStream* object for the source *cipherText*, a *MemoryStream* object for the *destinationStream* object, and the *CryptoStream* object that works with the algorithm to encrypt and decrypt data. The second parameter of the *CryptoStream* object is created by calling the *algorithm.CreateDecryptor* method.

Finally, a call is made to the *moveBytes* helper method, and the *destinationStream* object contains the decrypted bytes that are read, converted to a string, and displayed.

Asymmetric Cryptography

Asymmetric cryptography provides a very secure mechanism for encrypting and decrypting data, due to its use of a pair of keys called the private and public keys. The private key is held by one entity and securely locked down. It should never be passed to another entity. The public key is the opposite; you can give the public key to anyone who requests it.

Asymmetric cryptography encrypts a message using just one of the keys (public or private), but you must use the opposite key to decrypt the message. For example, you ask the bank for its public key, and you encrypt a message using that public key. Who can decrypt the message? Only the holder of the private key, which is the bank, can decrypt the message. This offers strong protection because the private key never crosses the network, which is very different from symmetric cryptography, in which the recipient needs your key to decrypt.

As a side note, if the bank encrypts a message using its private key, who can decrypt it? Anyone can decrypt this message because the bank's public key is required to decrypt, and the bank will give the public key to anyone. This doesn't sound too useful; however, if you try to decrypt with an invalid key, or if the message is corrupted, an exception is thrown. If you can decrypt a message successfully using the bank's public key, it proves that the message came from the bank. You encrypt a message with your private key to prove your identity; this forms the basis for digital signatures, which are covered later in this lesson.

The RSA algorithm is the most common asymmetric algorithm used today. RSA was created by Rivest, Shamir, and Adleman who, at the time, were all at MIT. The latter authors published their work in 1978, and the algorithm appropriately came to be known as RSA. RSA uses exponentiation modulo, a product of two large primes, to encrypt and decrypt, performing both public key encryption and public key digital signature. Its security is based on the presumed difficulty of factoring large integers.

RSA is all about mathematics; it's a simple equation with big numbers (as opposed to a complex equation with small numbers, which more closely resembles symmetric cryptography). What you are encrypting is only a value in the equation. By default, RSA uses 1024-bit key pairs, so your value must have 128 bytes (1024 bits, 8 bits/byte).

Next comes the padding. Raw encryption, without padding, is not secure because your (128 bytes) number could be very small. Raw encryption isn't possible using the Microsoft .NET Framework; the *EncryptValue* and *DecryptValue* methods of the